The GNU Pascal Manual

Jan-Jaap van der Heijden,
Peter Gerwinski,
Frank Heckenbach,
Berend de Boer,
Dominik Freche,
Eike Lange,
Peter N Lewis,

and others

Last updated Jan 2005

for version 20050331 (GCC 2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x)

Copyright © 1988-2005 Free Software Foundation, Inc.

For GPC 20050331 (GCC 2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x)

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "GNU General Public License", "The GNU Project", "The GNU Manifesto" and "Funding for Free Software" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the sections entitled "GNU General Public License", "The GNU Project", "The GNU Manifesto" and "Funding for Free Software" and this permission notice, may be included in translations approved by the Free Software Foundation instead of in the original English.

Short Contents

Table of Contents

GNU Pascal

This manual documents how to run, install and maintain the GNU Pascal Compiler (GPC), as well as its new features and incompatibilities, and how to report bugs. It corresponds to GPC 20050331 (GCC 2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x).

Welcome to GNU Pascal . . .

. . . the free 32/64-bit Pascal compiler of the GNU Compiler Collection (GNU CC or GCC). It combines a Pascal front-end with the proven GCC back-end for code generation and optimization. Other compilers in the collection currently include compilers for the Ada, C, C++, Objective C, Chill, FORTRAN, and Java languages. Unlike utilities such as p2c, this is a true compiler, not just a converter.

This version of GPC corresponds to GCC version 2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x.

The purpose of the GNU Pascal project is to produce a Pascal compiler (called GNU Pascal or GPC) which

- combines the clarity of Pascal with powerful tools suitable for real-life programming,
- supports both the Pascal standard and the Extended Pascal standard as defined by ISO, ANSI and IEEE (ISO 7185:1990, ISO/IEC 10206:1991, ANSI/IEEE 770X3.160-1989),
- supports other Pascal standards (UCSD Pascal, Borland Pascal, parts of Borland Delphi, Mac Pascal and Pascal-SC) in so far as this serves the goal of clarity and usability,
- may be distributed under GNU license conditions, and
- can generate code for and run on any computer for which the GNU C compiler can generate code and run on.

Pascal was originally designed for teaching. GNU Pascal provides a smooth way to proceed to challenging programming tasks without learning a completely different language.

The current release implements Standard Pascal (ISO 7185, levels 0 and 1), most of Extended Pascal (ISO 10206, aiming for full compliance), is highly compatible to Borland Pascal (version 7.0), has some features for compatibility to other compilers (such as VAX Pascal, Sun Pascal, Mac Pascal, Borland Delphi and Pascal-SC).

It provides a lot of useful GNU extensions not found in other Pascal compilers, e.g. to ease the interfacing with C and other languages in a portable way, and to work with files, directories, dates and more, mostly independent of the underlying operating system.

Included units provide support for regular expressions, arithmetic with integer, rational and real numbers of unlimited size, internationalization, inter-process communication, message digests and more. Demo programs show the usage of these units and of many compiler features.

This manual contains

- an overview of some of GPC's most interesting features, see Chapter 1 [Highlights], page 5,
- a list of new features since the last release, see Chapter 2 [News], page 9,
- the GNU Pascal Frequently Asked Questions List, see Chapter 3 [FAQ], page 15,
- installation instructions, see Chapter 4 [Installation], page 27,
- a QuickStart Guide for programmers used to the Turbo Pascal/Borland Pascal compiler, see Chapter 7 [Borland Pascal], page 237,
- a list of command-line options to invoke the compiler, see Chapter 5 [Invoking GPC], page 35,
- the Programmer's Guide to GPC, describing the Pascal programming language in general and GPC specifc aspects, see Chapter 6 [Programming], page 47,
- the alphabetical GPC language reference, see Chapter 8 [Reference], page 257,
- a list of keywords and operators supported by GNU Pascal, see Chapter 9 [Keywords], page 453,
- information on how to report bugs in GNU Pascal and how to get support, see Chapter 10 [Support], page 459,
- the list of known bugs and things to do, also listing bugs fixed and features implemented recently, see Chapter 11 [To Do], page 465,

- some information for those who are interested in how GNU Pascal works internally, see Chapter 12 [Internals], page 475,
- a list of contributors which tells you who developed and is maintaining GNU Pascal, see Appendix D [Acknowledgments], page 507,
- the GNU General Public License which informs you about your rights and responsibilites when using, modifying and distributing GNU Pascal, see Appendix A [Copying], page 491,
- and other texts about Free Software and the GNU Project intended to answer questions like "what is GNU?" you might have in mind now, see Appendix F [GNU], page 517.

If you are familiar with Standard Pascal (ISO 7185) programming, you can probably just go ahead and try to compile your programs. Also, most of the ISO Extended Pascal Standard (ISO 10206) is implemented into GNU Pascal. The Extended Pascal features still missing from GPC are `qualified` module import, `protected` module export variables, set types with variable bounds, structured value initializers and expressions as subrange lower bounds.

If you are a Borland Pascal programmer, you should probably start reading the QuickStart guide from BP to GNU Pascal, see Chapter 7 [Borland Pascal], page 237. If you are curious about the new features GPC offers, you can get an idea in the overview of GPC highlights (see Chapter 1 [Highlights], page 5), and read in more detail about them in the Programmer's Guide to GPC (see Chapter 6 [Programming], page 47) and in the alphabetical GPC Language Reference (see Chapter 8 [Reference], page 257).

And, please, think about how you can contribute to the GNU Pascal project, too. Please support our work by contributing yours in form of example programs, bug reports, documentation, or even actual improvements of the compiler.

All trademarks used in this manual are properties of their respective owners.

1 Some of GPC's most interesting features.

The GNU Pascal Compiler (GPC) is, as the name says, the Pascal compiler of the GNU family (`http://www.gnu.org/software/gcc/`). This means:

- GPC is a 32/64 bit compiler,
- does not have limits like the 64 kB or 640 kB limit known from certain operating systems – even on those systems –,
- runs on all operating systems supported by GNU C, including
  - GNU Hurd (`http://www.gnu.org/software/hurd/`),
  - Linux (`http://www.linux.org`) on Intel, AMD64, Sparc, Alpha, S390, and all other supported types of hardware,
  - the BSD family: FreeBSD (`http://www.freebsd.org`), NetBSD (`http://www.netbsd.org`), OpenBSD (`http://www.openbsd.org`),
  - DOS with 32 bits, using DJGPP (`http://www.delorie.com/djgpp/`) or EMX,
  - MS-Windows 9x/NT, using CygWin (`http://cygwin.com`) or mingw (`http://www.mingw.org`) or MSYS (`http://www.mingw.org/msys.shtml`),
  - OS/2 with EMX,
  - Mac OS X,
  - MIPS-SGI-IRIX,
  - Alpha-DEC-OSF,
  - Sparc-Sun-Solaris,
  - HP/UX,

  and more (note: the runtime system only supports ASCII based systems; that includes almost all of today's systems, but a few IBM machines still use EBCDIC; on those, the compiler might run, but the runtime support might need major changes),
- can act as a native or as a cross compiler between all supported systems,
- produces highly optimized code for all these systems,
- is Free Software (`http://www.gnu.org/philosophy/free-sw.html`) (Open-Source Software (`http://www.opensource.org`)) according to the **GNU General Public License** (`http://www.gnu.org/copyleft/gpl.html`),
- is compatible to other GNU languages and tools such as GNU C and the GNU debugger.

The compiler supports the following language standards and quasi-standards:

- ISO 7185 Pascal (see Appendix E [Resources], page 513),
- most of ISO 10206 Extended Pascal,
- Borland Pascal 7.0,
- parts of Borland Delphi, Mac Pascal and Pascal-SC (PXSC).

Some highlights:

- From Standard Pascal: Many popular Pascal compilers claim to extend Standard Pascal but miss these important features.
  - Conformant array parameters – the standardized and comfortable way to pass arrays of varying size to procedures and functions. [Example (conformantdemo.pas)]
  - Passing local procedures as procedural parameters – with full access to all variables of the "parent" procedure. [Example (iteratordemo.pas)]
  - Automatic file buffers and standard '`Get`' and '`Put`' procedures. Read ahead from files without temporary variables. [Example (filebuf1demo.pas)] This allows you, for instance, to validate numeric input from text files before reading without conversion through strings. [Example (filebuf2demo.pas)]

  - True packed records and arrays. Pack 8 Booleans into 1 byte. [Example (packdemo.pas)]
  - Internal files. You don't have to worry about creating temporary file names and erasing the files later. [Example (intfiledemo.pas)]
  - Global '`goto`'. (Yes, '`goto`' has its place when it is not restricted to the current routine.) [Example (parserdemo.pas)]
  - Automatically set discriminants of variant records in '`New`'. [Example (variantdemo.pas)]
  - Sets of arbitrary size. [Example (bigsetsdemo.pas)]
- From Extended Pascal:
  - Strings of arbitrary length. [Example (stringschemademo.pas)]
  - '`ReadStr`' and '`WriteStr`'. Read from and write to strings with the full comfort of '`ReadLn`'/'`WriteLn`'. [Example (rwstringdemo.pas)]
  - System-independent date/time routines. [Example (datetimedemo.pas)]
  - Set member iteration: '`for Ch in ['A' .. 'Z', 'a' .. 'z'] do ...`' [Example (bigsetsdemo.pas)]
  - Set extensions (symmetric difference, '`Card`')
  - Generalized '`Succ`' and '`Pred`' functions (`foo := Succ (bar, 5);`).
  - Complex numbers. [Example (mandelbrot.pas)] [Example (parserdemo.pas)]
  - Exponentiation operators ('`pow`' and '`**`') for real and complex numbers.
  - Initialized variables. [Example (initvardemo.pas)]
  - Functions can return array or record values.
  - Result variables. [Example (resultvardemo.pas)]
  - Modules.
  - Non-decimal numbers in base 2 through 36: '`base#number`'.
  - '`MinReal`', '`MaxReal`', '`EpsReal`', '`MaxChar`' constants.
  - Schemata – the Pascal way to get dynamic arrays without dirty tricks. [Example (schemademo.pas)]
  - Local variables may have dynamic size. [Example (dynamicarraydemo.pas)]
  - Array Slice Access – access parts of an array as a smaller array, even on the left side of an assignment [Example (arrayslicedemo.pas)]
- Compatible to Borland Pascal 7.0 with objects (BP):
  - Supports units, objects, . . . , and makes even things like '`absolute`' variables portable. [Example (absdemo.pas)]
  - Comes with portable versions of the BP standard units with full source.
  - True network-transparent CRT unit: You can run your CRT applications locally or while being logged in remotely, without any need to worry about different terminal types. Compatible to BP's unit, but with many extensions, such as overlapping windows. [Example (crtdemo.pas)]
  - Fully functional GUI (X11) version of CRT (also completely network transparent).
  - The '`Random`' function can produce the same sequence of pseudo-random numbers as BP does – if you need that instead of the much more elaborate default algorithm.
  - Supports BP style procedural variables as well as Standard Pascal's procedural parameters. [Example (procvardemo.pas)]
  - A '`Ports`' unit lets you access CPU I/O ports on systems where this makes sense. [Example (portdemo.pas)]

  - Special compatibility features to help migrating from BP to GPC, like a '`GPC for BP`' unit which provides some of GPC's features for BP, and some routines to access sets of large memory blocks in a uniform way under GPC and BP (even in real mode). [Example (bigmemdemo.pas)]
  - Comes with a BP compatible '`binobj`' utility. [Example (binobjdemo.pas)]
- From Borland Delphi:
  - '`abstract`' object types and methods
  - '`is`' and '`as`' operators to test object type membership
  - Comments with '`//`'
  - Empty parameter lists with '`()`'
  - Assertions
  - A '`SetLength`' procedure for strings makes it unnecessary to use dirty tricks like assignments to the "zeroth character".
  - '`Initialize`' and '`Finalize`' for low-level handling of variables.
  - '`initialization`' and '`finalization`' for units.
- From Pascal-SC (PXSC):
  - User-definable operators. Add your vectors with '`+`'.
- Carefully designed GNU extensions help you to make your real-world programs portable:
  - 64-bit signed and unsigned integer types.
  - Special types guarantee compatibility to other GNU languages such as GNU C. Directives like '`{$L foo.c}`' make it easy to maintain projects written in multiple languages, e.g., including code written in other languages into Pascal programs [Example (Pascal part) (c_gpc.pas)] [Example (C part) (c_gpc_c.c)],
  - or including Pascal code into programs written in other languages. [Example (Pascal part) (gpc_c_pas.pas)] [Example (Pascal unit) (gpc_c_unit.pas)] [Example (C part) (gpc_c_c.c)]
  - Extensions like '`BitSizeOf`' and '`ConvertFromBigEndian`' help you to deal with different data sizes and endianesses. [Example (endiandemo.pas)]
  - Little somethings like '`DirSeparator`', '`PathSeparator`', '`GetTempDirectory`' help you to write programs that look and feel "at home" on all operating systems.
  - The '`PExecute`' routine lets you execute child processes in a portable way that takes full advantage of multitasking environments. [Example (pexecutedemo.pas)] [Example (pexec2demo.pas)]
  - The GNU GetOpt routines give you comfortable access to Unix-style short and long command-line options with and without arguments. [Example (getoptdemo.pas)]
  - Routines like '`FSplit`' or '`FSearch`' or '`FExpand`' know about the specifics of the various different operating systems. [Example (fexpanddemo.pas)]
  - The '`FormatTime`' function lets you format date and time values, according to various formatting rules. [Example (formattimedemo.pas)]
- Useful and portable GNU standard units:
  - A '`Pipes`' unit gives you inter-process communication even under plain DOS. [Example (pipedemo.pas)] [Demo process for the example (demoproc.pas)]
  - With the '`RegEx`' unit you can do searches with *regular expressions*. [Example (regexdemo.pas)]
  - The GNU MultiPrecision ('`GMP`') unit allows you to do arithmetics with integer, real, and rational numbers of arbitrary precision. [Example: factorial (factorial.pas)] [Example: fibonacci (fibonacci.pas)] [Example: power (power.pas)] [Example: real power (realpower.pas)] [Example: pi (pi.pas)]

- Posix functions like '`ReadDir`', '`StatFS`' or '`FileLock`' provide an efficient, easy-to-use and portable interface to the operating system. [Example (readdirdemo.pas)] [Example (statfsdemo.pas)] [Example (filelockdemo.pas)]
- A '`DosUnix`' unit compensates for some of the incompatibilities between two families of operating systems. [Example (dosunixdemo.pas)]
- An '`MD5`' unit to compute MD5 message digests, according to RFC 1321. [Example (md5demo.pas)]
- A '`FileUtils`' unit which provides some higher-level file and directory handling routines. [Example (findfilesdemo.pas)]
- A '`StringUtils`' unit which provides some higher-level string handling routines. [Example (stringhashdemo.pas)]
- An '`Intl`' unit for internationalization. [Example (gettextdemo.pas)] [Example (localedemo.pas)]
- A '`Trap`' unit to trap runtime errors and handle them within your program. [Example (trapdemo.pas)]
- A '`TFDD`' unit that provides some tricks with text files, e.g. a "tee" file which causes everything written to it to be written to two other files. [Example (tfdddemo.pas)]
- A '`HeapMon`' unit to help you find memory leaks in your programs.

The demo programs mentioned above are available both on the WWW and in GPC source and binary distributions.

Disadvantages:

- The GNU debugger (GDB) still has some problems with Pascal debug info.
- Compilation with GPC takes quite long.

Co-workers welcome!

Able, committed programmers are always welcome in the GNU Pascal team. If you want to be independent of companies that you have to pay in order to get a compiler with more restrictive licensing conditions that only runs on *one* operating system, be invited to join the development team, Appendix D [Acknowledgments], page 507.

2 New Features of GNU Pascal.

GPC's new or changed features since the last (non alpha/beta) GPC release are listed here. Items without further description refer to new routines, variables or options.

Features implemented for compatibility to other compilers are marked with, e.g., ‘(B)’ for BP compatibility.

A few old and obsolete features have been dropped or replaced by cleaner, more flexible or otherwise more useful ones. This might lead to minor problems with old code, but we suppose they're rare and easy to overcome. Backward-incompatible changes are marked with ‘(@)’.

- new options ‘--[no]-nonlocal-exit’ (default in ‘--ucsd-pascal’ and ‘--mac-pascal’) (fjf988[fijk].pas, fjf1062*.pas) (U) (M)
- new options ‘--[no]-object-checking’ (on by default) to check on virtual method calls if VMT pointer is not nil and has matchin ‘Size’ and ‘NegatedSize’ entries, new options ‘--[no]-range-and-object-checking’ which are now equivalent to ‘$R[+-]’ (fjf1053*.pas) (B)
- new options ‘--[no]-pointer-checking’, ‘--[no]-pointer-checking-user-defined’; new variable ‘ValidatePointerPtr’ (fjf1052*.pas, pcerrorc.pas) (these options are *off* by default!)
- new options ‘--[no]-implicit-result’ (fjf1022*.pas) (on by default **only** in ‘--delphi’ mode!) (D) (@)
- new options ‘--{en,dis}able-predefined-identifier’ (fjf1037*.pas)
- ‘import’ within a routine (mod15[c-e].pas)
- the option ‘--no-debug-info’ was renamed to ‘--disable-debug-info’, and ‘--no-default-paths’ was renamed to ‘--disable-default-paths’ (@)
- qualified identifiers (chief18.pas, fjf260*.pas, fjf921*.pas, grp1.pas, kurzw1.pas, mod{9,10,13..17}*.pas) (@)
- with ‘-Wnested-comments’ and without ‘--nested-comments’ warn about comment openers found within comments (fjf1017*.pas)
- new options ‘--[no-]case-value-checking’ (fjf1012*.pas)
- optimize arithmetic with a complex and a real operand <20020118143553.B28837@artax.karlin.mff.cuni.cz>
- range checking <Pine.LNX.4.21.0012091605570.8168-100000@rusty.russwhit.com> (chuck5.pas, fjf446.pas, fjf989*.pas, fjf992*.pas, fsc{01..37}*.pas, miklos1[bc].pas, mir{016,021..028,030..047}*.pas, ole[12]*.pas, russ4*.pas)
- ‘Exit’ with an argument (non-local exits not yet supported) (fjf988*.pas) (U) (M)
- new options ‘--[no-]propagate-units’ (on by default with ‘--mac-pascal’, off in other dialects) (fjf987*.pas) (M); this also covers the item “unit inheritance”
- enable ‘Pointer’ in ‘--mac-pascal’ mode (Mac Pascal has a ‘Pointer’ function which does the same as a type-cast to ‘Pointer’; though adding ‘Pointer’ as a type allows more, it's backward-compatible) (M)
- ‘&’ and ‘|’ (shortcut ‘and’ and ‘or’) (fjf981*.pas) (M)
- ‘Leave’ and ‘Cycle’ (equivalent to ‘Break’ and ‘Continue’) (avo3.pas) (M)
- optimize ‘WriteLn (... *string_constant*)’ and ‘Write (... *string_constant*, *string_constant* ...)’
- ‘BindingType’ is now a packed record as EP demands (fjf975a.pas) (E)
- EP structured initializers (fjf964*.pas, fjf967*.pas, fjf968*.pas) (E)
- EP record, array, set values (constdef.pas, fjf966*.pas, fjf971*.pas) (E)
- ‘gp’: ‘PC’ now sets the compiler for both Pascal and C unless ‘CC’ is set explicitly

- 'Discard'
- 'Integer', 'Word', 'Cardinal' are now equivalent to 'PtrInt', 'PtrWord', 'PtrCard', no more (necessarily) to C's 'int' and 'unsigned int' (@)
- new types 'CInteger', 'CWord', 'CCardinal'
- new make variable 'GPC_PAGESIZE' to set the page size when building the manual (PDF, PostScript, DVI)
- 'qualified' and import lists are no more allowed after 'uses' (only after 'import', as EP demands) (@)
- the 'GMP' unit doesn't support gmp-2.x anymore (if you used it, just upgrade to a newer GMP version) (@)
- conflicts between object fields/methods and ancestor type names are detected as required by OOE (fjf945*.pas) (@) (O)
- repeated function headings (in 'forward' declarations and interfaces) are checked stricter: if one has a result variable, so must the other (according to the OOE draft) (@) (O)
- the 'Pipe' unit was renamed to 'Pipes' because of a name conflict (@)
- empty parameter lists can be written as '()' (chief54*.pas, delphi6*.pas) (D)
- GMP unit: 'mpf_sin', 'mpf_cos'
- the test suite output is now by default stored in DejaGnu compatible files 'gpc.log' and 'gpc.sum' in the 'p/test/' directory; other available test targets are 'pascal.check-short' and 'pascal.check-long' (@)
- new options '-W[no-]dynamic-arrays' (fjf931*.pas)
- new argument to '_p_initialize' (@)
- 'UMask'
- new option '--no-debug-source'
- new lexer (no directly user-visible difference, but should allow for better handling of lexer-based problems in the future)
- 'CompilerAssert' (fjf904*.pas)
- options '--[no]-assert' renamed to '--[no]-assertions' (necessary to avoid a conflict with GCC) (@)
- new options '--[no-]range-checking', also as short compiler directives '{$R+}'/'{$R-}' (default is on) (C, E, B, @)
- new options '--[no-]methods-always-virtual' (fjf903*.pas) (M)
- new options '--[no-]pointer-arithmetic', '--[no-]cstrings-as-strings', '-W[no-]absolute' (all of which '--[no-]extended-syntax' implies)
- 'Integer2StringBase', 'Integer2StringBaseExt'
- new constants 'NumericBaseDigits', 'NumericBaseDigitsUpper'
- allow assigning, passing by value and returning objects, with assignments of an object of derived type to one of a base type (chief35[ab].pas, fjf451*.pas, fjf696[ef].pas, fjf884*.pas), BP compatible except for a bug in the BP feature itself (see the comment in 'p/test/fjf451h.pas') (B)
- new options '-W[no-]object-assignment'
- warn (except in '--borland-pascal') if a virtual method overrides a non-virtual one (chief52*.pas)
- warn when an non-abstract object type has virtual methods, but no constructor (chief51*.pas)
- '--maximum-field-alignment' does not apply to 'packed' records
- 'ArcSin', 'ArcCos'

- trimming string relations as functions (‘EQPad’ etc.) (fjf873.pas)
- new options ‘-W[no-]interface-file-name’
- ‘SeekEOF’ and ‘SeekEOLn’ use ‘Input’ implicitly when no file is given (fjf871.pas) (B)
- tagging feature for ‘with’ statements (tom6.pas) <200012022215.eB2MFD614424@wsinpa16.win.tue.nl> (Sun Pascal)
- new option ‘--sun-pascal’
- field names and array indices in initalizers are recognized (waldek5*.pas) (options ‘-W[no-]field-name-problem’ removed, @)
- object directives ‘published’, ‘public’ (both equivalent), ‘protected’ (scope limited to object type and derived object types), ‘private’ (scope limited to current unit/module) (fjf864*.pas) (options ‘-W[no-]object-directives’ removed, @)
- the operator precedence and associativity of ‘+’ and ‘-’ is now as defined in EP by default (and as in BP with ‘--borland-pascal’) <Pine.LNX.4.44.0210251249500.6181-100000@duch.mimuw.edu.pl> (fjf863*.pas) (@)
- ‘Integer (16)’ etc. changed to ‘Integer attribute (Size = 16)’ (works for integer and Boolean types) (fjf861.pas) (@)
- types can have attributes (note: no preceding ‘;’) (fjf860*.pas)
- dynamic object methods (fjf859.pas) (B)
- in ‘--borland-pascal’ mode, ‘Assign’ unconditionally (re-)initializes its file parameter (fjf858.pas) (B)
- GPC doesn’t use ‘gpm’ files anymore (instead, each module has an implicit ‘*modulename*-all.gpi’ interface which is a regular ‘gpi’ file)
- make the program/module/unit finalizers non-public (‘static’ in C sense), omit them if easily possible
- new options ‘-W[no-]parentheses’ (fjf848*.pas)
- non-‘interface’ modules with empty implementation part (pmod1.pas, fjf843.pas)
- ‘maximum-field-alignment’ and ‘[no-]field-widths’ work as local compiler directives now (fjf842.pas)
- dropped ‘{$debug-statement}’ (should not be necessary anymore, now that debug info mostly works)
- new options ‘--[no-]longjmp-all-nonlocal-labels’
- object methods can have attributes (fjf826*.pas)
- new attributes ‘iocritical’ (fjf824*.pas), ‘ignorable’ (fjf839*.pas) for routines
- dropped computed ‘goto’ (never worked for nonlocal ‘goto’ into the main program, implementing it would be quite difficult, probably not worth it) (@)
- new type ‘AnyFile’ for parameters and pointer targets (fjf821*.pas)
- ‘TimeStamp’ is now a packed record as EP demands (fjf975b.pas) (E)
- Mac Pascal specific features are supported according to the dialect options (M)
- ‘--interface-only’ does not require ‘-S’ or ‘-c’ anymore (and does not create an object file)
- ‘initialization’, ‘finalization’ (D)
- ‘TimeZone’ in ‘TimeStamp’ counts seconds *east* of UTC now (not west, as before) (datetimedemo.pas) (@)
- ‘export *Foo* = all (...)’ (fjf811*.pas)
- new options ‘-W[no-]local-external’ (implied by ‘-Wall’)
- type-casts are BP compatible now, in particular, value type-casts between ordinal and real or complex types don’t work anymore (B) (@)

- all non-ISO-7185 keywords can be used as identifiers (with two small exceptions) (fjf440.pas)
- ‘pack-struct’ does not imply bit-level packing anymore (only explicit ‘packed’ records and arrays do) (@)
- new options ‘--[no-]ignore-packed’ (‘--ignore-packed’ is the default in BP mode) (fjf796*.pas) (B) (@)
- new option ‘--maximum-field-alignment=*N*’
- new options ‘{$[no-]pack-struct}’ as a compiler directive
- ‘attribute’ for routines doesn’t imply ‘forward’ anymore (so you don’t have to declare routines twice in a program or implementation part when setting the linker name or some other attribute) (@)
- ‘static’, ‘volatile’ and ‘register’ for variables and ‘inline’ for routines are no prefix-directives anymore, but ‘attribute’s (@)
- combining several dialect options (such as ‘--extended-pascal --borland-pascal’) doesn’t work anymore (what should this mean, anyway? Combine the features, but also the warnings from both!?!?!?) (@)
- ‘external’ without ‘name’ defaults to all-lowercase now (@)
- ‘c’, ‘c_language’ and ‘asmname’ are deprecated (@)
- ‘external name 'foo'’ (fjf780.pas), ‘external 'libname' name 'foo'’ (where ‘libname’ is ignored) (B)
- Mac Pascal directives ‘definec’, ‘macro’, ‘undefc’, ‘ifc’, ‘ifoptc’, ‘elsec’, ‘elifc’, ‘endc’, ‘errorc’ (treated as equivalent to the corresponding existing ones) (M)
- units without ‘implementation’ part (M)
- new options ‘--vax-pascal’, ‘--mac-pascal’
- attributes ‘const’ for variables and ‘name’ for variables, routines and modules; assembler names and module/unit file names can now be expressions (which must yield string constants) (fjf781*.pas, fjf809*.pas)
- the utilities ‘gpidump’ and ‘binobj’ are installed with GPC (B)
- new options ‘-W[no-]identifier-case’, ‘-W[no-]identifier-case-local’ (fjf751*.pas)
- new compiler directive ‘$R foo’, equivalent to ‘$L foo.resource’ (B)
- dropped ‘--[no-]borland-char-constants’ (now simply coupled to dialect options) (@)
- test suite: support progress messages (‘TEST_RUN_FLAGS=-p’ from the Makefile; ‘-p’ in testgpc); see http://fjf.gnu.de/misc/progress-messages.tar.gz
- ‘=’ and ‘<>’ comparisons of structures (arrays, records, ...) except strings and sets are forbidden now (@) (E)
- irrelevant operands and arguments (e.g.: ‘foo in []’; ‘bar * []’; ‘Im (baz)’ if ‘baz’ is of real type) are not necessarily evaluated anymore (which is allowed by the standard); instead, a warning is given if they have side-effects (@)
- accept only one program, unit, module interface or implementation or a module interface and the implementation of the same module in one file; new options ‘--[no-]ignore-garbage-after-dot’ (fjf735*.pas) (@)
- new options ‘-W[no-]implicit-io’ (fjf734*.pas)
- new options ‘--enable-keyword’, ‘--disable-keyword’ (fjf733*.pas)
- ‘CBoolean’ (fjf727.pas)
- dropped the usage of ‘GetMem’ as a function with one parameter (only the BP compatible usage as a procedure with two parameters remains) (@)
- accessing the variable ‘FileMode’ now requires using the ‘GPC’ (or, for BP compatibility, the ‘System’) unit (@)

- 'DupHandle'
- dropped the predefined dialect symbols '__CLASSIC_PASCAL__', '__STANDARD_PASCAL__', '__EXTENDED_PASCAL__', '__OBJECT_PASCAL__', '__UCSD_PASCAL__', '__BORLAND_PASCAL__', '__DELPHI__', '__PASCAL_SC__' and '__GNU_PASCAL__' (one can use '{$ifopt borland-pascal}' etc. instead) (@)
- 'Succ', 'Pred', 'Inc', 'Dec' for real numbers (fjf714*.pas)
- use environment variables 'GPC_UNIT_PATH', 'GPC_OBJECT_PATH'
- new options '-W[no-]float-equal'
- new option '--ucsd-pascal'
- dropped the syntax 'type foo = procedure (Integer, Real)' (i.e., without parameter names) (@)
- CRT: new argument 'On' to 'CRTSavePreviousScreen'
- 'SetUserID', 'SetGroupID'
- 'HeapChecking'
- new built-in procedure 'Assert'; new options '--[no]-assert' (also '{$C+}', '{$C-}' for Delphi compatibility) (fjf665*.pas) (D)
- 'ProcessGroup'
- StringUtils: 'QuoteEnum'
- 'CurrentRoutineName' (fjf752.pas)
- TFDD: new unit
- gpc-run: new options '-e FILE' and '-E FILE' (redirect/append standard error)

Have fun,

The GNU Pascal Development Team

3 The GNU Pascal Frequently Asked Questions List.

This is the Frequently Asked Questions List (FAQ) for GNU Pascal. If the FAQ and the documentation do not help you, you have detected a **bug** in it which should be reported, Section 10.1 [Mailing List], page 459. Please really do it, so we can improve the documentation.

3.1 GNU Pascal

3.1.1 What and why?

The purpose of the GNU Pascal project is to produce a Pascal compiler (called GNU Pascal or GPC) which

- combines the clarity of Pascal with powerful tools suitable for real-life programming,
- supports both the Pascal standard and the Extended Pascal standard as defined by ISO, ANSI and IEEE (ISO 7185:1990, ISO/IEC 10206:1991, ANSI/IEEE 770X3.160-1989),
- supports other Pascal standards (UCSD Pascal, Borland Pascal, parts of Borland Delphi, Mac Pascal and Pascal-SC) in so far as this serves the goal of clarity and usability,
- may be distributed under GNU license conditions, and
- can generate code for and run on any computer for which the GNU C compiler can generate code and run on.

Pascal was originally designed for teaching. GNU Pascal provides a smooth way to proceed to challenging programming tasks without learning a completely different language.

The current release implements Standard Pascal (ISO 7185, levels 0 and 1), most of Extended Pascal (ISO 10206, aiming for full compliance), is highly compatible to Borland Pascal (version 7.0), has some features for compatibility to other compilers (such as VAX Pascal, Sun Pascal, Mac Pascal, Borland Delphi and Pascal-SC).

It provides a lot of useful GNU extensions not found in other Pascal compilers, e.g. to ease the interfacing with C and other languages in a portable way, and to work with files, directories, dates and more, mostly independent of the underlying operating system.

Included units provide support for regular expressions, arithmetic with integer, rational and real numbers of unlimited size, internationalization, inter-process communication, message digests and more. Demo programs show the usage of these units and of many compiler features.

3.1.2 What is the current version?

The current version is 20050331.

Releases are available as a source archive and precompiled binaries for several common platforms from the GPC web site, `http://www.gnu-pascal.de`.

For details about new features, see the section '`News`' on the web site. On bugs fixed recently, see the '`Done`' section of the To-Do list (on the same web site).

GPC uses GCC as a back-end. It supports GCC version 2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x. (The newest supported GCC version is usually preferable, unless it contains serious bugs in itself.)

There are no fixed time frames for new releases. Releases are made when enough interesting changes have been made and the compiler is somewhat stable.

3.1.3 Is it compatible with Turbo Pascal (R)?

GPC is not a drop-in replacement for Borland's Turbo Pascal (R). Almost all BP language features are supported. Notable exceptions are the string format (as discussed below), or the '`Mem`' and '`Port`' pseudo arrays, though replacement functions for the latter on IA32 platforms exist in the '`Ports`' unit.

Almost all of BP's run time library is supported in GPC, either by built-in compiler features or in units with the same names as their BP counterparts.

For details about the compatibility, the few remaining incompatibilities and some useful alternatives to BP features, see the '`Borland Pascal`' chapter in the GPC Manual. (see Chapter 7 [Borland Pascal], page 237)

3.1.4 Which platforms are supported by GNU Pascal?

GPC uses the GCC backend, so it should run on any system that is supported by GNU CC. This includes a large variety of Unix systems, MS-DOS, OS/2 and Win32. A full list of platforms supported by GCC can be found in the file '`INSTALL`' of the GCC distribution. Not all of these have actually been tested, but it is known to run on these platforms:

ix86-gnu	(GNU Hurd)
ix86-linux	(Linux 2.x, ELF)
Linux/AMD64	
i486-linuxaout	
i486-linuxoldld	
i386-freebsd1.2.0	
AIX 4.2.1	
AIX 4.3	
DJGPP V2	(Dos)
EMX 0.9B	(OS/2, Dos)
Cygwin32 beta20 and higher	(MS-Windows95/98, MS-Windows NT)
mingw32	(MS-Windows95/98, MS-Windows NT)
MSYS	(MS-Windows)
mips-sgi-irix5.3	
mips-sgi-irix6.5	
sun-sparc-sunos4.1.4	
sparc-sun-solaris2.x	
sun-sparc-solaris 2.5.1	
sun-sparc-solaris 2.6	
sun-sparc-solaris 7	
sun-sparc-solaris 8	
alpha-unknown-linux	
alpha-dec-osf4.0b	
s390-ibm-linux-gnu	

OK people – send us your success stories, with canonical machine name!

3.2 Installing GPC

You find the most up-to-date installation instructions in the GPC Manual or the file '`INSTALL`' in source distributions, or on the GPC web site. (see Chapter 4 [Installation], page 27)

The following sections describe things you might need or want to install besides GPC itself.

3.2.1 What to read next

After installing GPC, please check the files in the directory '`/usr/local/doc/gpc`':

'`README`'	General Information about GPC
'`FAQ`'	This FAQ :–)
'`NEWS`'	Changes since the last release
'`BUGS`'	How to report bugs, about the Test Suite
'`AUTHORS`'	List of GPC authors
'`COPYING`'	The GNU General Public License
'`COPYING.LIB`'	The GNU Lesser General Public License

3.2.2 Which components do I need to compile Pascal code?

A complete Pascal compiler system should at least have:

1. The actual compiler, GPC.
2. An editor, assembler, linker, librarian and friends.
3. A C library. If you have a working C compiler, you already have this.
4. A debugger, if you want to debug your programs.

For most people, the GNU binutils and GNU debugger ('`gdb`') are a good choice, although some may prefer to use vendor specific tools.

3.2.3 How do I debug my Pascal programs?

To debug your programs, (a) GNU Pascal must be able to generate executables with debug info for your platform, and (b) you must have a debugger which understands this.

- If '`gpc -g -o hello hello.p`' says:

  ```
  gpc: -g not supported for this platform
  ```

  then GPC is unable to generate debugging info. Usually, installing '`gas`' (part of GNU binutils) instead of your system's assembler can overcome this. When you configure the GCC used for GPC, specify '`--with-gnu-as`', and possibly '`--with-gnu-ld`' and/or '`--with-stabs`'. More information can be found in the '`INSTALL`' file in the GNU CC source directory.
- Your system's debugger may not understand the debug info generated by GNU tools. In this case, installing '`gdb`' may help.

The bottom line: if you can debug GCC compiled programs, you should be able to do this with GPC too.

The GNU debugger ('`gdb`') currently does not have a "Pascal" mode, so it is unable to display certain Pascal structures etc. When debugging, please note that the Initial Letter In Each Identifier Is In Upper Case And The Rest Are In Lower Case. If you want to display variable '`foo`' in the debugger, type '`show Foo`' or '`display Foo`' instead.

Although '`gdb`' is an excellent debugger, it's user interface is not everybody's preference. If you like to debug under X11, please refer to the comp.windows.x FAQ: "Where can I get an X-based debugger?" at:
`http://www.faqs.org/faqs/x-faq/part6/section-2.html`

Some useful frontends include: XXGDB, tGDB and XWPE. See:
`http://www.ee.ryerson.ca:8080/~elf/xapps/Q-IV.html`

Very nice, but resource consuming is the Motif based DDD:
`http://sol.ibr.cs.tu-bs.de/softech/ddd/`

Furthermore, RHIDE (see Section 3.2.6 [IDE], page 19) contains built-in debugging suport, similar to the IDE of BP.

3.2.4 What additional libraries should I have?

You will need certain additional libraries when you compile some of the units. These can be found in the directory `http://www.gnu-pascal.de/libs/`.

Currently, there are the following libraries:

gmp
: Arithmetic for integers, rationals and real numbers with arbitrary size and precision. Used by the GMP unit.

rx
: Regular expression matching and substitution. Used by the RegEx unit.

ncurses
PDCurses
: Screen handling. Used by the CRT unit. Depending on your system, you have the following choices:

  Unix: You can compile terminal applications with ncurses and applications that run in an X11 window with PDCurses (though terminal applications can, of course, also run in an xterm under X11). ncurses is used by default. If you want to use PDCurses (a.k.a. XCurses), give the option '`-DX11`' when compiling CRT.

  Dos with DJGPP and MS-Windows with mingw: Only PDCurses is available and will be used by default.

  MS-Windows with Cygwin: PDCurses and ncurses are available. PDCurses is used by default. If you want to use ncurses, give the option '`-DUSE_NCURSES`' when compiling CRT.

  Other systems: Please see the '`README`'s and installation instructions of PDCurses and ncurses to find out which one(s) can be built on your system. See the conditionals at the end of crt.inc and crtc.h (and change them if necessary) on which library is used by default.

intl
: Internationalization. Used by the Intl unit. On some systems, it is part of the system library (libc).

ElectricFence
: This library is not used by any GPC unit. It is a debugging tool to assist you in finding memory allocation bugs. To use it, just link it to your program, either on the command line ('`-lefence`') or in the source code ('`{$L efence}`') which you might want to put into an '`{$ifdef DEBUG}`' or similar since using libefence is only recommended for debugging.

The source code of the libraries is available in the main '`libs`' directory. Most libraries come with one or several patches which should be applied before compiling them.

Binaries for some platforms are available in the '`binary/`*platform*' subdirectories. If you compile the libraries for other platforms, be invited to make the binaries available to us for distribution on the web site.

There are also the following files:

'`terminfo-linux.tar.gz`'
: This is a patch to enable ncurses programs to make use of the ability of Linux 2.2 and newer kernels to produce a block cursor when needed. The present patch can be installed without recompiling anything, just by copying some files into place. More details can be found in the '`README`' file included in this archive. The patch will not do any harm on older kernels. Please note that **not** only on Linux machines it is useful to install the patch. Installing them on any other machine will allow users who telnet in from a Linux console to profit from the block cursor capability. Besides, some Unix systems have installed older Linux terminfo entries or none at all, so it's a good thing, anyway, to give them a current version. The patch is included in the terminfo database of ncurses 5.0, so if you install ncurses 5.0 (source or binary), you

don't need to get the patch separately. But you can install it on a system with an older ncurses version if you don't feel like upgrading ncurses altogether.

'tsort-2.9i.zip'
: A little utility (extracted from util-linux-2.9i, but not Linux specific), needed for the configuration of the rx library. You need it only if you compile rx yourself (and if it's not already present on your system), not when using a rx binary.

3.2.5 Contributed units

Several people have contributed units for GPC. They are usually announced on the mailing list, Section 10.1 [Mailing List], page 459. Most of them can be found in `http://www.gnu-pascal.de/contrib/`.

3.2.6 Can you recommend an IDE?

Users of Borland Pascal may wonder if there's a replacement for the IDE (Integrated Development Environment). Here's a few suggestions:

- (X)Emacs. Some people think it's the answer to the question of Life, the Universe and Everything, others decide it's uGNUsable. Available from your friendly GNU mirror and most distributions.
- PENG. It's not free software, but it was written with GPC. It's very similar to Borland's IDE, but with many extensions. Binaries for DJGPP, Linux and Solaris can be downloaded from `http://fjf.gnu.de/peng/`.
- RHIDE. DJGPP users might want to try RHIDE. The latest (beta) release is compatible with GNU Pascal and allows stepping, tracing and watching like Borland's IDE. It can be downloaded from `http://www.rhide.com`.
- DevPascal. DevPascal is a Free Software IDE for mingw32. It can be downloaded from `http://www.gnu-pascal.de/contrib/chief/` or `http://www.bloodshed.net/devpascal.html`
- XWPE is another imitation of the Borland IDE, so users of Borland Pascal may find it a good alternative.

3.3 GNU Pascal on the DJGPP (MS-DOS) platform

This chapter discusses some potential problems with GNU Pascal on MS-DOS, using DJGPP.

3.3.1 What is DJGPP?

The following paragraph is from the site `http://www.delorie.com/djgpp/`:

DJGPP is a complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running DOS. It includes ports of many GNU development utilities. The development tools require a 80386 or newer computer to run, as do the programs they produce. In most cases, the programs it produces can be sold commercially without license or royalties.

3.3.2 If you need more information

GPC/DJGPP is a DJGPP V2 application, and most of the DJGPP documentation applies for GPC too. A great source of information is the DJGPP FAQ: `http://www.delorie.com/djgpp/v2faq/230b.zip`

Another place to look for DJGPP documentation is the DJGPP Knowledge Base, at this URL: `http://www.delorie.com/djgpp/doc/kb/`

3.3.3 What do I download?

As discussed in Section 3.2.2 [Components], page 17, other than GPC itself, you need an assembler, linker and friends, a C library and possibly a debugger. The site `http://www.delorie.com/djgpp/` recommended the following files and they will help you find a mirror:

'v2/djdev203.zip'	(C library)
'v2gnu/bnu2951b.zip'	(assembler, . . .)
'v2gnu/gcc2952b.zip'	(gcc)
'v2gnu/gdb418b.zip'	(debugger)
'v2gnu/mak379b.zip'	(make)
'v2gnu/txi40b.zip'	(texi)

This list is about 10 MB not counting GPC. You can use a binary version of GPC from the web site.

3.3.4 How do I install the compiler?

If you don't have DJGPP installed on your harddisk, create a directory for GNU Pascal ('`c:\gpc`'), and unzip the archives. Make sure you preserve the directory structure (use '`pkunzip -d`'). Now, add the directory where '`gpc.exe`' lives ('`c:\gpc\bin`') to your path and set the DJGPP environment variable to point to your '`djgpp.env`' file:

```
set DJGPP=c:\gpc\djgpp.env
```

Then, add this to your '`djgpp.env`' file:

```
---------------------------------------------------------
[gpcpp]
C_INCLUDE_PATH=%/>;C_INCLUDE_PATH%%DJDIR%/lang/pascal;%DJDIR%/include

[gpc]
COMPILER_PATH=%/>;COMPILER_PATH%%DJDIR%/bin
LIBRARY_PATH=%/>;LIBRARY_PATH%%DJDIR%/lib;%DJDIR%/contrib/grx20/lib
---------------------------------------------------------
```

The GPC online documentation is in GNU info format; you need the Info reader ('`txi390b.zip`') to read it, or use the built-in Info reader of the RHIDE or PENG IDE. To add the GPC documentation to the info directory file, edit the '`c:\gpc\info\dir`' file, and locate this section:

```
---------------------------------------------------------
* GCC: (gcc.inf).
The GNU C, C++, and Objective-C Compiler

* GDB: (gdb.inf).
The GNU Debugger (gdb and gdb-dpmi).

---------------------------------------------------------
```

To add GPC, change it to look like this:

```
---------------------------------------------------------
* GCC: (gcc.inf).
The GNU C, C++, and Objective-C Compiler

* GPC: (gpc.inf).
The GNU Pascal Compiler
```

```
* GDB: (gdb.inf).
The GNU Debugger (gdb and gdb-dpmi).

---------------------------------------------------------
```

Specific information for low-memory conditions and more can be found in the DJGPP FAQ and documentation.

3.3.5 I cannot read the Info documentation!

To read the Info documentation, you need the 'info' program from 'txi390b.zip' or an IDE like RHIDE or PENG.

3.3.6 GPC says: no DPMI

You don't have a DPMI server installed, and DJGPP v2 requires it to run. You can either use one of the commercial DPMI servers (e.g., run 'gpc' in a DOS box under MS-Windows) or download and install CWSDPMI ('csdpmi3b.zip') which is a free DPMI server written for DJGPP.

3.3.7 I have troubles with assembly code

The GNU Assembler ('as.exe'), or 'gas', called by GCC accepts "AT&T" syntax which is different from "Intel" syntax. Differences are discussed in section 17.1 of the DJGPP FAQ.

A guide is available which was written by Brennan Mr. Wacko Underwood brennan@mack.rt66.com and describes how to use inline assembly programming with DJGPP, at this URL: http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

There's also a GPC assembler tutorial at
http://www.gnu-pascal.de/contrib/misc/gpcasm.zip

Section 17.3 of the DJGPP FAQ discusses some methods to convert "Intel" syntax to "AT&T" syntax.

However, please note that assembler code is unportable, i.e. it will work on IA32 ("x86") and compatible processors if written for them, but will not even compile for other processors. So by writing assembler code in your programs, you will limit their usefulness substantially.

If you think you "need" assembler code for speed – and you've checked that your assembler code actually runs faster than Pascal code compiled with suitable optimizations – you might want to put both Pascal and assembler versions of the critical sections in your program, and let, e.g., an '{$ifdef i386}' decide which one to use. This way, your program will at least compile on all processors.

3.3.8 Tell me how to do DPMI, BIOS and other DOS related things.

DPMI, BIOS and other functions are no different than other system functions. Refer to the GPC Manual on how to access your system's C-library. This small example shows how to use DPMI, copying some structures and function prototypes of '<dpmi.h>':

```
program DPMIDemo;

{ Only for DJGPP }

{$X+}
```

```
{ ‘Byte’ is ‘unsigned char’ in C,
  ‘ShortCard’ is ‘unsigned short’ in C,
  ‘MedCard’ is ‘unsigned long’ in C,
  ‘Word’ is ‘unsigned’ in C,
  etc. (all these types are built-in). }

type
  TDpmiVersionRet = record
    Major     : Byte;
    Minor     : Byte;
    Flags     : ShortCard;
    CPU       : Byte;
    Master_PIC: Byte;
    Slave_PIC : Byte;
  end;

type
  TDpmiFreeMemInfo = record
    LargestAvailableFreeBlockInBytes,
    MaximumUnlockedPageAllocationInPages,
    MaximumLockedPageAllocationInPages,
    LinearAddressSpaceSizeInPages,
    TotalNumberOfUnlockedPages,
    TotalNumberOfFreePages,
    TotalNumberOfPhysicalPages,
    FreeLinearAddressSpaceInPages,
    SizeOfPagingFilePartitionInPages,
    Reserved1,
    Reserved2,
    Reserved3: MedCard;
  end;

function DpmiGetVersion (var Version: TDpmiVersionRet): Integer;
         external name '__dpmi_get_version';

function DpmiGetFreeMemoryInformation
         (var MemInfo: TDpmiFreeMemInfo): Integer;
         external name '__dpmi_get_free_memory_information';

var
  Version: TDpmiVersionRet;
  MemInfo: TDpmiFreeMemInfo;

begin
  if DpmiGetVersion (Version) = 0 then
    begin
      WriteLn ('CPU type:       ', Version.CPU, '86');
      WriteLn ('DPMI major:       ', Version.Major);
      WriteLn ('DPMI minor:       ', Version.Minor);
    end
  else
```

```
      WriteLn ('Error in DpmiGetVersion');
    if DpmiGetFreeMemoryInformation (MemInfo) = 0 then
      WriteLn ('Free DPMI memory: ',
               MemInfo.TotalNumberOfFreePages, ' pages.')
    else
      WriteLn ('Error in DpmiGetMemoryInformation');
  end.
```

3.3.9 I got an exception when accessing an 'array [1 .. 4000000] of Byte'.

Per default, the maximum stack size of a DJGPP application is 256K. If you need more, you have to adjust it with the stubedit program, i.e.:

```
stubedit your_app.exe minstack=5000K
```

Another way is to add the following code to your program to define a minimum stack size (here: 2 MB). This value will be honored even if a user sets a lower value by using stubedit, so this method might be a little safer. (The linker name '_stklen' is essential; the Pascal identifier doesn't matter. The constant doesn't have to be used anywhere in the program. It is recommended to put this declaration in the main program file, not in any unit/module, so programs using a unit/module can set whatever limit they need.)

```
{$ifdef __GO32__}
const
  MinStackSize: Cardinal = $200000; attribute (name = '_stklen');
{$endif}
```

Still, it might be a good idea to use pointers for large structures, and allocate the memory at runtime.

DJGPP has to allocate the stack in physical memory at program startup, so one might have to be careful with too large stack limits. Most other systems allocate stack pages on demand, so the only reason to set a limit at all might be to prevent a runaway recursion from eating up all memory . . .

On Unix-like systems, you can set a resource limit, but you usually don't do it in normal programs, but rather in the shell settings (bash: 'ulimit'; csh: 'limit'; syscall: 'setrlimit'(2)).

3.4 Strings

3.4.1 What's this confusion about strings?

Turbo Pascal strings have a length byte in front. Since a byte has the range 0 .. 255, this limits a string to 255 characters. However, the Pascal string schema, as defined in section 6.4.3.3.3 of the ISO 10206 Extended Pascal standard, is a schema record:

```
type
  String (Capacity: Integer) = record
    Length: 0 .. Capacity;
    String: packed array [1 .. Capacity + 1] of Char
  end;
```

The '+ 1' is a GPC extension to make it feasible to automatically add the '#0' terminator when passing or assigning them to CStrings. Thus at the expense of a little added complexity (must declare capacity, don't use 'GetMem' without explicit initialization of the 'Capacity' field, and the additional space requirement) you can now have very long strings.

3.4.2 Overlaying strings in variant records

Q: Should the different variants in a variant record overlay in the same memory? Previous Pascals I have used have guaranteed this, and I've got low-level code that relies on this. The variants are not the same length, and they are intended not to be.

A: No, this is intentional so that the discriminants are not overwritten, and they can be properly initialized in the first place. Consider:

```
record
case Boolean of
  False: (s1: String (42));
  True:  (s2: String (99));
end;
```

If the strings would overlay, in particular their discriminants would occupy the same place in memory. How should it be initialized? Either way, it would be wrong for at least one of the variants ...

So, after a discussion in the ISO Pascal newsgroup (`news:comp.lang.pascal.ansi-iso`) where this topic came up concerning file variables (which also require some automatic initialization and finalization), we decided to do this in GPC for all types with automatic initialization and finalization (currently files, objects and schemata, including strings, in the future this might also be Delphi compatible classes and user-defined initialized and finalized types), since the standard does not guarantee variants to overlay, anyway ...

There are two ways in GPC to get guaranteed overlaying (both non-standard, of course, since the standard does not assume anything about internal representations; both BP compatible), '`absolute`' declarations and variable type casts. E.g., in order to overlay a byte array '`b`' to a variable '`v`':

```
var
  b: array [1 .. SizeOf (v)] of Byte absolute v;
```

Or you can use type-casting:

```
type
  t = array [1 .. SizeOf (v)] of Byte;
```

then '`t (v)`' can be used as a byte array overlayed to '`v`'.

3.4.3 Why does '`s[0]`' not contain the length?

Q: In standard Pascal you expect '`s[1]`' to align with the first character position of '`s`' and thus one character to the left is the length of '`s`'. Thus '`s[0]`' is the length of '`s`'. True?

A: This holds for UCSD/BP strings (which GPC does not yet implement, but that's planned). The only strings Standard Pascal knows are arrays of char without any length field.

GPC also supports Extended Pascal string schemata (see Section 3.4.1 [String schema], page 23), but they also don't have a length byte at "position 0", but rather a '`Length`' field (which is larger than one byte).

3.4.4 Watch out when using strings as parameters

Q: Any "gotchas" with string parameters?

A: Be careful when passing string literals as parameters to routines accepting the string as a value parameter and that internally modify the value of the parameter. Inside the routine, the value parameter gets a fixed capacity – the length of the string literal that was passed to it. Any attempt to assign a longer value will not work.

This only applies if the value parameter is declared as '`String`'. If it is declared as a string with a given capacity (e.g., '`String (255)`'), it gets this capacity within the routine.

3.4.5 Support for BP compatible short strings

Q: Two different kinds of strings with the same name – 'String' – does make a bit of confusion. Perhaps the oldstyle strings could be renamed 'short string' ?

A: When we implement the short strings, we'll have to do such a distinction. Our current planning goes like this:

'String (*n*)': string schema (EP compatible)

'String [*n*]': short string (UCSD/BP compatible, where *n* must be <= 255)

'String': dependent on flags, by default undiscriminated schema, but in BP mode (or with a special switch) short string of capacity 255 (UCSD/BP compatible).

Q: So when will these short strings be available?

A: It's been planned for over a year. The delay has been caused by more pressing problems.

3.4.6 What about C strings?

A C string ('char *') is an array of char, terminated with a '#0' char.

C library functions require C, not Pascal style string arguments. However, Pascal style strings are automatically converted to C style strings when passed to a routine that expects C style strings. This works only if the routine reads from the string, not if it modifies it.

E.g., this is how you could access the 'system()' call in your C library (which is not necessary anymore, since 'Execute' is already built-in):

```
program SysCall;

function System (CmdLine: CString): Integer; external name 'system';

var
  Result: Integer;

begin
  Result := System ('ls -l');
  WriteLn ('system() call returned: ', Result)
end.
```

You could use the type 'PChar' instead of 'CString'. Both 'CString' and 'PChar' are predefined as '^Char' – though we recommend 'CString' because it makes it clearer that we're talking about some kind of string rather than a single character.

A lot of library routines in Pascal for many applications exist in the GPC unit and some other units. Where available, they should be preferred (e.g. 'Execute' rather than 'system()', and then you won't have to worry about 'CString's.)

Do **not** pass a C style string as a 'const' or 'var' argument if the C prototype says 'const char *' or you will probably get a segfault.

3.5 Getting Help

Please read the GPC Manual (info files or other formats) as well as the 'README' and 'BUGS' files that come with GPC (usually installed in directory '/usr/local/doc/gpc'), plus other docs that might help (the DJGPP FAQ if you use DJGPP, etc.) before you send email to the maintainers or mailing list.

In particular, the 'BUGS' file contains information on how to submit bug reports in the most efficient way.

The 'Support' chapter of the GPC Manual tells you where to find more information about GPC and how to contact the GPC developers. (see Chapter 10 [Support], page 459)

3.6 Miscellaneous

3.6.1 I want to contribute; where do I start?

If you want to contribute, please write to the mailing list, Section 10.1 [Mailing List], page 459.

3.6.2 Where is the GNU Pascal web site?

The GPC homepage on the web, for information and downloads, is `http://www.gnu-pascal.de`.

The GPC To-Do list, listing the latest features and fixed bugs can also be found there.

3.6.3 About this FAQ

Current Maintainer: Russ Whitaker, `russ@ashlandhome.net`

This is the second incarnation of the GNU Pascal FAQ list, based on the previous FAQ by J.J. van der Heijden. Comments about, suggestions for, or corrections to this FAQ list are welcome.

Please make sure to include in your mail the version number of the document to which your comments apply (you can find the version at the beginning of this FAQ list).

Many people have contributed to this FAQ, only some of them are acknowledged above. Much of the info in, and inspiration for this FAQ list was taken from the GPC mailing list traffic, so you may have (unbeknownst to you) contributed to this list.

4 How to download, compile and install GNU Pascal.

This chapter covers:

- Downloading GPC sources or binaries
- Installation instructions for a GPC binary distribution
- Compilation of the source distribution on a Unix system
- Compilation notes for specific platforms
- Building and installing a cross-compiler
- Crossbuilding a compiler

4.1 Where and what to download

You can download the source code of the current GNU Pascal release from

```
http://www.gnu-pascal.de/current/
```

and binaries for some platforms from

```
http://www.gnu-pascal.de/binary/
```

The binary archive files are named '`gpc-`*version*`.`*platform*`.`*extension*' – for example '`gpc-2.1.alpha-unknown-linux-gnu.tar.gz`' for GPC version 2.1 on an Alpha workstation running the Linux kernel with GNU C Library, or '`gpc-20000616.i386-pc-msdosdjgpp`' for GPC version 20000616 on an Intel IA32 compatible PC running DOS with DJGPP.

After you have downloaded the correct archive file for your platform, please read the installation notes on how to install such a binary distribution.

If you are running Dos or MS Windows, you will need additional tools – see "What else to download and where" below.

Current snapshots

GNU Pascal is subject to steady development. Alpha and beta snapshots (source only, use at your own risk) can be found at:

```
http://www.gnu-pascal.de/alpha/
http://www.gnu-pascal.de/beta/
```

What else to download and where

When you are using GNU Pascal on a DOS system, you will need either the DJGPP or the EMX development environment (see below). On an OS/2 system, you will need EMX. On an MS Windows 95/98/NT system you will need either the CygWin or the mingw32 ot the MSYS environment.

GNU Pascal uses the compiler back-end from the GNU Compiler Collection, GNU CC or GCC. If you want to compile GPC, you will need the source of GCC as well as the source of GPC itself. From the same place as GPC, please download GCC '`2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x`'. (It is also available from any GNU mirror; see `http://www.gnu.org/software/gcc/`.)

Libraries

For some of GPC's units, you will need some standard libraries. In particular:

Unit	Platform	Library
`CRT`	`Unix/terminal`	`ncurses >= 5.0` (1), (2)
`CRT`	`Unix/X11`	`PDCurses` (2)

CRT	Dos, MS-Windows	PDCurses (3)
GMP	any	gmp
RegEx	any	rx
(debugging)	Unix, MS-Windows	ElectricFence (4)

Notes:

(1) ncurses version 5.0 or newer is strongly recommended because older versions contain a bug that severely affects CRT programs.

(2) You can install both ncurses and PDCurses on a Unix system, and choose at compile time whether to generate a terminal or X11 version of your program.

(3) ncurses also runs under MS-Windows with CygWin (not mingw32, however), but doesn't appear to behave much differently from PDCurses on that platform.

(4) ElectricFence is not used by any unit, but can be used for debugging memory allocation bugs by simply linking it (see the accompanying documentation).

You can find those libraries on many places on the Net. Also, many GNU/Linux distributions, DJGPP mirrors and other OS distributions already contain some of the libraries. In any case, you can find the sources of the libraries (sometimes together with patches that you should apply before building if you choose to build from the sources) and binaries for some platforms in

```
http://www.gnu-pascal.de/libs/
```

For more information and descriptions of these libraries, see Section 3.2.4 [Libraries], page 18.

DJGPP

DJGPP (`http://www.delorie.com/djgpp/`) is available from any SimTel mirror in the '`gnu/djgpp`' subdirectory; for addresses look into the DJGPP FAQ (`http://www.delorie.com/djgpp/v2faq/`). To use GNU Pascal you need at least

- the C library, '`v2/djdev201.zip`', and
- '`binutils`' (assembler, etc.), '`v2gnu/bnu270b.zip`'.

We also recommend you to get:

- the '`make`' utility, '`v2gnu/mak375b.zip`'
- the GNU debugger, '`v2gnu/gdb416b.zip`'
- the DJGPP FAQ, '`v2faq/faq211b.zip`'
- the GRX graphics library, `http://www.gnu.de/software/GRX/`
- PENG, `http://fjf.gnu.de/peng/`, an integrated development environment, similar to BP's one, written in GNU Pascal, or
- RHIDE, '`v2app/rhide.zip`', another integrated development environment, or
- DevPascal, `http://www.bloodshed.net/devpascal.html`, an integrated development environment for mingw32.

EMX

EMX is an environment for creating 32-bit applications for DOS and OS/2. To develop EMX programs with GNU Pascal you need at least

- the EMX runtime package, '`emxrt.zip`',
- the EMX development system, '`emxdev*.zip`', and
- the GNU development tools, '`gnudev*.zip`'.

If your DOS box has DPMI (it does if you are using MS Windows or OS/2) you will also need RSX, available from the same sites as EMX in the subdirectory '`rsxnt`'.

The GNU development tools contain the GNU C compiler which is in fact not needed to use GNU Pascal. However, the C library *is* needed.

CygWin

CygWin is an environment which implements a POSIX layer under MS Windows, giving it large parts of the functionality of Unix. CygWin contains development tools such as an assembler, a linker, etc. GPC needs for operation. More information about CygWin can be found at

```
http://cygwin.com
```

mingw32

The Minimalists' GNU Win32 environment, mingw32, allows a large number of Unix programs – including GPC and GCC – to run under MS Windows 95/98/NT using native MS libraries. mingw32 ressources can be found at

```
http://www.mingw.org
```

4.2 Installation instructions for a GPC binary distribution

To install a binary distribution, `cd` to the root directory and unpack the archive while preserving the stored directory structure. Under a Unix compatible system with GNU `tar` installed, the following (performed as '`root`') will do the job:

```
# cd /
# tar xzf archive.tar.gz
```

If you are using a '`tar`' utility other than GNU `tar`, it might be necessary to do the above in an explicit pipe:

```
# cd /
# gzip -c -d archive.tar.gz | tar xf -
```

Some binary distributions are now distributed packed with '`bzip2`'. You can recognize them by their file name suffix '`.bz2`' instead of '`.gz`'. For such archives, make sure you have '`bunzip2`' installed and then use the following command:

```
# cd /
# tar xjf archive.tar.bz2
```

Or:

```
# cd /
# bunzip2 -c -d archive.tar.bz2 | tar xf -
```

If you want to install a GPC binary distribution in another directory than it was prepared for (for example, if you do not have root access to the computer and want to install GPC somewhere under your home directory), you can do the following:

- Unpack the archive file in a directory of your choice (see above).
- '`cd`' to the "prefix" directory of the distribution (for instance '`usr/local`').
- Run the script `install-gpc-binary`, available from `http://www.gnu-pascal.de/binary/`.
- Follow the instructions in the script.

To install a ZIP archive under DOS with '`PKunzip`', '`cd`' to the appropriate directory (usually '`\`' for EMX, '`\DJGPP`' for DJGPP), then call '`PKunzip`' with the '`-d`' option:

```
C:\> cd djgpp
C:\DJGPP> pkunzip -d archive.zip
```

where '*archive*`.zip`' is the name of the distribution file.

For DJGPP you must edit your '`djgpp.env`' in the '`DJGPP`' directory to complete the installation: Please copy the entries from '`[gcc]`' to create a new '`[gpc]`' section. The result may look as follows:

```
[gcc]
COMPILER_PATH=%/>;COMPILER_PATH%%DJDIR%/bin
LIBRARY_PATH=%/>;LIBRARY_PATH%%DJDIR%/lib

[gpc]
COMPILER_PATH=%/>;COMPILER_PATH%%DJDIR%/bin
LIBRARY_PATH=%/>;LIBRARY_PATH%%DJDIR%/lib
```

If you are using the DJGPP version of GPC but do not have a 'DJGPP' directory, please download and install DJGPP (see Section 4.1 [Download], page 27).

Binary distributions include 'libgcc.a' and 'specs', files that are normally part of GCC. If you have GCC installed, they will be replaced unless you manually install the archive.

4.3 Compiling GPC

The preferred way to distribute GNU software is distribution of the source code. However, it can be a non-trivial exercise to build GNU Pascal on some non-Unix systems, so we also provide ready-to-run binaries for a number of platforms. (See Section 4.2 [Binary Distributions], page 29 for how to install a binary distribution.)

GPC is based on the GNU Compiler Collection, GNU CC or GCC. You will need the GCC sources to build it. It must be the same version as the one GPC is implemented with – `2.8.1, 2.95.x, 3.2.x, 3.3.x or 3.4.x` as of this writing. Although you need GCC to build the GNU Pascal compiler, you don't need GCC to compile Pascal programs once GNU Pascal is installed. (However, using certain libraries will require compiling C wrappers, so it is a good idea to install the C compiler as well.)

Because GNU Pascal shares its back-end with GCC, it should run on any system supported by GCC. A full list of platforms supported by GCC can be found in section "Chapter 4" in *"Using and Porting GNU CC"*.

The GCC source can be obtained from any mirror of the GNU FTP site, `ftp://ftp.gnu.org/gnu/gcc/`. The "core" distribution is sufficient for GPC.

Here is the generic procedure for installing GNU Pascal on a Unix system. See Section 4.4 [Compilation Notes], page 33 for extra information needed to install GPC on DOS-like platforms.

1. Checking the prerequisites

   Make sure that GNU make is installed and that you use it in the following steps. When unsure, you can try 'make --version' and/or 'gmake --version'. It should tell you that it is GNU make. If you don't have it, you can obtain it from `http://www.gnu.org/software/make/`.

   (In the following, we will simply speak of 'make' when invoking GNU make; you might need to call 'gmake' instead.)

   You also need a 'patch' program. If such a program is not installed on your system, you can get GNU patch from `http://www.gnu.org/directory/patch.html`.

   For extracting the example programs from the documentation to the 'doc/docdemos' directory a non-crippled 'sed' is needed. GNU sed is known to work.

   If you have downloaded a "minimal" source distribution, most derived files have to be rebuilt. This is done automatically when building GPC, but you need additional tools:

   'bash', 'bzip2', **GNU** 'sed', GNU 'awk', GNU 'm4', 'bison' (at least version 2.0), 'flex' (version 2.5.27), 'autoconf' (version 2.12), 'texinfo' (at least version 4.2), 'help2man'.

   Make sure that these are installed. The minimal distributions are compressed with 'bzip2' instead of 'gzip', so substitute it in the instructions below.

   If your bison and flex programs are installed under different names, you may have to set some or all of the following environment variables before running 'configure':

```
FLEX=/path/to/flex
LEX=/path/to/flex
BISON=/path/to/bison
YACC=/path/to/bison
INTLBISON=/path/to/bison
```

If you want to build the GPC WWW pages you will also need a TEX distribution (including 'pdftex' and 'dvips').

If you run into trouble during the installation process, please check whether you are using outdated versions of the required utilities and upgrade if necessary.

The GNU versions of the packages above are available from `http://www.gnu.org/software/`, in a subdirectory whose name is the name of the package.

2. Unpacking the source

   From a directory of your choice (e.g. '/home/fred'), unpack the GCC and GNU Pascal source distributions. This will create separate subdirectories for GCC and GPC. Let us assume 'gcc-2.95.3' and 'gpc-20030209' in this example.

   ```
   % cd /home/fred
   % gzip -c -d gcc-core-2.95.3.tar.gz | tar xf -
   % gzip -c -d gpc-20030209.tar.gz | tar xf -
   ```

   'cd' to the GPC directory and move the contents (a subdirectory 'p') to the subdirectory 'gcc' of the GCC directory:

   ```
   % mv /home/fred/gpc-20030209/p /home/fred/gcc-2.95.3/gcc/
   ```

   Instead of moving the directory, it is now also possible to make a symbolic link (if the OS supports symlinks). This is useful if you want to build GPC with several different GCC versions:

   ```
   % ln -s /home/fred/gpc-20030209/p /home/fred/gcc-2.95.3/gcc/p
   ```

   It is recommended, though not required, to use a separate directory for building the compiler, rather than compiling in the source directory. In this example, let us create '/home/fred/gpc-build' for this purpose:

   ```
   % mkdir /home/fred/gpc-build
   ```

   If you use a separate directory, you do not need to write into the GCC source directory once you have patched the GCC source (see below), and can build GPC for more than one platform from the same source tree.

   In case you are re-using a directory where you have already built GCC and/or GPC for a different target machine, do 'make distclean' to delete all files that might be invalid. One of the files this deletes is 'Makefile'; if 'make distclean' complains that 'Makefile' does not exist, it probably means that the directory is already suitably clean.

3. Configuring and building GCC

   GNU Pascal is automatically configured with GCC. Configuration of GCC is treated in depth in section "Chapter 4" in *"Using and Porting GNU CC"*. The normal procedure is as follows:

   'cd' to the GPC build directory. From there, run the 'configure' script in the GCC source directory:

   ```
   % cd /home/fred/gpc-build
   % /home/fred/gcc-2.95.3/configure --enable-languages=pascal
   ```

   This creates all the necessary config files, links and Makefile in the GCC object directory.

   Note 1: The configuration will prompt you for patching the GCC source for GPC support, so you need write access to that directory. All changes to GCC are surrounded by '#ifdef

`GPC ... #endif`', so they should not interfere when you build a C compiler from this source tree.

Note 2: The '`--enable-languages=pascal`' option means that we only want to build the Pascal compiler and not, for instance, the C++ compiler.

Note 3: The standard base directory for installing GCC and GPC is '`/usr/local`'. If you want to install files to an alternate directory *dir*, specify '`--prefix=`*dir*' when you run '`configure`'. For installing into a system directory such as '`/usr/local`' you will, of course, need appropriate permissions (often root). Therefore, if you want to install GPC on a system where you don't have those permissions, you must specify a prefix (e.g., '`$HOME/usr`').

4. Putting other GNU tools in place

   Some environments require other GNU tools (such as the GNU assembler or linker) instead of the standard system tools for GCC to work. (See the GCC installation instructions for details.) If this is the case for your system, install the required tools in the GPC build directory under the names '`as`', '`ld`', or whatever is appropriate. This will enable the compiler to find the proper tools for compilation of the program '`enquire`' (a part of GCC) and to install the GNU tools to a place where they are found by GCC but do not interfere with the standard system tools.

   Alternatively, you can do subsequent compilation using a value of the `PATH` environment variable such that the necessary GNU tools come before the standard system tools.

5. Compiling GPC

   Once you are satisfied with the configuration as determined by '`configure`', you can build the compiler:

   ```
   % make
   ```

   Notice that this procedure will build the C compiler (and maybe some other compilers) too, because that is used to compile the GPC runtime library.

   Optionally, you may supply CFLAGS, LDFLAGS or RTSFLAGS. CFLAGS is used for compiler and RTS, RTSFLAGS are for RTS only, i.e.: '`make CFLAGS="-O2" RTSFLAGS=-Wall`'

   Note: The documentation may fail to build from *.texi sources if GCC 2.95.x tries to use an older version of '`makeinfo`' supplied in GCC package itself. This can be prevented by supplying explicit instruction to use your system's '`makeinfo`':

   ```
   % make MAKEINFO=`which makeinfo`
   ```

   optionally followed by the rest of arguments.

6. Completing the installation

   When everything has been compiled, you can check the installation process with:

   ```
   % make -n install
   ```

   To complete the installation, run the command '`make install`'. You need write access to the target directories ('`/usr/local/bin`', '`/usr/local/lib`', '`/usr/local/info`', '`/usr/local/doc`', and '`/usr/local/man`' in this example), so this is usually done as '`root`':

   ```
   % su -c "make install"
   ```

   If you want to install *only* the Pascal compiler (for example if you already have the correct version of GCC installed), '`cd`' to the '`gcc`' subdirectory of the build directory (e.g. '`/home/fred/gpc-build/gcc`') and run '`make pascal.install`'. This installation process does **not** overwrite existing copies of '`libgcc.a`' or '`specs`', should they exist.

   However, if you do not have the exactly matching GCC version installed, you might need some additional files (otherwise GPC will complain about missing files at runtime). You can install them by doing '`make pascal.install-with-gcc`' in the '`gcc`' subdirectory of the build directory.

   There is a (partial) translation of the GPC manual into Croatian available now. It is not installed by default. If you want to install it, do a '`pascal.install-hr`' in the '`gcc`' directory. This will install the manpage '`gpc-hr.1`' and the info documentation '`gpc-hr.info*`'.

Other formats like PS, PDF and HTML can be built manually (it's also easy to add appropriate make targets for them when needed).

Also from the '`gcc`' subdirectory you can do some more "exotic" builds. For instance, you can build the GPC WWW pages by typing '`make pascal.html`' or a binary distribution by typing '`make pascal.bindist`'. See the '`Makefile`' in that directory for more examples.

4.4 Compilation notes for specific platforms

4.4.1 MS-DOS with DJGPP

The only compiler that is capable of compiling the GNU Compiler Collection (GNU CC or GCC) under MS-DOS is GCC itself. In order to compile GPC or GCC for MS-DOS with DJGPP you will therefore need either a working copy of DJGPP installed, or you will have to cross-build from a non-MS-DOS system.

Building GPC under MS-DOS with DJGPP follows the same scheme as building GPC under a Unix-like system: Place the '`p`' subdirectory in the '`gcc`' directory and follow the instructions for compiling GCC. This requires '`bash`' and many other tools installed, and you must be very careful at many places to circumvent the limitations of the DOS platform.

Our preferred way to build GPC for DJGPP is to cross-build it from a Unix-like platform – which is much easier. For instructions, see Section 4.5 [Cross-Compilers], page 33 and Section 4.6 [Crossbuilding], page 34.

4.4.2 MS-DOS or OS/2 with EMX

EMX is a free 32-bit DOS extender which adds some properties of Unix to MS-compatible DOS and IBM's OS/2 operating systems.

As of this writing, we are not aware of current versions of GCC for EMX, and EMX support in GPC has not been maintained. Please contact us if you know about recent development in EMX and are interested in continuing EMX support in GPC.

4.4.3 MS Windows 95/98/NT

There are two ports of the GNU development tools to MS Windows 95/98/NT: CygWin and mingw32.

The CygWin environment implements a POSIX layer under MS Windows, giving it large parts of the functionality of Unix. Thus, compiling GCC and GPC under the CygWin environment can be done following the instructions for compiling it under a Unix-like system (see Section 4.3 [Compiling GPC], page 30).

The Minimalists' GNU Win32 environment, mingw32, uses the native '`crtdll.dll`' library of MS Windows. It is much smaller than CygWin, but it is not self-hosting and must be crossbuilt from another system (see Section 4.6 [Crossbuilding], page 34).

4.5 Building and Installing a cross-compiler

GNU Pascal can function as a cross-compiler for many machines. Information about GNU tools in a cross-configuration can be found at '`ftp://ftp.cygnus.com/pub/embedded/crossgcc/`'.

Since GNU Pascal generates assembler code, you need a cross-assembler that GNU Pascal can run, in order to produce object files. If you want to link on other than the target machine, you need a cross-linker as well. It is straightforward to install the GNU binutils to act as cross-tools – see the installation instructions of the GNU binutils for details.

You also need header files and libraries suitable for the target machine that you can install on the host machine. Please install them under '*prefix*/*platform*/include/', for instance '/usr/local/i386-pc-msdosdjgpp/include/' for a cross-compiler from a typical Unix-like environment to MS-DOS with DJGPP.

Configuration and compilation of the compiler can then be done using the scripts 'cfgpc' and 'mkgpc' which are included in the source distribution in the subdirectory 'p/script'. Please call them with the '-h' option for instructions.

4.6 Crossbuilding a compiler

Using a cross-compiler to build GNU Pascal results in a compiler binary that runs on the cross-target platform. This is called "crossbuilding". A possible reason why anybody would want to do this, is when the platform on which you want to run the GNU Pascal compiler is not self-hosting. An example is mingw32.

To crossbuild GNU Pascal, you have to install a cross-compiler for your target first, see Section 4.5 [Cross-Compilers], page 33.

As when building a cross-compiler, configuration and compilation of the compiler can be done using the scripts 'cfgpc' and 'mkgpc' which are included in the source distribution in the subdirectory 'p/script'. Please call them with the '-h' option for instructions.

5 Command Line Options supported by GNU Pascal.

GPC is a command-line compiler, i.e., to compile a program you have to invoke '`gpc`' passing it the name of the file you want to compile, plus options.

GPC supports all command-line options that GCC knows, except for many preprocessor options. For a complete reference and descriptions of all options, see section "GCC Command Options" in *the GCC Manual*. Below, you will find a list of the additional options that GPC supports, and a list of GPC's most important options (including some of those supported by GCC as well).

You can mix options and file names on the command line. For the most part, the order doesn't matter. Order does matter, e.g., when you use several options of the same kind; for example, if you specify '`-L`' more than once, the directories are searched in the order specified. *Note:* Since many options have multiletter names; multiple single-letter options may *not* be grouped as is possible with many other programs: '`-dr`' is very different from '`-d -r`'.

Many options have long names starting with '`--`' or, completely equivalent '`-f`'. E.g., '`--mixed-comments`' is the same as '`-fmixed-comments`'. Some options tell GPC when to give warnings, i.e. diagnostic messages that report constructs which are not inherently erroneous but which are risky or suggest there may have been an error. Those options start with '`-W`'.

Most GPC specific options can also be changed during one compilation by using compiler directives in the source, e.g. '`{$X+}`' or '`{$extended-syntax}`' for '`--extended-syntax`' (see Section 6.9 [Compiler Directives], page 89).

GPC understands the same environment variables GCC does (see section "Environment Variables Affecting GCC" in *the GCC manual*). In addition, GPC recognizes '`GPC_EXEC_PREFIX`' with the same meaning that '`GCC_EXEC_PREFIX`' has to GCC. GPC also recognizes '`GCC_EXEC_PREFIX`', but '`GPC_EXEC_PREFIX`' takes precedence.

Some of the long options (e.g., '`--unit-path`') take an argument. This argument is separated with a '`=`' sign, e.g.:

```
--unit-path=/home/foo/units
```

5.1 GPC options besides those of GCC.

The following table lists the command line options GPC understands in addition to those understood by GCC.

`--debug-tree`
: (For GPC developers.) Show the internal representation of a given tree node (name or address).

`--debug-gpi`
: (For GPC developers.) Show what is written to and read from GPI files (huge output!).

`--debug-automake`
: (For GPC developers.) Give additional information about the actions of automake.

`--debug-source`
: Output the source while it is being processed to standard error.

`--no-debug-source`
: Do not output the source while it is being processed (default).

`--disable-debug-info`
: Inhibit '`-g`' options (temporary work-around, this option may disappear in the future).

`--progress-messages`
: Output source file names and line numbers while compiling.

`--no-progress-messages`
: Do not output source file names and line numbers while compiling (default).

`--progress-bar`
: Output number of processed lines while compiling.

`--no-progress-bar`
: Do not output number of processed lines while compiling (default).

`--automake-gpc`
: Set the Pascal compiler invoked by automake.

`--automake-gcc`
: Set the C compiler invoked by automake.

`--automake-g++`
: Set the C++ compiler invoked by automake.

`--amtmpfile`
: (Internal switch used for automake).

`--autolink`
: Automatically link object files provided by units/modules or '`{$L ...}`' (default).

`--no-autolink`
: Do not automatically link object files provided by units/modules/'`{$L ...}`'.

`--automake`
: Automatically compile changed units/modules/'`{$L ...}`' files and link the object files provided.

`--no-automake`
: Same as '`--no-autolink`'.

`--autobuild`
: Automatically compile all units/modules/'`{$L ...}`' files and link the object files provided.

`--no-autobuild`
: Same as '`--no-autolink`'.

`--maximum-field-alignment`
: Set the maximum field alignment in bits if '`pack-struct`' is in effect.

`--ignore-packed`
: Ignore '`packed`' in the source code (default in '`--borland-pascal`').

`--no-ignore-packed`
: Do not ignore '`packed`' in the source code (default).

`--ignore-garbage-after-dot`
: Ignore anything after the terminating '`.`' (default in '`--borland-pascal`').

`--no-ignore-garbage-after-dot`
: Complain about anything after the terminating '`.`' (default).

`--extended-syntax`
: same as '`--ignore-function-results --pointer-arithmetic --cstrings-as-strings -Wno-absolute`' (same as '`{$X+}`').

`--no-extended-syntax`
: Opposite of '`--extended-syntax`' (same as '`{$X-}`').

`--ignore-function-results`
: Do not complain when a function is called like a procedure.

`--no-ignore-function-results`
: Complain when a function is called like a procedure (default).

`--pointer-arithmetic`
: Enable pointer arithmetic.

`--no-pointer-arithmetic`
: Disable pointer arithmetic (default).

`--cstrings-as-strings`
: Treat CStrings as strings.

`--no-cstrings-as-strings`
: Do not treat CStrings as strings (default).

`-Wabsolute`
: Warn about variables at absolute adresses and '`absolute`' variable with non-constant addresses (default).

`-Wno-absolute`
: Do not warn about variables at absolute adresses and '`absolute`' variable with non-constant addresses.

`--short-circuit`
: Guarantee short-circuit Boolean evaluation (default; same as '`{$B-}`').

`--no-short-circuit`
: Do not guarantee short-circuit Boolean evaluation (same as '`{$B+}`').

`--mixed-comments`
: Allow comments like '`{ ... *)`' as required in ISO Pascal (default in ISO 7185/10206 Pascal mode).

`--no-mixed-comments`
: Ignore '`{`' and '`}`' within '`(* ... *)`' comments and vice versa (default).

`--nested-comments`
: Allow nested comments like '`{ { } }`' and '`(* (* *) *)`'.

`--no-nested-comments`
: Do not allow nested comments (default).

`--delphi-comments`
: Allow Delphi style '`//`' comments (default).

`--no-delphi-comments`
: Do not allow Delphi style '`//`' comments.

`--macros`
: Expand macros (default).

`--no-macros`
: Do not expand macros (default with '`--ucsd-pascal`', '`--borland-pascal`' or '`--delphi`').

`--truncate-strings`
: Truncate strings being assigned to other strings of too short capacity.

`--no-truncate-strings`
: Treat string assignments to other strings of too short capacity as errors.

`--exact-compare-strings`
: Do not blank-pad strings for comparisons.

`--no-exact-compare-strings`
: Blank-pad strings for comparisons.

`--double-quoted-strings`
: Allow strings enclosed in "\" (default).

`--no-double-quoted-strings`
: Do not allow strings enclosed in "\" (default with dialect other than '--mac-pascal').

`--longjmp-all-nonlocal-labels`
: Use 'longjmp' for all nonlocal labels.

`--no-longjmp-all-nonlocal-labels`
: Use 'longjmp' only for nonlocal labels in the main program (default).

`--nonlocal-exit`
: Allow non-local 'Exit' statements (default in '--ucsd-pascal' and '--mac-pascal').

`--no-nonlocal-exit`
: Do not allow non-local 'Exit' statements (default).

`--io-checking`
: Check I/O operations automatically (same as '{$I+}') (default).

`--no-io-checking`
: Do not check I/O operations automatically (same as '{$I-}').

`--pointer-checking-user-defined`
: Use user-defined procedure for validating pointers.

`--no-pointer-checking-user-defined`
: Do not use user-defined procedure for validating pointers (default).

`--pointer-checking`
: Validate pointers before dereferencing.

`--no-pointer-checking`
: Do not validate pointers before dereferencing (default).

`--object-checking`
: Check for valid objects on virtual method calls (default).

`--no-object-checking`
: Do not check for valid objects on virtual method calls.

`--range-checking`
: Do automatic range checks') (default).

`--no-range-checking`
: Do not do automatic range checks (same as '{$R-}').

`--range-and-object-checking`
: Same as '--range-checking --object-checking', same as '{$R+}'.

`--no-range-and-object-checking`
: Same as '--no-range-checking --no-object-checking', same as '{$R-}'.

`--case-value-checking`
: Cause a runtime error if a 'case' matches no branch (default in ISO Pascal modes).

`--no-case-value-checking`
: Do not cause a runtime error if a 'case' matches no branch (default).

`--stack-checking`
: Enable stack checking (same as '`{$S+}`').

`--no-stack-checking`
: Disable stack checking (same as '`{$S-} (default)`').

`--read-base-specifier`
: In read statements, allow input base specifier '`n#`' (default).

`--no-read-base-specifier`
: In read statements, do not allow input base specifier '`n#`' (default in ISO 7185 Pascal).

`--read-hex`
: In read statements, allow hexadecimal input with '`$`' (default).

`--no-read-hex`
: In read statements, do not allow hexadecimal input with '`$`' (default in ISO 7185 Pascal).

`--read-white-space`
: In read statements, require whitespace after numbers.

`--no-read-white-space`
: In read statements, do not require whitespace after numbers (default).

`--write-clip-strings`
: In write statements, truncate strings exceeding their field width ('`Write (SomeLongString : 3)`').

`--no-write-clip-strings`
: Do not truncate strings exceeding their field width.

`--write-real-blank`
: Output a blank in front of positive reals in exponential form (default).

`--no-write-real-blank`
: Do not output a blank in front of positive reals in exponential form.

`--write-capital-exponent`
: Write real exponents with a capital '`E`'.

`--no-write-capital-exponent`
: Write real exponents with a lowercase '`e`'.

`--transparent-file-names`
: Derive external file names from variable names.

`--no-transparent-file-names`
: Do not derive external file names from variable names (default).

`--field-widths`
: Optional colon-separated list of default field widths for Integer, Real, Boolean, LongInt, LongReal.

`--no-field-widths`
: Reset the default field widths.

`--pedantic`
: Reject everything not allowed in some dialect, e.g. redefinition of its keywords.

`--no-pedantic`
: Don't give pedantic warnings.

`--typed-address`
: Make the result of the address operator typed (same as '`{$T+}`', default).

`--no-typed-address`
: Make the result of the address operator an untyped pointer (same as '`{$T-}`').

`--enable-keyword`
: Enable a keyword, independently of dialect defaults.

`--disable-keyword`
: Disable a keyword, independently of dialect defaults.

`--implicit-result`
: Enable implicit '`Result`' for functions (default only in '`--delphi`').

`--no-implicit-result`
: Disable implicit '`Result`' for functions.

`--enable-predefined-identifier`
: Enable a predefined identifier, independently of dialect defaults.

`--disable-predefined-identifier`
: Disable a predefined identifier, independently of dialect defaults.

`--assertions`
: Enable assertion checking (default).

`--no-assertions`
: Disable assertion checking.

`--setlimit`
: Define the range for '`set of Integer`' etc..

`--gpc-main`
: External name for the program's entry point (default: '`main`').

`--propagate-units`
: Automatically re-export all imported declarations.

`--no-propagate-units`
: Do not automatically re-export all imported declarations.

`--interface-only`
: Compile only the interface part of a unit/module and exit (creates '`.gpi`' file, no '`.o`' file.

`--implementation-only`
: Do not produce a GPI file; only compile the implementation part.

`--executable-file-name`
: Name for the output file, if specified; otherwise derive from main source file name.

`--unit-path`
: Directories where to look for unit/module sources.

`--no-unit-path`
: Forget about directories where to look for unit/module sources.

`--object-path`
: Directories where to look for additional object (and source) files.

`--no-object-path`
: Forget about directories where to look for additional object (and source) files.

`--executable-path`
: Path where to create the executable file.

`--no-executable-path`
: Create the executable file in the directory where the main source is (default).

`--unit-destination-path`
: Path where to create object and GPI files of Pascal units.

`--no-unit-destination-path`
: Create object and GPI files of Pascal units in the current directory (default).

`--object-destination-path`
: Path where to create additional object files (e.g. of C files, not Pascal units).

`--no-object-destination-path`
: Create additional object files (e.g. of C files, not Pascal units) in the current directory (default).

`--disable-default-paths`
: Do not add a default path to the unit and object path.

`--gpi-destination-path`
: (Internal switch used for automake).

`--uses`
: Add an implicit '`uses`' clause.

`--init-modules`
: Initialize the named modules in addition to those imported regularly; kind of a kludge.

`--cidefine`
: Define a case-insensitive macro.

`--csdefine`
: Define a case-sensitive macro.

`--big-endian`
: Tell GPC that the system is big-endian (for those targets where it can vary).

`--little-endian`
: Tell GPC that the system is little-endian (for those targets where it can vary).

`--print-needed-options`
: Print the needed options.

`-Wwarnings`
: Enable warnings (same as '`{$W+}`').

`-Wno-warnings`
: Disable all warnings (same as '`{$W-}`').

`-Widentifier-case-local`
: Warn about an identifier written with varying case within one program/module/unit.

`-Wno-identifier-case-local`
: Same as '`-Wno-identifier-case`'.

`-Widentifier-case`
: Warn about an identifier written with varying case.

`-Wno-identifier-case`
: Do not warn about an identifier written with varying case (default).

`-Winterface-file-name`
Warn when a unit/module interface differs from the file name.

`-Wno-interface-file-name`
Do not warn when a unit/module interface differs from the file name (default).

`--methods-always-virtual`
Make all methods virtual (default in ‘--mac-pascal’).

`--no-methods-always-virtual`
Do not make all methods virtual (default).

`-Wimplicit-abstract`
Warn when an object type not declared ‘abstract’ contains an abstract method (default).

`-Wno-implicit-abstract`
Do not warn when an object type not ‘declared’ abstract contains an abstract method.

`-Winherited-abstract`
Warn when an abstract object type inherits from a non-abstract one (default).

`-Wno-inherited-abstract`
Do not warn when an abstract object type inherits from a non-abstract one.

`-Wobject-assignment`
Warn when when assigning objects or declaring them as value parameters or function results (default).

`-Wno-object-assignment`
Do not warn when assigning objects or declaring them as value parameters or function results (default in ‘--borland-pascal’).

`-Wimplicit-io`
Warn when ‘Input’ or ‘Output’ are used implicitly.

`-Wno-implicit-io`
Do not warn when ‘Input’ or ‘Output’ are used implicitly (default).

`-Wfloat-equal`
Warn about ‘=’ and ‘<>’ comparisons of real numbers.

`-Wno-float-equal`
Do not warn about ‘=’ and ‘<>’ comparisons of real numbers.

`-Wtyped-const`
Warn about misuse of typed constants as initialized variables (default).

`-Wno-typed-const`
Do not warn about misuse of typed constants as initialized variables.

`-Wnear-far`
Warn about use of useless ‘near’ or ‘far’ directives (default).

`-Wno-near-far`
Do not warn about use of useless ‘near’ or ‘far’ directives.

`-Wunderscore`
Warn about double/leading/trailing underscores in identifiers.

`-Wno-underscore`
Do not warn about double/leading/trailing underscores in identifiers.

`-Wsemicolon`
: Warn about a semicolon after '`then`', '`else`' or '`do`' (default).

`-Wno-semicolon`
: Do not warn about a semicolon after '`then`', '`else`' or '`do`'.

`-Wlocal-external`
: Warn about local '`external`' declarations.

`-Wno-local-external`
: Do not warn about local '`external`' declarations.

`-Wdynamic-arrays`
: Warn about arrays whose size is determined at run time (including array slices).

`-Wno-dynamic-arrays`
: Do not warn about arrays whose size is determined at run time (including array slices).

`-Wmixed-comments`
: Warn about mixed comments like '`{ ... *)`'.

`-Wno-mixed-comments`
: Do not warn about mixed comments.

`-Wnested-comments`
: Warn about nested comments like '`{ { } }`'.

`-Wno-nested-comments`
: Do not warn about nested comments.

`--classic-pascal-level-0`
: Reject conformant arrays and anything besides ISO 7185 Pascal.

`--standard-pascal-level-0`
: Synonym for '`--classic-pascal-level-0`'.

`--classic-pascal`
: Reject anything besides ISO 7185 Pascal.

`--standard-pascal`
: Synonym for '`--classic-pascal`'.

`--extended-pascal`
: Reject anything besides ISO 10206 Extended Pascal.

`--object-pascal`
: Reject anything besides (the implemented parts of) ANSI draft Object Pascal.

`--ucsd-pascal`
: Try to emulate UCSD Pascal.

`--borland-pascal`
: Try to emulate Borland Pascal, version 7.0.

`--delphi`
: Try to emulate Borland Pascal, version 7.0, with some Delphi extensions.

`--pascal-sc`
: Be strict about the implemented Pascal-SC extensions.

`--vax-pascal`
: Support (a few features of) VAX Pascal.

`--sun-pascal`
: Support (a few features of) Sun Pascal.

`--mac-pascal`
: Support (some features of) traditional Macintosh Pascal compilers.

`--gnu-pascal`
: Undo the effect of previous dialect options, allow all features again.

5.2 The most commonly used options to GPC

As the most simple example, calling

```
gpc foo.pas
```

tells GPC to compile the source file '`foo.pas`' and to produce an executable of the default name which is '`foo.exe`' on EMX, '`a.exe`' on Cygwin, both '`a.out`' and '`a.exe`' on DJGPP, and '`a.out`' on most other platforms.

Users familiar with BP, please note that you have to give the file name extension '`.pas`': GPC is a common interface for a Pascal compiler, a C, ObjC and C++ compiler, an assembler, a linker, and perhaps an Ada and a FORTRAN compiler. From the extension of your source file GPC figures out which compiler to run. GPC recognizes Pascal sources by the extension '`.pas`', '`.p`', '`.pp`' or '`.dpr`'. GPC also accepts source files in other languages (e.g., '`.c`' for C) and calls the appropriate compilers for them. Files with the extension '`.o`' or without any special recognized extension are considered to be object files or libraries to be linked.

Another example:

```
gpc -O2 -Wall --executable-file-name --automake --unit-path=units foo.pas
```

This will compile the source file '`foo.pas`' to an executable named '`foo`' ('`--executable-file-name`') with fairly good optimization ('`-O2`'), warning about possible problems ('`-Wall`'). If the program uses units or imports modules, they will be searched for in a directory called '`units`' ('`--unit-path`') and automatically compiled and linked ('`--automake`').

The following table lists the most commonly used options to GPC.

`--automake`
: Check whether modules/units used must be recompiled and do the recompilation when necessary.

`--unit-path=`*dir*`[:`*dir*`...]`
: Search the given directories for units and object files.

`--object-path=`*dir*`[:`*dir*`...]`
: Search the given directories for object files.

`--unit-destination-path=`*dir*
: Place compiled units (GPI and object files) into the directory *dir*. The default is the current directory.

`--object-destination-path=`*dir*
: Place compiled object files (e.g., from C files, but not from Pascal units) into the directory *dir*. The default is the directory given with '`--unit-destination-path`'.

`--executable-path=`*dir*
: Place the executable compiled into the directory *dir*. The default is the main source file's directory.

`-o` *file*
: Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file, etc.

: Since only one output file can be specified, it does not make sense to use '`-o`' when compiling more than one input file, unless you are producing an executable file as output.

`--executable-file-name[=`*name*`]`
: Derive the executable file name from the source file name, or use *name* as the executable file name. The difference to the '`-o`' option is that '`--executable-file-name`' considers the '`--executable-path`', while '`-o`' does not and accepts a file name with directory. Furthermore, '`--executable-file-name`' only applies to executables, not to other output formats selected.

`-L`*dir*
: Search the directory *dir* for libraries. Can be given multiple times.

`-I`*dir*
: Search the directory *dir* for include files. Can be given multiple times.

`-l`*library*
: Search the library named *library* when linking. This option must be placed on the command line *after* all source or object files or other libraries that reference the library.

`-O[`*n*`]`
: Select the optimization level. Without optimization (or '`-O0`' which is the default), the compiler's goal is to reduce the compilation time and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the same routine and get exactly the results you would expect from the source code.

  With optimization, the compiler tries to reduce code size and execution time. The higher the value of *n*, the more optimizations will be done, but the longer the compilation will take.

  If you use multiple '`-O`' options, with or without *n*, the last such option is the one that is effective.

`-g`
: Produce debugging information suitable for '`gdb`'. Unlike some other compilers, GNU Pascal allows you to use '`-g`' with '`-O`'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs still in the testing phase.

`-s`
: Remove all symbol table and relocation information from the executable. Note: this has no influence on the performance of the compiled executable.

`-Wall`
: Give warnings for a number of constructs which are not inherently erroneous but which are risky or suggest there may have been an error. There are additional warning options not implied by '`-Wall`', see the GCC warning options (see section "Options to Request or Suppress Warnings" in *the GCC manual*), while '`-Wall`' only warns about such constructs that should be easy to avoid in programs. Therefore, we suggest using '`-Wall`' on most sources.

  Note that some warnings (e.g., those about using uninitialized variables) are never given unless you compile with optimization (see above), because otherwise the compiler doesn't analyze the usage patterns of variables.

`-Werror`
: Turn all warnings into errors.

`-S`
: Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each source file. By default, the assembler file name for a source file is made by replacing the extension with '`.s`'.

`-c`
: Compile and assemble the source files, but do not link. The output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the extension with '`.o`'.

`-static` On systems that support dynamic linking, this prevents linking with the shared libraries, i.e. forces static linking. On other systems, this option has no effect.

`-D`*macro*`[=`*def*`]`
: Define the macro and conditional symbol *macro* as *def* (or as '`1`' if *def* is omitted).

`-b` *machine*
: The argument *machine* specifies the target machine for compilation. This is useful when you have installed GNU Pascal as a cross-compiler.

`-v` Print (on standard error) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

`--classic-pascal-level-0`
`--classic-pascal`
`--extended-pascal`
`--object-pascal`
`--ucsd-pascal`
`--borland-pascal`
`--pascal-sc`
: GNU Pascal supports the features of several different Pascal standards and dialects. By default, they are all enabled. These switches tell GPC to restrict itself to the features of the specified standard. It does not enable any additional features. Warnings about certain dangerous constructs which would be valid in the specified dialect (e.g., assignment to a typed constant with '`--borland-pascal`') are suppressed.

  By default, GNU Pascal allows the redefinition of some keywords. Each of these switches causes GNU Pascal to forbid the redefinition of keywords of the specified standard.

  Valid ISO 7185 Pascal programs should compile properly with or without '`--classic-pascal`'. However, without this option, certain GNU extensions and Pascal features from other dialects are supported as well. With this option, they are rejected.

  These options are not intended to be *useful*; they exist only to satisfy pedants who would otherwise claim that GNU Pascal fails to support the ISO Standard or is not *really* compatible to Borland Pascal, or whatever. We recommend, rather, that users take advantage of the extensions of GNU Pascal and disregard the limitations of other compilers.

`-pedantic-errors`
: Produce errors rather than warnings for portability violations. Unlike in C, this does *not* imply the '`-pedantic`' option, so you can, for instance, use '`-pedantic-errors`' without '`-pedantic`', but with '`--extended-pascal`'.

`--gpc-main=`*name*
: Name the entry point of the main program '*name*' instead of '`main`' on the linker level. This is useful, e.g., when working with some C libraries which define their own '`main`' function and require the program's main entry point to be named differently. (This option should preferably be used as a compiler directive in the unit or module which links to that strange C library, rather than be given on the command-line.)

6 The Programmer's Guide to GPC

This chapter is still under development.

This chapter tells you how the source of a valid GNU Pascal program should look like. You can use it as tutorial about the GNU Pascal language, but since the main goal is to document all special GPC features, implementation-dependent stuff, etc., expect a steep learning curve.

This chapter does *not* cover how to compile your programs and to produce an executable – this is discussed above in Chapter 5 [Invoking GPC], page 35.

6.1 Source Structures

A source file accepted by GNU Pascal may contain up to one program, zero or more ISO-style modules, and/or zero or more UCSD-style units. Units and modules can be mixed in one project.

One trivial example for a valid GPC source file follows. Note that the code below may either be in one source file, or else the unit and the program may be in separate source files.

```
unit DemoUnit;

interface

procedure Hello;

implementation

procedure Hello;
begin
  WriteLn ('Hello, world!')
end;

end.
```

```
program UnitDemo;

uses
  DemoUnit;

begin
  Hello
end.
```

6.1.1 The Source Structure of Programs

A generic GNU Pascal program looks like the following:

```
program name (Input, Output);

import_part

declaration_part

begin
  statement_part
```

```
end.
```

The `program` headline may be omitted in GPC, but a warning will be given except in '`--borland-pascal`' mode.

While the program parameters (usually '`Input`', '`Output`') are obligatory in ISO Pascal if you want to use '`ReadLn`' and '`WriteLn`', they are optional in GNU Pascal. GPC will warn about such missing parameters in '`--extended-pascal`' mode. However if you give parameters to the program headline, they work like ISO requires.

The *import_part* consists either of an ISO-style '`import`' specification or a UCSD/Borland-style '`uses`' clause. While '`import`' is intended to be used with interfaces exported by ISO 10206 Extended Pascal modules, and '`uses`' is intended to be used with units, this is not enforced. (See also [uses], page 441, [import], page 336.)

The *declaration_part* consists of label, constant, type, variable or subroutine declarations in free order. However, every identifier must be declared before it is used. The only exception are type identifiers pointing to another type identifier which may be declared below.

The *statement_part* consists of a sequence of statements.

As an extension, GPC supports a "declaring statement" which can be used in the statement part to declare variables (see [var], page 444).

6.1.2 Label Declaration

A label declaration has the following look:

```
label
  label_name, ..., label;
```

A label declaration part starts with the reserved word `label`, which contains a list of labels.

See also

[label], page 345, [goto], page 331

6.1.3 Constant Declaration

A constant declaration has the following look:

```
const
  constant_identifier = constant_expression;
  ...
  constant_identifier = constant_expression;
```

A constant declaration part starts with the reserved word `const`. It declares a *constant_identifier* which is defined by *constant_expression*. This expression has to be evaluatable during compilation time, i.e. it can include numbers, parentheses, predefined operators, sets and type casts (the last, however, is a Borland extension). In ISO 7185 Pascal, *constant_expression* must be a constant or a set. All Pascal Dialects but ISO-Pascal allow the use of these intrinsic functions in *constant_expression*:

[Abs], page 257, [Round], page 407, [Trunc], page 435, [Chr], page 290, [Ord], page 377, [Length], page 347, [Pred], page 388, [Succ], page 428, [SizeOf], page 421, [Odd], page 374.

In Borland Pascal, in the constant declaration part variables can be declared as well, which are given an initial value. These variables are called "typed constants". It is good style to avoid this use, especially since Extended Pascal and GNU Pascal allow to initialize a variable in variable declaration part or give a type a preset value on declaration.

```
const
  FiveFoo      = 5;
  StringFoo    = 'string constant';
  AlphabetSize = Ord ('Z') - Ord ('A') + 1;

type
  PInteger     = ^Integer;     { Define a pointer to an Integer }

const
  { Constant which holds a pointer to an Integer at address 1234 }
  AddressFoo   = PInteger (1234);
```

- BP does not know initialized variables, only typed constants. Even worse, it allows them to be misused as variables, without even warning. GPC supports this (unwillingly ;–), and warns unless in '`--borland-pascal`' mode.

  An example of a typed constant:

  ```
  const
    i: Integer = 0;
  ```

  If you want to use it as a constant only, that's perfectly fine. However, if you modify '`i`', we suggest to translate the declaration to an initialized variable. The EP syntax is:

  ```
  var
    i: Integer value 0;
  ```

  GPC supports this as well as the following mixture of dialects:

  ```
  var
    i: Integer = 0;
  ```

  Furthermore, you can also assign initialization values to types:

  ```
  program InitTypeDemo;

  type
    MyInteger = Integer value 42;

  var
    i: MyInteger;

  begin
    WriteLn (i)
  end.
  ```

  Here, all variables of type MyInteger are automatically initialized to 42 when created.

- Arrays initializers look like this in BP:

  ```
  program BPArrayInitDemo;

  const
    MyStringsCount = 5;

  type
    Ident = String [20];

  const
    MyStrings: array [1 .. MyStringsCount] of Ident =
      ('export', 'implementation', 'import',
  ```

```
          'interface', 'module');

    begin
    end.
```

And the following way in EP:

```
program EPArrayInitDemo;

const
  MyStringsCount = 5;

type
  Ident = String (20);

var
  MyStrings: array [1 .. MyStringsCount] of Ident value
    [1: 'export'; 2: 'implementation'; 3: 'import';
     4: 'interface'; 5: 'module'];

begin
end.
```

There seem to be pros and cons to each style. GPC supports both as well as just about any thinkable mixture of them.

Some folks don't like having to specify an index since it requires renumbering if you want to add a new item to the middle. However, if you index by an enumerated type, you might be able to avoid major renumbering by hand.

See also

Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53

6.1.4 Type Declaration

A type declaration looks like this:

```
type
  type_identifier = type_definition;
  ...
  type_identifier = type_definition;
```

or, with preset content:

```
type
  type_identifier = type_definition value constant_expression;
  ...
  type_identifier = type_definition value constant_expression;
```

A type declaration part begins with the reserved word `type`. It declares a *type_identifier* which is defined by *type_definition*. A type definition either can be an array, a record, a schema, a set, an object, a subrange, an enumerated type, a pointer to another type_identifier or simply another type_identifier which is to alias. If a schema type is to be declared, *type_identifier* is followed by a discriminant enclosed in parentheses:

```
type_identifier (discriminant) = schema_type_definition;
```

If `value` is specified, followed by a constant satisfying the type definition, every variable of this type is initialized with *constant_expression*, unless it is initialized by `value` itself. The reserved word `value` can be replaced by '`=`', however `value` is not allowed in ISO-Pascal and Borland Pascal, and the replacement by '`=`' is not allowed in Extended Pascal.

Type declaration example

```
type
  { This side is the }     { That side is the }
  { type declaration }     { type definition  }

  Arrayfoo            = array [0 .. 9] of Integer;  { array definition }
  Recordfoo           = record                      { record definition }
                          Bar: Integer;
                        end;

       { schema def with discriminants ``x, y: Integer'' }
  SchemaFoo (x, y: Integer) = array [x .. y] of Integer;
  CharSetFoo          = set of Char;              { Def of a set }
  ObjectFoo           = object                    { Def of an object }
                          procedure DoAction;
                          constructor Init;
                          destructor Done;
                        end;
  SubrangeFoo         = -123..456;                { subrange def }

  EnumeratedFoo       = (Pope,John,the,Second);   { enum type def }
       { Def of a pointer to another type identifier }
  PInteger            = ^arrayfoo;
       { Def of an alias name for another type identifier }
  IdentityFoo         = Integer;
       { Def of an integer which was initialized by 123 }
  InitializedFoo      = Integer value 123;
```

See also

Section 6.2.1 [Type Definition], page 64, Section 6.2 [Data Types], page 64, Section 6.1.5 [Variable Declaration], page 51

6.1.5 Variable Declaration

A variable declaration looks like this:

```
var
  var_identifier: type_identifier;
  ...
  var_identifier: type_identifier;
```

or

```
var
  var_identifier: type_definition;
  ...
  var_identifier: type_definition;
```

and with initializing value:

```
var
  var_identifier: type_identifier value constant_expression;
  ...
  var_identifier: type_identifier value constant_expression;
```

or

```
var
  var_identifier: type_definition value constant_expression;
  ...
  var_identifier: type_definition value constant_expression;
```

A variable declaration part begins with the reserved word `var`. It declares a *var_identifier* whose type either can be specified by a type identifier, or by a type definion which either can be an array, a record, a set, a subrange, an enumerated type or a pointer to an type identifier. If `value` is specified followed by a constant expression satisfying the specified type, the variable declared is initialized with *constant_expression*. The reserved word `value` can be replaced by '`=`', however `value` is not allowed in ISO-Pascal and Borland Pascal, and the replacement by '`=`' is not allowed in Extended Pascal.

See also

Section 6.2.1 [Type Definition], page 64, Section 6.1.4 [Type Declaration], page 50, Section 6.2 [Data Types], page 64, Section 6.1.7.12 [The Declaring Statement], page 59, Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53

6.1.6 Subroutine Declaration

6.1.6.1 The Procedure

```
procedure procedure_identifier;
declaration_part
begin
  statement_part
end;
```

or with a parameter list:

```
procedure procedure_identifier (parameter_list);
declaration_part
begin
  statement_part
end;
```

A procedure is quite like a sub-program: The *declaration_part* consists of label, constant, type, variable or subroutine declarations in free order. The *statement_part* consists of a sequence of statements. If *parameter_list* is specified, parameters can be passed to the procedure and can be used in *statement_part*. A recursive procedure call is allowed.

See also

Section 6.1.6.2 [The Function], page 53, Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53

6.1.6.2 The Function

```
function function_identifier: function_result_type;
declaration_part
begin
  statement_part
end;
```

or with a parameter list:

```
function function_identifier (parameter_list): result_type;
declaration_part
begin
  statement_part
end;
```

A function is a subroutine which has a return value of type *function_result_type*. It is structured like the program: the *declaration_part* consists of label, constant, type, variable or subroutine declarations in free order. The *statement_part* consists of a sequence of statements. If *parameter_list* is specified, parameters can be passed to the function and can be used in *statement_part*. The result is set via an assignment:

```
function_identifier := expression
```

Recursive function calls are allowed. Concerning the result type, ISO 7185 Pascal and Borland Pascal only allow the intrinsic types, subranges, enumerated types and pointer types to be returned. In Extended Pascal, *function_result_type* can be every assignable type. Of course, there are no type restrictions in GNU Pascal as well. If extended syntax is switched on, functions can be called like procedures via procedure call statement.

See also

Section 6.1.6.1 [The Procedure], page 52, Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53, Section 6.2 [Data Types], page 64

6.1.6.3 The Operator

GNU Pascal allows to define operators which can be used the infix style in expressions. For a more detailed description, see Section 6.3 [Operators], page 82

6.1.6.4 Subroutine Parameter List Declaration

```
parameter; ...; parameter
```

Each parameter can start with a prefix (see below) describing how the parameters are passed, followed by a comma seperated list of one or more *parameter_identifiers* and an optional *parameter_type*.

```
procedure DoIt (var x, y, z: OneType; a, b: AnotherType; var q);
```

To understand parameter passing, first some definitions.

actual parameter
: the parameter passed in to the routine.

formal parameter
: the parameter as used inside the procedure.

by value
: the value of the actual parameter is copied on to the stack.

by reference
: the address of the actual parameter is copied on to the stack.

L-value (left hand of a ':=' statement) something that can be assigned to (not a constant, or const or protected variable or other immutable item).

R-value (right hand of a ':=' statement) anything you can get the value of (could be a constant, an expression, a variable (whether const or protected or not) or just about anything.

addressable
: something you can get the address of (not a field of a packed structure or a variable with 'attribute (register)' (GPC extension)).

aliasing accessing memory via two different names (e.g. a global variable passed by reference to a procedure can be accessed either as the global variable or the formal paramater). Generally this is very bad practice.

Technical note: Parameters are not always passed on the stack, they may also be passed in registers, especially on RISC machines.

The prefix defines how a variable is passed on the stack and how you can access the *formal_parameter* inside the procedure. The prefix can be one of:

nothing

```
procedure DoIt (x: SomeType);
```

Technical: The actual parameter is passed by value or reference, but if passed by reference, it is then copied to a local copy on the stack. Aliasing has no effect on x.

What it means: you can modify 'x' inside the routine, but your changes will not affect the actual parameter (and vice versa). The actual parameter can be a constant or other immutable object, or a protected or const variable.

protected

```
procedure DoIt (protected x: SomeType);
```

Technical: The actual parameter is passed by value or reference, but if passed by reference, it is then copied to a local copy on the stack. Aliasing has no effect on x. `protected` is a Extended Pascal extension.

What it means: if you modify the actual parameter, this will not affect 'x' inside the routine. The actual parameter can be a constant or other immutable object, or a protected or const variable. You are forbidden from modifying x inside the routine.

var

```
procedure DoIt (var x: SomeType);
```

Technical: The actual parameter is passed by reference. Aliasing will definitely change 'x'.

What it means: modifications to 'x' inside the routine will change the actual parameter passed in. The actual parameter must be an addressable L-value (ie, it must be something you can take the address of and assign to).

A parameter of this kind is called variable parameter and internally corresponds to an L-value pointer (to the specified type identifier if any). This declaration is necessary if the parameter is to be modified within the routine and to hold its value still after return.

const

```
procedure DoIt (const x: SomeType);
```

Technical: The actual parameter is passed by value or reference. The compiler will make a copy of the actual parameter to have something it can address if the actual parameter is not addressable. You are forbidden from modifying 'x' inside the routine, and therefore you cannot modify the actual parameter. Aliasing may or may not change 'x'. `const` is a Borland Pascal extension.

What it means: You can pass any R-value. You cannot modify 'x' inside the routine. If you change the actual parameter while inside the routine, 'x' will have an undefined value.

protected var

```
procedure DoIt (protected var x: SomeType);
```

Technical: The actual parameter is passed by reference. The compiler will never make a copy of the actual parameter. You are forbidden from modifying 'x' inside the routine, and therefore you cannot modify the actual parameter. Aliasing will definitely change 'x'.

What it means: You can pass anything addressable. You cannot modify 'x' inside the routine. If you change the actual parameter while inside the routine, 'x' will change as well.

In GPC, the `protected var` mode guarantees that the parameter is always passed by reference, making it the correct choice for calling C routines with '`const`' pointer parameters.

If you omit the formal parameter type specification, then any type may be passed to that parameter. Generally this is a bad idea, but occasionally it can be useful, especially for low level code.

As an Extended Pascal extension, you can also declare procedural parameters directly:

`procedure` *parameter_identifier*

or:

`function` *parameter_identifier*`:` *parameter_identifier_result_type*

Example for parameter lists:

```
program ParameterDemo;

procedure Foo (var Bar; var Baz: Integer; const Fred: Integer);

  procedure Glork1 (function Foo: Integer; procedure Bar (Baz: Integer));
  begin
    Bar (Foo)
  end;

begin
  Baz := Integer (Bar) + Fred
end;

var
  a, b, c: Integer;

begin
  Foo (a, b, c)
end.
```

See also

Section 6.2 [Data Types], page 64, [var], page 444, [const], page 296, [protected], page 391

6.1.7 Statements

6.1.7.1 Assignment

The way an assignment looks like:

L-value `:=` *expression*;

This statement assigns any valid expression to *L-value*. Make sure that the result of *expression* is compatible with *L-value*, otherwise an compilation error is reported. The '`:=`' is called assignment operator. As long as *L-value* and *expression* are type compatible, they are assignment compatible for *any definable type* as well.

6.1.7.2 begin end Compound Statement

It looks like that:

```
begin
  statement;
  statement;
  ...
  statement
end
```

This statement joins several *statements* together into one compound statement which is treated as a single statement by the compiler. The finishing semicolon before '`end`' can be left out.

6.1.7.3 if Statement

This statement has the following look:

```
if boolean_expression then
  statement
```

or with an alternative statement:

```
if boolean_expression then
  statement1
else
  statement2
```

The '`if`' ... '`then`' statement consists of a boolean expression and a *statement*, which is conditionally executed if the evaluation of *boolean_expression* yields true.

If '`if`' ... '`then`' ... '`else`' is concerned, *statement1* is executed depending on *boolean_expression* being true, otherwise *statement2* is executed alternatively. Note: the statement before else *must not* finish with a semicolon.

6.1.7.4 case Statement

```
case expression of
  selector: statement;
  ...
  selector: statement;
end
```

or, with alternative statement sequence:

```
case ordinal_expression of
  selector: statement;
  ...
  selector: statement;
otherwise                    { ‘‘else’’ instead of ‘‘otherwise’’ allowed }
  statement;
  ...
  statement;
end
```

or, as part of the invariant `record` type definition:

```
type
  foo = record
    field_declarations
  case bar: variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
  end;
```

or, without a variant selector field,

```
type
  foo = record
    field_declarations
  case variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
  end;
```

The `case` statement compares the value of *ordinal_expression* to each *selector*, which can be a constant, a subrange, or a list of them separated by commas, being compatible with the result of *ordinal_expression*. Note: duplicate selectors or range crossing is not allowed unless {$borland-pascal} is specified. In case of equality the corresponding statement is executed. If `otherwise` is specified and no appropriate selector matched the expression, the series of statements following `otherwise` is executed. As a synonym for `otherwise`, `else` can be used. The semicolon before `otherwise` is optional.

`@@` ???? The expression *must* match one of the selectors in order to continue, unless an alternative statement series is specified.

For `case` in a variant record type definition, see Section 6.2.11.3 [Record Types], page 71.

See also

Section 6.1.7.3 [if Statement], page 56

6.1.7.5 for Statement

For ordinal index variables:

```
for ordinal_variable := initial_value to final_value do
  statement
```

or

```
for ordinal_variable := initial_value downto final_value do
  statement
```

For sets:

```
for set_element_type_variable in some_set do
  statement
```

For pointer index variables:

```
for pointer_variable := initial_address to final_address do
  statement
```

or

```
for pointer_variable := initial_address downto final_address do
  statement
```

The *for* statement is a control statement where an index variable assumes every value of a certain range and for every value the index variable assumes *statement* is executed. The range can be specified by two bounds (which must be of the same type as the index variable, i.e. ordinal or pointers) or by a set.

For ordinal index variables:

- If '`to`' is specified, the index counter is increased by one as long as *initial_value* is less or equal to *final value*,
- if '`downto`' is specified, it is decreased by one as long as *initial_value* is greater or equal to *final_value*.

For pointer index variables:

- If '`to`' is specified, the index counter is increased by the size of the type the index variable points to (if it is a typed pointer, otherwise by one if it is typeless) as long as *initial_address* is less or equal to *final_address*,
- if '`downto`' is specified, it is decreased by a corresponding value as long as *initial_address* is greater or equal to *final_address*.

Since gpc provides a flat memory modell, all addresses are linear, so they can be compared. Still, such loops should be used (if at all) only for iterating through successive elements of an array.

For sets:

- *statement* is executed with the index variable (which must be ordinal and of the same type as the set elements) assuming every element in *some_set*, however note that a set is a not-ordered structure.

Please note: A modification of the index variable may result in unpredictable action.

See also

Section 6.2.11.6 [Set Types], page 76, Section 6.6 [Pointer Arithmetics], page 84, Section 6.1.7.7 [repeat Statement], page 59, Section 6.1.7.5 [for Statement], page 57

6.1.7.6 while Statement

The while loop has the following form

```
while boolean_expression do
  statement
```

The `while` statement declares a loop which is executed while *boolean_expression* is true. Since the terminating condition is checked before execution of the loop body, *statement* may never be executed.

See also

Section 6.1.7.7 [repeat Statement], page 59, Section 6.1.7.5 [for Statement], page 57

6.1.7.7 repeat Statement

```
repeat
  statement;
  ...
  statement;
until boolean_expression
```

The `repeat` ... `until` statement declares a loop which is repeated until *boolean_expression* is true. Since the terminating condition is checked after execution of the loop body, the statement sequence is executed at least once.

See also

Section 6.1.7.6 [while Statement], page 58, Section 6.1.7.5 [for Statement], page 57

6.1.7.8 asm Inline

`@@` ????

```
asm (StatementList: String);
```

The `asm` inline statement is a GNU Pascal extension. It requires its parameter to be AT&T-noted assembler statements, and therefore it is not compatible with that one of Borland Pascal. *statementlist* is a string containing asm statements separated by semicolons.

6.1.7.9 with Statement

6.1.7.10 goto Statement

`@@` ???? This statement looks like this:

```
goto label
```

(Under construction.)

6.1.7.11 Procedure Call

```
subroutine_name;
```

This statement calls the subroutine *subroutine_name* which can either be a procedure or, if GNU extended syntax is turned on, a function. In this case, the result is ignored.

6.1.7.12 The Declaring Statement

This statement allows to declare a variable within a statement part. It looks like this:

```
var
  var_identifier: type_identifier;
```

or

```
var
  var_identifier: type_definition;
```

and with initializing value:

```
var
  var_identifier: type_identifier value expression;
```

or

```
var
  var_identifier: type_definition value expression;
```

Unlike in declaration parts, the initializing *expression* does not have to be a constant expression. Note that every declaring statement has to start with `var`. The name space of the variable extends from its declaration to the end of the current matching statement sequence (which can be a statement part (of the program, a function, a procedure or an operator) or, within that part, a begin end compound statement, a repeat loop, or the else branch of a case statement). This statement is a GNU Pascal extension.

See also

Section 6.2.1 [Type Definition], page 64, Section 6.2 [Data Types], page 64

6.1.7.13 Loop Control Statements

These are

```
Continue;
```

and

```
Break;
```

These simple statements *must not* occur outside a loop, i.e. a '`for`', '`while`' or '`repeat`' statement. '`Continue`' transfers control to the beginning of the loop right by its call, '`Break`' exits the current loop turn and continues loop execution.

6.1.8 Import Part and Module/Unit Concept

6.1.8.1 The Source Structure of ISO 10206 Extended Pascal Modules

`@@` Description missing here

A module can have one or more '`export`' clauses and the name of an '`export`' clause doesn't have to be equal to the name of the module.

Sample module code with separate `interface` and `implementation` parts:

```
module DemoModule interface;  { interface part }

export DemoModule = (FooType, SetFoo, GetFoo);

type
  FooType = Integer;

procedure SetFoo (f: FooType);
function  GetFoo: FooType;

end.

module DemoModule implementation;  { implementation part }

import
  StandardInput;
  StandardOutput;

var
```

```
    Foo: FooType;

  { Note: the effect is the same as a ‘forward’ directive would have:
    parameter lists and result types are not allowed in the
    declaration of exported routines, according to EP. In GPC, they
    are allowed, but not required. }
  procedure SetFoo;
  begin
    Foo := f
  end;

  function GetFoo;
  begin
    GetFoo := Foo
  end;

  to begin do
    begin
      Foo := 59;
      WriteLn ('Just an example of a module initializer. See comment below')
    end;

  to end do
    begin
      Foo := 0;
      WriteLn ('Goodbye')
    end;

  end.
```

Alternatively the module interface and implementation may be combined as follows:

```
  module DemoMod2;  { Alternative method }

  export Catch22 = (FooType, SetFoo, GetFoo);

  type
    FooType = Integer;

  procedure SetFoo (f: FooType);
  function  GetFoo: FooType;

  end; { note: this ‘end’ is required here, even if the
         module-block below would be empty. }

  var
    Foo: FooType;

  procedure SetFoo;
  begin
    Foo := f
  end;
```

```
function GetFoo;
begin
  GetFoo := Foo
end;

end.
```

Either one of the two methods may be used like this:

```
program ModuleDemo (Output);

import DemoModule;

begin
  SetFoo (999);
  WriteLn (GetFoo);
end.
```

```
program ModDemo2 (Output);

import Catch22 in 'demomod2.pas';

begin
  SetFoo (999);
  WriteLn (GetFoo);
end.
```

Somewhat simpler GPC modules are also supported. **Please note:** This is not supported in the Extended Pascal standard.

This is a simpler module support that does not require exports, imports, module headers etc.

These non-standard simple GPC modules look like the following example. They do not have an export part, do not have a separate module-block, do not use import/export features.

Instead, you have to emulate the exporting/importing yourself using '`attribute`' and '`external name`'.

```
module DemoMod3;

type
  FooType = Integer;

var
  Foo: FooType;

procedure SetFoo (f: FooType); attribute (name = 'SetFoo');
begin
  Foo := f
end;

function GetFoo: FooType; attribute (name = 'GetFoo');
begin
  GetFoo := Foo;
end;

end.
```

```
program ModDemo3 (Output);
```

```
{$L demomod3.pas}  { explicitly link module }

{ Manually do the "import" from DemoMod3 }
type
  FooType = Integer;

procedure SetFoo (f: FooType); external name 'SetFoo';
function  GetFoo: FooType;     external name 'GetFoo';

begin
  SetFoo (999);
  WriteLn (GetFoo)
end.
```

Module initialization and finalization:

The `to begin do` module initialization and `to end do` module finalization constructs now work on *every* target.

By the way: The "GPC specific" module definition is almost identical to the PXSC standard. With an additional keyword 'global' which puts a declaration into an export interface with the name of the module, it will be the same. `@@`This is planned.

6.1.8.2 The Source Structure of UCSD/Borland Pascal Units

A generic GNU Pascal unit looks like the following:

```
unit name;

interface

import_part

interface_part

implementation

implementation_part

initialization_part

end.
```

The *name* of the unit should coincide with the name of the file with the extension stripped. (If not, you can tell GPC the file name with '`uses foo in 'bar.pas'`', see [uses], page 441.)

The *import_part* is either empty or contains a '`uses`' clause to import other units. It may also consist of an ISO-style '`import`' specification. Note that the implementation part is not preceeded by a second import part in GPC (see [import], page 336).

The *interface_part* consists of constant, type, and variable declarations, procedure and function headings which may be freely mixed.

The *implementation_part* is like the declaration part of a program, but the headers of procedures and functions may be abbreviated: Parameter lists and function results may be omitted for procedures and functions already declared in the interface part.

The *initialization_part* may be missing, or it may be a '`begin`' followed by one or more statements, such that the unit has a statement part between this '`begin`' and the final '`end`'.

Alternatively, a unit may have ISO-style module initializers and finalizers, see [to begin do], page 433, [to end do], page 433.

Note that GPC does *not* yet check whether all interface declarations are resolved in the same unit. The implementation of procedures and functions which are in fact not used may be omitted, and/or procedures and functions may be implemented somewhere else, even in a different language. However, relying on a GPC bug (that will eventually be fixed) is not a good idea, so this is not recommended. Instead, declare such routines as '`external`'.

A unit exports everything declared in the interface section. The exported interface has the name of the unit and is compatible with Extended Pascal module interfaces since GPC uses the same code to handle both.

6.2 Data Types

6.2.1 Type Definition

As described in Section 6.1.4 [Type Declaration], page 50, a type declaration part looks like this:

```
type
  type_identifier = type_definition;
  ...
  type_identifier = type_definition;
```

where the left side is the type declaration and the right one the type definition side. GNU Pascal offers various possibilities to implement highly specialized and problem-specific data types.

6.2.2 Ordinal Types

An ordinal type is one that can be mapped to a range of whole numbers. It includes integer types, character types, enumerated types and subrange types of them.

A character type is represented by the intrinsic type '`Char`' which can hold elements of the operating system's character set (usually ASCII). Conversion between character types and integer types is possible with the intrinsic functions `Ord` and `Chr`.

An enumerated type defines a range of elements which are referred to by identifiers. Conversion from enumerated types to integer types is possible with the intrinsic function `Ord`. Conversion from integer to ordinal types is only possible by type-casting or using the extended form of '`Succ`'.

```
var
  Foo: Char;       { foo can hold a character }
  Num: '0' .. '9'; { Can hold decimal digits, is a subrange type of Char }
  Day: (Monday, Tuesday, Wednesday, Thursday,
        Friday, Saturday, Sunday);  { Can hold weekdays }
```

See also

[Ord], page 377, [Chr], page 290, Section 6.7 [Type Casts], page 85

6.2.3 Integer Types

Besides '`Integer`', GNU Pascal supports a large zoo of integer types. Some of them you will find in other compilers, too, but most are GNU Pascal extensions, introduced for particular needs. Many of these types are synonyms for each other. In total, GPC provides 20 built-in integer types, plus seven families you can play with. (Four of these "families" are signed and unsigned, packed and unpacked subrange types; the others are explained below.)

See also: Section 6.2.11.1 [Subrange Types], page 70.

6.2.3.1 The CPU's Natural Integer Types

For most purposes, you will always use '`Integer`', a signed integer type which has the "natural" size of such types for the machine. On most machines GPC runs on, this is a size of 32 bits, so '`Integer`' usually has a range of '`-2147483648..2147483647`' (see [Integer], page 343).

If you need an unsigned integer type, the "natural" choice is '`Cardinal`', also called '`Word`'. Like '`Integer`', it has 32 bits on most machines and thus a range of '`0..4294967295`' (see [Cardinal], page 285, [Word], page 448).

These natural integer types should be your first choice for best performance. For instance on an IA32 CPU operations with '`Integer`' usually work faster than operations with shorter integer types like '`ShortInt`' or '`ByteInt`' (see below).

6.2.3.2 The Main Branch of Integer Types

'`Integer`', '`Cardinal`', and '`Word`' define the three "main branches" of GPC's integer types. You won't always be able to deal with the natural size; sometimes something smaller or longer will be needed. Especially when interfacing with libraries written in other languages such as C, you will need equivalents for their integer types.

The following variants of integer types (plus one Boolean type) are guaranteed to be compatible to the respective types of GNU C as listed below (whereas '`Integer`', '`Cardinal`' and '`Word`' themselves are *not* guaranteed to be compatible to any given C type). The sizes given, however, are *not* guaranteed. They are just typical values currently used on some platforms, but they may be actually shorter or longer on any given platform.

signed	**unsigned**	**also unsigned**	**GNU C equivalent**	**size in bits (example)**
`ByteInt`	`ByteCard`	`Byte`	`[un]signed char`	8
`ShortInt`	`ShortCard`	`ShortWord`	`[unsigned] short int`	16
`CInteger`	`CCardinal`	`CWord`	`[unsigned] int`	32
`MedInt`	`MedCard`	`MedWord`	`[unsigned] long int`	32
`LongInt`	`LongCard`	`LongWord`	`[unsigned] long long int`	64
—	`SizeType`	—	`size_t`	32
`PtrDiffType`	—	—	`ptrdiff_t`	32
`PtrInt`	`PtrCard`	`PtrWord`	—	32
—	`CBoolean`	—	`_Bool`, `bool`	8

Since we don't know whether '`LongInt`' will always remain the "longest" integer type available – maybe GNU C will get '`long long long int`', one day, which we will support as '`LongLongInt`' – we have added the synonym '`LongestInt`' for the longest available singed integer type, and the same holds for '`LongestCard`' and '`LongestWord`'.

6.2.3.3 Integer Types with Specified Size

In some situations you will need an integer type of a well-defined size. For this purpose, GNU Pascal provides type attributes (see [attribute], page 274). The type

```
Integer attribute (Size = 42)
```

is guaranteed to have a precision of 42 bits. In a realistic context, you will most often give a power of two as the number of bits, and the machine you will need it on will support variables of that size. If this is the case, the specified precision will simultaneously be the amount of storage needed for variables of this type.

In short: If you want to be sure that you have a signed integer with 32 bits width, write '`Integer attribute (Size = 32)`', not just '`Integer`' which might be bigger. The same works with unsigned integer types such as '`Cardinal`' and '`Word`' and with Boolean types.

This way, you *can't* get a higher precision than that of '`LongestInt`' or '`LongestCard`' (see Section 6.2.3.2 [Main Branch Integer Types], page 65). If you need higher precision, you can look at the '`GMP`' unit (see Section 6.15.5 [GMP], page 177) which provides integer types with arbitrary precision, but their usage is different from normal integer types.

6.2.3.4 Integer Types and Compatibility

If you care about ISO compliance, *only* use '`Integer`' and subranges of '`Integer`'.

Some of GPC's non-ISO integer types exist in Borland Pascal, too: '`Byte`', '`ShortInt`', '`Word`', and '`LongInt`'. The sizes of these types, however, are not the same as in Borland Pascal. Even for '`Byte`' this is not guaranteed (while probable, though).

When designing GNU Pascal, we thought about compatibility to Borland Pascal. Since GNU Pascal is (at least) a 32-bit compiler, '`Integer`' *must* have (at least) 32 bits. But what to do with '`Word`'? Same size as '`Integer`' (like in BP) or 16 bits (like in BP)? We decided to make '`Word`' the "natural-sized" unsigned integer type, thus making it (at least) 32 bits wide. Similarly, we decided to give '`LongInt`' twice the size of '`Integer`' (like in BP) rather than making it 32 bits wide (like in BP). So '`LongInt`' has 64 bits, and '`ShortInt`' has 16 bits on the IA32 platform.

On the other hand, to increase compatibility to Borland Pascal and Delphi, GPC provides the alias name '`Comp`' for '`LongInt`' (64 bits on IA32) and '`SmallInt`' for '`ShortInt`' (16 bits on IA32). Note that BP treats '`Comp`' as a "real" type and allows assignments like '`MyCompVar := 42.0`'. Since we don't consider this a feature, GPC does not copy this behaviour.

6.2.3.5 Summary of Integer Types

Here is a summary of all integer types defined in GPC. The sizes and ranges are only *typical* values, valid on some, but not all platforms. Compatibility to GNU C however *is* guaranteed.

[ByteInt], page 283
: signed 8-bit integer type, '`-128..128`',
compatible to '`signed char`' in GNU C.

[ByteCard], page 283
: unsigned 8-bit integer type, '`0..255`',
compatible to '`unsigned char`' in GNU C.

[ShortInt], page 417
: signed 16-bit integer type, '`-32768..32767`',
compatible to '`short int`' in GNU C.

[ShortCard], page 417
: unsigned 16-bit integer type, '`0..65535`',
compatible to '`unsigned short int`' in GNU C.

[Integer], page 343
: signed 32-bit integer type, '`-2147483648..2147483647`',
compatible to '`int`' in GNU C.

[Cardinal], page 285
: unsigned 32-bit integer type, '`0..4294967295`',
compatible to '`unsigned int`' in GNU C.

[MedInt], page 361
: signed 32-bit integer type, '-2147483648..2147483647', compatible to 'long int' in GNU C.

[MedCard], page 360
: unsigned 32-bit integer type, '0..4294967295', compatible to 'unsigned long int' in GNU C.

[LongInt], page 354
: signed 64-bit integer type, '-9223372036854775808..9223372036854775807', compatible to 'long long int' in GNU C.

[LongCard], page 350
: unsigned 64-bit integer type, '0..18446744073709551615', compatible to 'unsigned long long int' in GNU C.

[LongestInt], page 352
: signed 64-bit integer type, '-9223372036854775808..9223372036854775807'.

[LongestCard], page 352
: unsigned 64-bit integer type, '0..18446744073709551615'.

[Comp], page 293
: signed 64-bit integer type, '-9223372036854775808..9223372036854775807'.

[SmallInt], page 422
: signed 16-bit integer type, '-32768..32767'.

[SizeType], page 422
: integer type (usually unsigned) to represent the size of objects in memory

[PtrDiffType], page 392
: signed integer type to represent the difference between two positions in memory

[PtrInt], page 393
: signed integer type of the same size as a pointer

[PtrCard], page 392
: unsigned integer type of the same size as a pointer

To specify the number of bits definitely, use type attributes, [attribute], page 274.

```
program IntegerTypesDemo (Output);

var
  ByteVar: Byte;
  ShortIntVar: ShortInt;
  Foo: MedCard;
  Big: LongestInt;

begin
  ShortIntVar := 1000;
  Big := MaxInt * ShortIntVar;
  ByteVar := 127;
  Foo := 16#deadbeef
end.
```

See also: Section 6.2.11.1 [Subrange Types], page 70.

6.2.4 Built-in Real (Floating Point) Types

GPC has three built-in floating point types to represent real numbers. Each of them is available under two names (for compatibility to other compilers and languages).

For most purposes, you will always use '`Real`' which is the only one of them that is part of Standard and Extended Pascal. If memory constraints apply, you might want to choose '`ShortReal`' for larger arrays. On the other hand, if high precision is needed, you can use '`LongReal`'. When interfacing with libraries written in other languages such as C, you will need the equivalents for their real types.

Note that not all machines support longer floating point types, so '`LongReal`' is the same as '`Real`' on these machines. Also, some machines may support a longer type, but not do all arithmetic operations (e.g. the '`Sin`' function, [Sin], page 420) in a precision higher than that of '`Real`'. If you need higher precision, you can look at the '`GMP`' unit (see Section 6.15.5 [GMP], page 177) which provides rational and real numbers with arbitrary precision, but their usage is different from normal real types.

The following real types are guaranteed to be compatible to the real types of GNU C. The sizes given, however, are *not* guaranteed. They are just typical values used on any IEEE compatible floating point hardware, but they may be different on some machines.

type name	alternative name	GNU C equivalent	size in bits (typically)
`ShortReal`	`Single`	`float`	32
`Real`	`Double`	`double`	64
`LongReal`	`Extended`	`long double`	80

6.2.5 Strings Types

There are several ways to use strings in GNU Pascal. One of them is the use of the intrinsic string type '`String`' which is a predefined schema type. The schema discriminant of this type holds the maximal length, which is of type Integer, so values up to `MaxInt` can be specified. For '`String`', an assignment is defined. There are many built-in functions and procedures for comfortable strings handling.

`@@` ???? String procedures and functions.

Another way to use strings is to use arrays of type '`Char`'. For these, an intrinsic assignment is defined as well. Besides, '`String`' and '`Char`' are assignment compatible. The preferred way, however, is '`String`' because of the numerous possibilities for string handling.

6.2.6 Character Types

Character types are a special case of ordinal types.

See also

Section 6.2.2 [Ordinal Types], page 64, [Chr], page 290, [Ord], page 377, [Pred], page 388, [Succ], page 428.

6.2.7 Enumerated Types

```
type
  enum_type_identifier = (identifier, ..., identifier);
```

An enumerated type is a a special case of ordinal types and defines a range of elements which are referred to by identifiers. Enumerated types are ordered by occurence in the identifier list. So, they can be used as index types in an array definition, and it is possible to define subranges of them. Since they are ordered, they can be compared to one another. The intrinsic function

`Ord` applied to *name_identifier* returns the number of occurence in the identifier list (beginning with zero), `Pred` and `Succ` return the predecessor and successor of *name_identifier*. '`Boolean`' is a special case of an enumerated type.

See also

Section 6.2.2 [Ordinal Types], page 64, Section 6.2.11.2 [Array Types], page 70, Section 6.2.11.1 [Subrange Types], page 70, [Ord], page 377, [Boolean], page 280, [Char], page 288, [Pred], page 388, [Succ], page 428.

6.2.8 File Types

Files are used to store data permanently, normally on hard drives or floppies. There are tree types of files available: text files, typed and untyped files.

Text files are used to store text in them, where typed files are used to store many entries of the same type in them, e.g. addresses. Text files and typed files are accessible by '`Read`' and '`Write`' operations and do not need the parameter '`BlockSize`' in '`Reset`' or '`Rewrite`'. On the other hand, untyped files are used to store any type of information in them but you need to use '`BlockWrite`' or '`BlockRead`' to store or retrieve data out of this file.

```
var
  F1: Text;   { a textfile }
  F2: file of Real;   { a typed filed used to store real values in it }
  F3: File;   { an untyped file }
```

See also

Section 6.10.1 [File Routines], page 92, [Write], page 449, [Read], page 397, [BlockRead], page 279, [BlockWrite], page 280, [Reset], page 402, [Rewrite], page 405

6.2.9 Boolean (Intrinsic)

The intrinsic Boolean represents boolean values, i.e. it can only assume true and false (which are predefined constants). This type corresponds to the enumerated type

```
type
  Boolean = (False, True);
```

Since it is declared this way, it follows:

```
Ord (False) = 0
Ord (True) = 1
False < True
```

There are four intrinsic logical operators. The logical `and`, `or` and `not`. In Borland Pascal and GNU Pascal, there is a logical "exclusive or" `xor`.

See also

Section 6.2.7 [Enumerated Types], page 68, [and], page 262, [or], page 375, [not], page 371, [xor], page 451

6.2.10 Pointer (Intrinsic)

The intrinsic Pointer Type is a so-called unspecified or typeless pointer (i.e. a pointer which does not point to any type but holds simply a memory address).

See also

Section 6.2.11.7 [Pointer Types], page 76, [nil], page 370

6.2.11 Type Definition Possibilities

6.2.11.1 Subrange Types

GNU Pascal supports Standard Pascal's subrange types:

```
program SubrangeDemo;
type
  MonthInt = 1 .. 12;
  Capital = 'A' .. 'Z';
  ControlChar = ^A .. ^Z;  { `^A' = `Chr (1)' is a BP extension }
begin
end.
```

Also possible: Subranges of enumerated types:

```
program EnumSubrangeDemo;
type
  { This is an enumerated type. }
  Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  { This is a subrange of `Days'. }
  Working = Mon .. Fri;

begin
end.
```

To increase performance, variables of such a type are aligned in a way which makes them fastest to access by the CPU. As a result, '1 .. 12' occupies 4 bytes of storage on an IA32 CPU.

For the case you want to save storage at the expense of speed, GPC provides a 'packed' variant of these as an extension:

```
program PackedSubrangeDemo;
type
  MonthInt = packed 1 .. 12;
begin
end.
```

A variable of this type occupies the shortest possible (i.e., addressable) space in memory – one byte on an IA32 compatible CPU.

See also: [packed], page 381.

6.2.11.2 Array Types

```
type
  array_type_identifier = array [index_type] of element_type
```

or

```
type
  array_type_identifier = array [index_type, ..., index_type] of element_type
```

The reserved word `array` defines an array type. *index_type* has to be an ordinal type, subrange type or an enumerated type, where several index types, separated by commas, are allowed.

element_type is an arbitrary type. An element of an array is accessed by *array_type_variable* [*index_number*]. The upper and lower index bounds can be determined by the intrinsic functions `High` and `Low`.

```
type
  IntArray = array [1 .. 20] of Integer;
  Foo      = array [(Mo, Tu, We, Th, Fr, Sa, Su)] of Char;
  Bar      = array [0 .. 9, 'a' .. 'z', (Qux, Glork1, Fred)] of Real;
  Baz1     = array [1 .. 10] of IntArray;
  { equal (but declared differently): }
  Baz2     = array [1 .. 10, 1 .. 20] of Integer;
```

See also

[High], page 333, [Low], page 356

6.2.11.3 Record Types

```
type
  record_type_identifier = record
    field_identifier: type_definition;
    ...
    field_identifier: type_definition;
  end;
```

or, with a variant part,

```
type
  record_type_identifier = record
    field_identifier: type_definition;
    ...
    field_identifier: type_definition;
  case bar: variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
  end;
```

or, without a variant selector field,

```
type
  record_type_identifier = record
    field_identifier: type_definition;
    ...
    field_identifier: type_definition;
  case variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
  end;
```

The reserved word `record` defines a structure of fields. Records can be '`packed`' to save memory usage at the expense of speed.

The reserved word '`record`' and record types are defined in ISO 7185 Pascal. According to ISO Pascal, the variant type must be an identifier. GNU Pascal, like UCSD and Borland Pascal, also allows a subrange here.

A record field is accessed by *record_type_variable . field_identifier*

See also: [packed], page 381, Section 6.1.7.4 [case Statement], page 56.

6.2.11.4 Variant Records

GPC supports variant records like in EP and BP. The following construction is not allowed in Extended Pascal, but in BP and GPC:

```
program BPVariantRecordDemo;

type
  PersonRec = record
    Age: Integer;
  case EyeColor: (Red, Green, Blue, Brown) of
    Red, Green : (WearsGlasses: Boolean);
    Blue, Brown: (LengthOfLashes: Integer);
  end;

begin
end.
```

In EP, the variant field needs a type identifier, which, of course, also works in GPC:

```
program EPVariantRecordDemo;

type
  EyeColorType = (Red, Green, Blue, Brown);

  PersonRec = record
    Age: Integer;
  case EyeColor: EyeColorType of
    Red, Green : (WearsGlasses: Boolean);
    Blue, Brown: (LengthOfLashes: Integer);
  end;

begin
end.
```

6.2.11.5 EP's Schema Types including 'String'

Schemata are types that depend on one or more variables, called *discriminants*. They are an ISO 10206 Extended Pascal feature.

```
type
  RealArray (n: Integer) = array [1 .. n] of Real;
  Matrix (n, m: PositiveInteger) = array [1 .. n, 1 .. m] of Integer;
```

The type '`RealArray`' in this example is called a Schema with the discriminant '`n`'.

To declare a variable of such a type, write:

```
var
  Foo: RealArray (42);
```

The discriminants of every global or local schema variable are initialized at the beginning of the procedure, function or program where the schema variable is declared.

Schema-typed variables "know" about their discriminants. Discriminants can be accessed just like record fields:

```
program Schema1Demo;
type
  PositiveInteger = 1 .. MaxInt;
  RealArray (n: Integer) = array [1 .. n] of Real;
  Matrix (n, m: PositiveInteger) = array [1 .. n, 1 .. m] of Integer;

var
  Foo: RealArray (42);

begin
  WriteLn (Foo.n)  { yields 42 }
end.
```

Schemata may be passed as parameters. While types of schema variables must always have specified discriminants (which may be other variables), formal parameters (by reference or by value) may be of a schema type without specified discriminant. In this, the actual parameter may posses any discriminant. The discriminants of the parameters get their values from the actual parameters.

Also, *pointers* to schema variables may be declared without a discriminant:

```
program Schema2Demo;
type
  RealArray (n: Integer) = array [1 .. n] of Real;
  RealArrayPtr = ^RealArray;
var
  Bar: RealArrayPtr;
begin
end.
```

When applying '`New`' to such a pointer, you must specify the intended value of the discriminant as a parameter:

```
New (Bar, 137)
```

As a GNU Pascal extension, the above can also be written as

```
Bar := New (RealArrayPtr, 137)
```

The allocated variable behaves like any other schema variable:

```
program Schema3Demo;
type
  RealArray (n: Integer) = array [1 .. n] of Real;
  RealArrayPtr = ^RealArray;
var
  Bar: RealArrayPtr;
  i: Integer;
begin
  Bar := New (RealArrayPtr, 137);
  for i := 1 to Bar^.n do
    Bar^[i] := 42
end.
```

Since the schema variable "knows" its size, pointers to schemata can be disposed just like other pointers:

```
Dispose (Bar)
```

Schemata are not limited to arrays. They can be of any type that normally requires constant values in its definition, for instance subrange types, or records containing arrays etc. (Sets do not yet work.)

References to the schema discriminants are allowed, and the `with` statement is also allowed, so one can say:

```
program SchemaWithDemo;
type
  RealArray (n: Integer) = array [1 .. n] of Real;
var
  MyArray: RealArray (42);
begin
  WriteLn (MyArray.n);  { writes 42 }
  with MyArray do
    WriteLn (n);        { writes 42 }
end.
```

Finally, here is a somewhat exotic example. Here, a '`ColoredInteger`' behaves just like an ordinary integer, but it has an additional property '`Color`' which can be accessed like a record field.

```
program SchemaExoticDemo;

type
  ColorType = (Red, Green, Blue);
  ColoredInteger (Color: ColorType) = Integer;

var
  Foo: ColoredInteger (Green);

begin
  Foo := 7;
  if Foo.Color = Red then
    Inc (Foo, 2)
  else
    Foo := Foo div 3
end.
```

An important schema is the predefined '`String`' schema (according to Extended Pascal). It has one predefined discriminant identifier `Capacity`. GPC implements the `String` schema as follows:

```
type
  String (Capacity: Cardinal) = record
    Length: 0 .. Capacity;
    Chars: packed array [1 .. Capacity + 1] of Char
  end;
```

The `Capacity` field may be directly referenced by the user, the `Length` field is referenced by a predefined string function `Length (Str)` and contains the current string length. `Chars` contains the chars in the string. The `Chars` and `Length` fields cannot be directly referenced by a user program.

If a formal value parameter is of type '`String`' (with or without discriminant), the actual parameter may be either a `String` schema, a fixed string (character array), a single character, a string literal or a string expression. If the actual parameter is a '`String`' schema, it is copied for the parameter in the usual way. If it is not a schema, a '`String`' schema is created automatically, the actual parameter is copied to the new variable and the `Capacity` field of the new variable is set to the length of the actual parameter.

Actual parameters to '`var`' parameters of type '`String`' must be '`String`' schemata, not string literals or character arrays.

```
program StringDemo (Output);

type
  SType = String (10);
  SPtr  = ^String;

var
  Str : SType;
  Str2: String (100000);
  Str3: String (20) value 'string expression';
  DStr: ^String;
  ZStr: SPtr;
  Len : Integer value 256;
  Ch  : Char value 'R';

{ `String' accepts any length of strings }
procedure Foo (z: String);
begin
  WriteLn ('Capacity: ', z.Capacity);
  WriteLn ('Length  : ', Length (z));
  WriteLn ('Contents: ', z);
end;

{ Another way to use dynamic strings }
procedure Bar (SLen: Integer);
var
  LString: String (SLen);
  FooStr: type of LString;
begin
  LString := 'Hello world!';
  Foo (LString);
  FooStr := 'How are you?';
  Foo (FooStr);
end;

begin
  Str  := 'KUKKUU';
  Str2 := 'A longer string variable';
  New (DStr, 1000);  { Select the string Capacity with `New' }
  DStr^ := 'The maximum length of this is 1000 chars';
  New (ZStr, Len);
  ZStr^ := 'This should fit here';
  Foo (Str);
  Foo (Str2);
  Foo ('This is a constant string');
  Foo ('This is a ' + Str3);
  Foo (Ch);  { A char parameter to string routine }
  Foo ('');  { An empty string }
  Foo (DStr^);
  Foo (ZStr^);
  Bar (10000);
```

```
end.
```

In the above example, the predefined procedure `New` was used to select the capacity of the strings. Procedure `Bar` also has a string whose size depends of the parameter passed to it and another string whose type will be the same as the type of the first string, using the `type of` construct.

All string and character types are compatible as long as the destination string is long enough to hold the source in assignments. If the source string is shorter than the destination, the destination is automatically blank padded if the destination string is not of string schema type.

6.2.11.6 Set Types

set_type_identifier `= set of` *set_element_type*`;`

set_type_identifier is a set of elements from *set_element_type* which is either an ordinal type, an enumerated type or a subrange type. Set element representatives are joined together into a set by brackets:

`[`*set_element*`, ..., `*set_element*`]`

'`[]`' indicates the empty set, which is compatible with all set types. Note: Borland Pascal restricts the maximal set size (i.e. the range of the set element type) to 256, GNU Pascal has no such restriction. The number of elements a set variable is holding can be determined by the intrinsic set function Card (which is a GNU Pascal extension, in Extended Pascal and Borland Pascal you can use SizeOf instead but note the element type size in bytes, then) to the set. There are four intrinsic binary set operations: the union '`+`', the intersection '`*`' and the difference '`-`'. The symmetric difference '`><`' is an Extended Pascal extension.

See also

[Card], page 284, [SizeOf], page 421

6.2.11.7 Pointer Types

pointer_type_identifier `= ^`*type_identifier*`;`

A pointer of the type *pointer_type_identifier* holds the address at which data of the type *type_identifier* is situated. Unlike other identifier declarations, where all identifiers in definition part have to be declared before, in a pointer type declaration *type_identifier* may be declared after *pointer_type_identifier*. The data pointed to is accessed by '*pointer_type_variable*`^`'. To mark an unassigned pointer, the '`nil`' constant (which stands for "not in list") has to be assigned to it, which is compatible with all pointer types.

```
type
  ItselfFoo = ^ItselfFoo;  { possible but mostly senseless }

  PInt      = ^Integer;    { Pointer to an Integer }

  PNode     = ^TNode;      { Linked list }
  TNode     = record
    Key     : Integer;
    NextNode: PNode;
  end;

var
  Foo, Bar: PInt;
```

```
begin
  Foo := Bar;  { Modify address which foo is holding }
  Foo^ := 5;   { Access data foo is pointing to }
end.
```

GPC also suports pointers to procedures or function and calls through them. This is a non-standard feature.

```
program ProcPtrDemo (Output);

type
  ProcPtr = ^procedure (i: Integer);

var
  PVar: ProcPtr;

procedure WriteInt (i: Integer);
begin
  WriteLn ('Integer: ', i : 1)
end;

begin
  { Let PVar point to function WriteInt }
  PVar := @WriteInt;

  { Call the function by dereferencing the function pointer }
  PVar^ (12345)
end.
```

See also: Section 6.2.10 [Pointer (Intrinsic)], page 69.

6.2.11.8 Procedural and Functional Types

For procedures without a parameter list:

procedure_type_identifier `= procedure` *name_identifier*`;`

or functions:

function_type_identifier `=`
        `function` *name_identifier*`:` *function_result_type*`;`

For procedures with a parameter list:

procedure_type_identifier `=`
        `procedure` *name_identifier* `(`*parameter_list*`);`

or functions:

function_type_identifier `=`
        `function` *name_identifier* `(`*parameter_list*`):` *function_result_type*`;`

Procedural types can be used as procedures or functions respectively, but also a value can be assigned to them. Procedural types are a Borland Pascal extension. In Borland Pascal, *function_result_type* can only be one of these types: ordinal types, real types, pointer types, the intrinsic '`String`' type. In GNU Pascal every function result type for procedural types is allowed.

BP has procedural and functional types:

```
type
  CompareFunction = function (Key1, Key2: String): Integer;
```

```
function Sort (Compare: CompareFunction);
begin
  ...
end;
```

Standard Pascal has procedural and functional parameters:

```
function Sort (function Compare (Key1, Key2: String): Integer);
begin
  ...
end;
```

Both ways have pros and cons, e.g. in BP you can save, compare, trade, etc. procedural values, or build arrays of them, while the SP way does not require a type declaration and prevents problems with uninitialized or invalid pointers (which in BP will usually crash the program).

GPC supports both ways. An important feature of Standard Pascal (but not BP) that GPC also supports is the possibility to pass *local* routines as procedural or functional parameters, even if the called routine is declared far remote. The called routine can then call the passed local routine and it will have access to the original caller's local variables.

```
program LocalProceduralParameterDemo;

procedure CallProcedure (procedure Proc);
begin
  Proc
end;

procedure MainProcedure;
var LocalVariable: Integer;

  procedure LocalProcedure;
  begin
    WriteLn (LocalVariable)
  end;

begin
  LocalVariable := 42;
  CallProcedure (LocalProcedure)
end;

begin
  MainProcedure
end.
```

See also: Section 6.1.6.1 [The Procedure], page 52, Section 6.1.6.2 [The Function], page 53, Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53, Section 6.1.7.11 [Procedure Call], page 59.

6.2.11.9 Object Types

Object types are used to encapsulate data and methods. Furthermore, they implement a mechanism for inheritance.

See also

Section 6.8 [OOP], page 86

6.2.11.10 Initial values to type denoters

A type may be initialized to a value of expression when it is declared, like a variable, as in:

```
program TypeVarInitDemo;
type
  Int10   = Integer value 10;
  FooType = Real;
  MyType  = Char value Pred ('A');
  EType   = (a, b, c, d, e, f, g) value d;

const
  Answer = 42;

var
  ii : Int10;                    { Value of ii set to 10 }
  ch : MyType  value Pred ('z');
  aa : Integer value Answer + 10;
  foo: FooType value Sqr (Answer);
  e1 : EType;                    { value set to d }
  e2 : EType value g;            { value set to g }
begin
end.
```

Extended Pascal requires the type initializers to be constant expressions. GPC allows any valid expression.

Note, however, that the expressions that affect the size of storage allocated for objects (e.g. the length of arrays) may contain variables only inside functions or procedures.

GPC evaluates the initial values used for the type when an identifier is declared for that type. If a variable is declared with a type-denoter that uses a type-name which already has an initial value the latter initialization has precedence.

`@@` GPC does not know how to calculate constant values for math functions in the runtime library at compile time, e.g. '`Exp (Sin (2.4567))`', so you should not use these kind of expressions in object size expressions. (Extended Pascal allows this.)

6.2.11.11 Restricted Types

GPC supports '`restricted`' types, defined in Extended Pascal. A value of a restricted type may be passed as a value parameter to a formal parameter possessing its underlying type, or returned as the result of a function. A variable of a restricted type may be passed as a variable parameter to a formal parameter possessing the same type or its underlying type. No other operations, such as accessing a component of a restricted type value or performing arithmetic, are possible.

```
program RestrictedDemo;

type
  UnrestrictedRecord = record
    a: Integer;
  end;
```

```
  RestrictedRecord = restricted UnrestrictedRecord;

var
  r1: UnrestrictedRecord;
  r2: RestrictedRecord;
  i: restricted Integer;
  k: Integer;

function AccessRestricted (p: UnrestrictedRecord): RestrictedRecord;
var URes: UnrestrictedRecord;
begin
  { The parameter is treated as unrestricted, even though the actual
    parameter may be restricted }
  URes.a := p.a;
  { It is allowed to assign a function result }
  AccessRestricted := URes;
end;

begin
  r1.a := 354;

  { Assigning a restricted function result to a restricted variable }
  { @@ Verify if this should really be allowed????? }
  r2 := AccessRestricted (r1);

  { Passing a restricted value to unrestricted formal parameter is ok }
  r2 := AccessRestricted (r2);

  {$ifdef BUG}
  { *** The following statements are not allowed *** }
  k := r2.a;      { field access (reading) }
  r2.a := 100;    { field access (writing) }
  r1 := r2;       { assignment source is restricted }
  r2 := r1;       { assignment target is restricted }
  r1 := AccessRestricted (r2); { assigning a restricted function
                                 result to an unrestricted object }
  i  := 16#ffff;  { assignment target is restricted }
  k  := i + 2;    { arithmetic with restricted value }
  {$endif}
end.
```

6.2.12 Machine-dependencies in Types

6.2.12.1 Endianness

Endianness means the order in which the bytes of a value larger than one byte are stored in memory. This affects, e.g., integer values and pointers while, e.g., arrays of single-byte characters are not affected. The GPC '`String`' schema, however, contains '`Capacity`' and '`Length`' fields before the character array. These fields are integer values larger than one byte, so the '`String`' schema *is* affected by endianness.

Endianness depends on the hardware, especially the CPU. The most common forms are:

- *Little-endian*

  Little-endian machines store the least significant byte on the lowest memory address (the word is stored *little-end-first*).

  E.g., if the 32 bit value `$deadbeef` is stored on memory address `$1234` on a little-endian machine, the following bytes will occupy the memory positions:

Address	**Value**
`$1234`	`$ef`
`$1235`	`$be`
`$1236`	`$ad`
`$1237`	`$de`

  Examples for little-endian machines are IA32 and compatible microprocessors and Alpha processors.

- *Big-endian*

  Big-endian machines store the most significant byte on the lowest memory address (the word is stored *big-end-first*).

  E.g., if the 32 bit value `$deadbeef` is stored on memory address `$1234` on a big-endian machine, the following bytes will occupy the memory positions:

Address	**Value**
`$1234`	`$de`
`$1235`	`$ad`
`$1236`	`$be`
`$1237`	`$ef`

  Examples for big-endian machines are the Sparc and Motorola m68k CPU families and most RISC processors. Big-endian byte order is also used in the Internet protocols.

Note: There are processors which can run in both little-endian and big-endian mode, e.g. the MIPS processors. A single program, however, (unless it uses special machine code instructions) will always run in one endianness.

Under normal circumstances, programs do not need to worry about endianness, the CPU handles it by itself. Endianness becomes important when exchanging data between different machines, e.g. via binary files or over a network. To avoid problems, one has to choose the endianness to use for the data exchange. E.g., the Internet uses big-endian data, and most known data formats have a specified endianness (usually that of the CPU on which the format was originally created). If you define your own binary data format, you're free to choose the endianness to use.

To deal with endianness, GPC predefines the symbol '`__BYTES_LITTLE_ENDIAN__`' on little-endian machines and '`__BYTES_BIG_ENDIAN__`' on big-endian machines. Besides, the Run Time System defines the constant '`BytesBigEndian`' as False on little-endian machines and True on big-endian machines.

There are also the symbols '`__BITS_LITTLE_ENDIAN__`', '`__BITS_BIG_ENDIAN__`', '`__WORDS_LITTLE_ENDIAN__`', '`__WORDS_BIG_ENDIAN__`' and the constants '`BitsBigEndian`' and '`WordsBigEndian`' which concern the order of bits within a byte (e.g., in packed records) or of words within multiword-numbers, but these are usually less important.

The Run Time System also contains a number of routines to convert endianness and to read or write data from/to binary files in a given endianness, independent of the CPU's endianness. These routines are described in the RTS reference (see Section 6.14 [Run Time System], page 105), under '`endianness`'. The demo program '`endiandemo.pas`' contains an example on how to use these routines.

6.2.12.2 Alignment

(Under construction.) `@@` ????

6.3 Operators

GNU Pascal supports all operators of ISO Pascal and Borland Pascal. In addition, you can define your own operators according to the Pascal-SC (PXSC) syntax.

6.3.1 Built-in Operators

The following table lists all built-in GNU Pascal operators, ordered by precedence: '`<`' etc. have the lowest precedence, '`not`' etc. the highest. As usual, the precedence of operators can be superseded with parentheses.

In an assignment statement, '`:=`' has lower precedence than all operators. (This is rather obvious from the syntax of assignment statements, and is merely noted for programmers familiar with C where '`=`' is an operator.)

```
<    =    >    in   <>   >=   <=
+    -    or   +<   -<   +>   ->
*    /    div  mod  and  shl  shr  xor  *<   /<   *>   />
pow  **   ><
not  @
```

The Pascal-SC (PXSC) operators '`+<`', '`-<`', '`+>`', '`->`', '`*<`', '`/<`', '`*>`', and '`/>`' are not yet implemented into GNU Pascal but may be defined by the user (see below).

6.3.2 User-defined Operators

GNU Pascal allows the (re-)definition of binary operators according to the Pascal-SC (PXSC) syntax. The following vector addition example illustrates how to do this:

```
program OperatorDemo;

type
  Vector3 = record
    x, y, z: Real;
  end;

var
  a, b, c: Vector3 = (1, 2, 3);

operator + (u, v: Vector3) w: Vector3;
begin
  w.x := u.x + v.x;
  w.y := u.y + v.y;
  w.z := u.z + v.z;
end;

begin
  c := a + b
end.
```

Between the closing parenthesis of the argument list and the result variable ('`w`' in the above example), GPC allows an optional equal sign. This is not allowed in PXSC, but it is consistent with Extended Pascal's function result variable definitions, where the equal sign is obligatory (but also optional in GPC).

The argument types needn't be equal, and the name of the operator may be an identifier instead of a known symbol. You cannot define *new* symbols in GPC.

The PXSC operators '+>', '+<', etc. for exact numerical calculations currently are not implemented in GPC, but you can define them. Also, the other real-type operators do *not* meet the requirements of PXSC; a module which fixes that would be a welcome contribution.

6.4 Procedure And Function Parameters

6.4.1 Parameters declared as 'protected' or 'const'

All the following works in GPC:

```
procedure Foo (protected a, b, c: Integer);    { 3 arguments }
procedure Foo (a, b, c, protected: Integer);   { 4 arguments }
procedure Foo (a, b, protected, c: Integer);   { 4 arguments }
procedure Foo (protected: Integer);            { 1 argument  }
procedure Foo (var protected: Integer);        { 1 argument  }
procedure Foo (protected protected: Integer);  { 1 argument  }
```

Furthermore, GPC supports `const`, according to BP, which is equivalent to either `protected` or `protected var`, up to the compiler's discretion.

6.4.2 The Standard way to pass arrays of variable size

@@ (Under construction.)

A feature of Standard Pascal level 1.

6.4.3 BP's alternative to Conformant Arrays

Borland Pascal "open array" formal parameters are implemented into GPC. Within the function body, they have integer type index with lower bound 0.

In constrast to conformant arrays (which are not supported by BP), open arrays allow any ordinal type as the index of the actual parameter (which is useful, e.g., if you want to be able to pass values of any enumeration type). However, they lose information about the lower bound (which is a problem, e.g., if you want to return information to the caller that relates to the actual array index, like the function 'IOSelect' in the Run Time System does).

6.5 Accessing parts of strings (and other arrays)

GPC allows the access of parts ("slices") of strings as defined in Extended Pascal. For example:

```
program StringSliceDemo;

const
  HelloWorld = 'Hello, world!';

begin
  WriteLn (HelloWorld[8 .. 12])  { yields 'world' }
end.
```

As an extension, it also allows write access to a string slice:

```
program SliceWriteDemo;

var
  s: String (42) = 'Hello, world!';
```

```
begin
  s[8 .. 12] := 'folks';
  WriteLn (s)  { yields 'Hello, folks!' }
end.
```

As a further extension, GPC allows slice access also to non-string arrays. However, the usefulness of this feature is rather limited because of Pascal's strict type checking rules: If you have, e.g., an '`array [1 .. 10] of Integer`' and take a slice '`[1 .. 5]`' of it, it will not be compatible to another '`array [1 .. 5] of Integer`' because distinct array types are not compatible in Pascal, even if they look the same.

However, array slice access can be used in connection with conformant or open array parameters. See the program '`arrayslicedemo.pas`' (in the '`demos`' directory) for an example.

6.6 Pointer Arithmetics

GPC allows to increment, decrement, compare, and subtract pointers or to use them in '`for`' loops just like the C language.

GPC implements the address operator `@` (a Borland Pascal extension).

```
program PointerArithmeticDemo;
var
  a: array [1 .. 7] of Char;
  p, q: ^Char;
  i: Integer;

{$X+}  { We need extended syntax for pointer arithmetic }

begin
  for p := @a[1] to @a[7] do
    p^ := 'x';

  p := @a[7];
  q := @a[3];
  while p > q do
    begin
      p^ := 'y';
      Dec (p)
    end;

  p := @a[7];
  q := @a[3];
  i := q - p;    { yields 4 }
end.
```

Incrementing a pointer by one means to increment the address it contains by the size of the variable it is pointing to. For typeless pointers ('`Pointer`'), the address is incremented by one instead.

Similar things hold when decrementing a pointer.

Subtracting two pointers yields the number of variables pointed to between both pointers, i.e. the difference of the addresses divided by the size of the variables pointed to. The pointers must be of the same type.

6.7 Type Casts

In some cases, especially in low-level situations, Pascal's strong typing can be an obstacle. To temporarily circumvent this, GPC defines explicit "type casts" in a Borland Pascal compatible way.

There are two kinds of type casts, value type casts and variable type casts.

Value type casts

To convert a value of one data type into another type, you can use the target type like the name of a function that is called. The value to be converted can be a variable or an expression. Both the value's type and the destination type must be ordinal or pointer types. The ordinal value (extended to pointers to mean the address) is preserved in the cast.

An example:

```
program TypeCastDemo;

var
  Ch: Char;
  i: Integer;

begin
  i := Integer (Ch)
end.
```

Another, more complicated, example:

```
program TypeCst2Demo;

type
  CharPtr = ^Char;
  CharArray = array [0 .. 99] of Char;
  CharArrayPtr = ^CharArray;

var
  Foo1, Foo2: CharPtr;
  Bar: CharArrayPtr;

{$X+} { We need extended syntax in order to use ‘‘Succ’’ on a pointer }

begin
  Foo1 := CharPtr (Bar);
  Foo2 := CharPtr (Succ (Bar))
end.
```

However, because of risks involved with type casts, explained below, and because type-casts are non-standard, you should try to avoid type casts whenever possible – and it should be possible in most cases. For instance, the first example above could use the built-in function "Ord" instead of the type cast:

```
i := Ord (Ch);
```

The assignments in the second example could be written in the following way without any type casts:

```
Foo1 := @Bar^[0];
Foo2 := @Bar^[1];
```

Note: In the case of pointers, a warning is issued if the dereferenced target type requires a bigger alignment than the dereferenced source type (see Section 6.2.12.2 [Alignment], page 81).

Variable type casts

It is also possible to temporarily change the type of a variable (more generally, any "lvalue", i.e. something whose address can be taken), without converting its contents in any way. This is called variable type casting.

The syntax is the same as for value type casting. The type-casted variable is still the same variable (memory location) as the original one, just with a different type. Outside of the type cast, the variable keeps its original type.

There are some important differences between value and variable type casting:

- Variable type casting only works on lvalues, not on expressions.
- The result of a variable type casting is still an lvalue, so it can be used, e.g., on the left side of an assignment, or as the operand of an address operator, or passed by reference to a procedure.
- No values are converted in variable type-casting. The contents of the variable, seen as a raw bit pattern, are just interpreted according to the new type.
- Because bit patterns are just interpreted differently, the source and target type must have the same size. If this is not the case, GPC will give a warning.
- Beware: Variable type casts might have unexpected effects on different platforms since you cannot rely on a specific way the data is stored (e.g. see Section 6.2.12.1 [Endianness], page 80).

There are cases where a type-cast could be either a value or a variable cast. This is when both types are ordinal or pointer, and of the same size, and the value is an lvalue. Fortunately, in those cases, the results of both forms are the same, since the same ordinal values (or pointer addresses) are represented by the same bit patterns (when of the same size). Therefore, it doesn't matter which form of type-casting is actually used in these cases.

When dealing with objects (see Section 6.8 [OOP], page 86), it is sometimes necessary to cast a polymorphic pointer to an object into a pointer to a more specialized (derived) object (after checking the actual type). However, the '`as`' operator is a safer approach, so type-casts should be used there only for backward-compatibility (e.g., to BP).

See also: [absolute], page 258, Section 6.2.12.2 [Alignment], page 81, Section 6.2.12.1 [Endianness], page 80, Section 6.8 [OOP], page 86, [Ord], page 377, [Chr], page 290, [Round], page 407, [Trunc], page 435.

6.8 Object-Oriented Programming

GNU Pascal follows the object model of Borland Pascal 7.0. The BP object extensions are almost fully implemented into GPC. This includes inheritance, virtual and non-virtual methods, constructors, destructors, pointer compatibility, extended 'New' syntax (with constructor call and/or as a Boolean function), extended 'Dispose' syntax (with destructor call).

The Borland object model is different from the ISO draft, but it will not be too difficult now to implement that too (plus the Borland Delphi Object Extensions which are quite similar to the ISO draft).

The syntax for an object type declaration is as follows:

```
program ObjectDemo;

type
  Str100 = String (100);

  FooParentPtr = ^FooParent;
  FooPtr = ^Foo;
```

```
  FooParent = object
    constructor Init;
    destructor Done; virtual;
    procedure Bar (c: Real); virtual;
    function Baz (b, a, z: Char) = s: Str100;  { not virtual }
  end;

  Foo = object (FooParent)
    x, y: Integer;
    constructor Init (a, b: Integer);
    destructor Done; virtual;
    procedure Bar (c: Real); virtual;  { overrides `FooParent.Bar' }
    z: Real;  { GPC extension: data fields after methods }
    function Baz: Boolean;  { new function }
  end;

constructor FooParent.Init;
begin
  WriteLn ('FooParent.Init')
end;

destructor FooParent.Done;
begin
  WriteLn ('I''m also done.')
end;

procedure FooParent.Bar (c: Real);
begin
  WriteLn ('FooParent.Bar (', c, ')')
end;

function FooParent.Baz (b, a, z: Char) = s: Str100;
begin
  WriteStr (s, 'FooParent.Baz (', b, ', ', a, ', ', z, ')')
end;

constructor Foo.Init (a, b: Integer);
begin
  inherited Init;
  x := a;
  y := b;
  z := 3.4;
  FooParent.Bar (1.7)
end;

destructor Foo.Done;
begin
  WriteLn ('I''m done.');
  inherited Done
end;
```

```
procedure Foo.Bar (c: Real);
begin
  WriteLn ('Foo.Bar (', c, ')')
end;

function Foo.Baz: Boolean;
begin
  Baz := True
end;

var
  Ptr: FooParentPtr;

begin
  Ptr := New (FooPtr, Init (2, 3));
  Ptr^.Bar (3);
  Dispose (Ptr, Done);
  New (Ptr, Init);
  with Ptr^ do
    WriteLn (Baz ('b', 'a', 'z'))
end.
```

Remarks:

- The ordering of data fields and methods can be mixed.
- GPC supports the '`public`' and '`private`' declarations like BP, and in addition also '`protected`' (scope limited to the current type and its descendants).
- Constructors and destructors are ordinary functions, internally. When a constructor is called, GPC creates some inline code to initialize the object; destructors do nothing special.

A pointer to '`FooParent`' may be assigned the address of a '`Foo`' object. A '`FooParent`' formal '`var`' parameter may get a '`Foo`' object as the actual parameter. In such cases, a call to a '`virtual`' method calls the child's method, whereas a call to a non-'`virtual`' method selects the parent's one:

```
var
  MyFooParent: FooParentPtr;
  SomeFoo: Foo;

[...]

  SomeFoo.Init (4, 2);
  MyFooParent := @SomeFoo;
  MyFooParent^.bar (3.14);  { calls `foo.bar' }
  MyFooParent^.baz ('b', 'a', 'z');  { calls `fooParent.baz' }
  if SomeFoo.baz then  { calls `foo.baz' }
    WriteLn ('Baz!');
```

In a method, an overwritten method of a parent object can be called either prefixing it with the parent type name, or using the keyword '`inherited`':

```
procedure Foo.Bar (c: Real);
begin
  z := c;
  inherited bar (z)  { or: FooParent.Bar (z) }
```

```
end;
```

Use 'FooParent.bar (z)' if you want to be sure that *this* method is called, even if somebody decides not to derive 'foo' directly from 'fooParent' but to have some intermediate object. If you want to call the method 'bar' of the immediate parent – whether it be 'fooParent' or whatever – use 'inherited bar (z)'.

To allocate an object on the heap, use 'New' in one of the following manners:

```
var
  MyFoo: FooPtr;

  [...]

  New (MyFoo, Init (4, 2));

  MyFooParent := New (FooPtr, Init (4, 2))
```

The second possibility has the advantage that 'MyFoo' needn't be a 'FooPtr' but can also be a 'FooParentPtr', i.e. a pointer to an ancestor of 'foo'.

Destructors can and should be called within Dispose:

```
Dispose (MyFooParent, Fini)
```

6.9 Compiler Directives And The Preprocessor

GPC, like UCSD Pascal and BP, treats comments beginning with a '$' immediately following the opening '{' or '(*' as a compiler directive. As in Borland Pascal, {$...} and (*$...*) are equivalent. When a single character plus a '+' or '-' follows, this is also called a compiler switch. All of these directives are case-insensitive (but some of them have case-sensitive arguments). Directives are local and can be changed during one compilation (except include files etc. where this makes no sense).

In general, compiler directives are compiler-dependent. (E.g., only the include directive {$I FileName} is common to UCSD and BP.) Because of BP's popularity, GPC supports all of BP's compiler directives (and ignores those that are unnecessary on its platforms – these are those not listed below), but it knows a lot more directives.

Some BP directives are – of course not by chance – just an alternative notation for C preprocessor directives. But there are differences: BP's *conditional* definitions ('{$define Foo}') go into another name space than the program's definitions. Therefore you can define conditionals and check them via {$ifdef Foo}, but the program will not see them as an identifier 'Foo', so macros do not exist in Borland Pascal.

GPC does support macros, but disables this feature when the '--no-macros' option or the dialect option '--borland-pascal' or '--delphi' is given, to mimic BP's behaviour. Therefore, the following program will react differently when compiled with GPC either without special options or with, e.g., the '--borland-pascal' option (and in the latter case, it behaves the same as when compiled with BP).

```
program MacroDemo;

const Foo = 'Borland Pascal';

{$define Foo 'Default'}

begin
  WriteLn (Foo)
end.
```

Of course, you should not rely on such constructs in your programs. To test if the program is compiled with GPC, you can test the '`__GPC__`' conditional, and to test the dialect used in GPC, you can test the dialect, e.g., with '`{$ifopt borland-pascal}`'.

In general, almost every GPC specific command line option (see Section 5.1 [GPC Command Line Options], page 35) can be turned into a compiler directive (exceptions are those options that contain directory names, such as '`--unit-path`', because they refer to the installation on a particular system, and therefore should be set system-wide, rather than in a source file):

```
--foo       {$foo}
--no-foo    {$no-foo}
-Wbar       {$W bar}     { note the space after the 'W' }
-Wno-bar    {$W no-bar}
```

The following table lists some such examples as well as all those directives that do not correspond to command-line options or have syntactical alternatives (for convenience and/or BP compatibility).

```
--[no-]short-circuit   $B+ $B- like in Borland Pascal:
                               $B- means short-circuit Boolean
                               operators; $B+ complete evaluation

--[no-]io-checking     $I+ $I- like in Borland Pascal:
                               enable/disable I/O checking

--[no-]range-checking  $R+ $R- like in Borland Pascal:
                               enable/disable range checking

--[no-]stack-checking  $S+ $S- like in Borland Pascal:
                               enable/disable stack checking

--[no-]typed-address   $T+ $T- like in Borland Pascal:
                               make the result of the address
                               operator and the Addr function a
                               typed or untyped pointer

-W[no-]warnings        $W+ $W- enable/disable warnings. Note: in
                               '--borland-pascal' mode, the
                               short version is disabled because
                               $W+/$W- has a different meaning in
                               Borland Pascal (which can safely be
                               ignored in GPC), but the long version
                               is still available.

--[no-]extended-syntax $X+ $X- mostly like in Borland Pascal:
                               enable/disable extended syntax
                               (ignore function resuls, operator
                               definitions, 'PChar', pointer
                               arithmetic, ...)

--borland-pascal               disable or warn about GPC features
--extended-pascal              not supported by the standard or
--pascal-sc                    dialect given, do not warn about its
etc.                           ''dangerous'' features (especially BP).
                               The dialect can be changed during one
```

	compilation via directives like, e.g., '{$borland-pascal}'.
{$M Hello!}	write message 'Hello!' to standard error during compilation. In '--borland-pascal' mode, it is ignored it if only numbers follow (for compatibility to Borland Pascal's memory directive)
{$define FOO} or {$CIDefine FOO}	like in Borland Pascal: define *FOO* (for conditional compilation) (case-insensitively)
--cidefine=FOO	the same on the command line
{$CSDefine FOO}	define *FOO* case-sensitively
-D FOO or --csdefine=FOO or --define=FOO	the same on the command line Note: '--define' on the command line is case-sensitive like in GCC, but '{$define}' in the source code is case-insensitive like in BP
{$define loop while True do} or {$CIDefine loop ...}	define 'loop' to be 'while True do' as a macro like in C. The name of the macro is case-insensitive. Note: Macros are disabled in '--borland-pascal' mode because BP doesn't support macros.
--cidefine="loop=..."	the same on the command line
{$CSDefine loop ...}	define a case-sensitive macro
--csdefine="loop=..." or --define="loop=..."	the same on the command line
{$I FileName}	like in Borland Pascal: include 'filename.pas' (the name is converted to lower case)
{$undef FOO}	like in Borland Pascal: undefine FOO
{$ifdef FOO} ... {$else} ... {$endif}	conditional compilation (like in Borland Pascal). Note: GPC predefines the symbol '__GPC__' (with two leading and trailing underscores).

```
{$include "filename.pas"}      include (case-sensitive)

{$include <filename.pas>}      the same, but don't search in the
                               current directory
```

. . . and all the other C preprocessor directives.

You also can use the preprocessor directives in C style, e.g. '`#include`', but this is deprecated because of possible confusion with Borland Pascal style '`#42`' character constants. Besides, in the Pascal style, e.g. '`{$include "foo.bar"}`', there may be more than one directive in the same line.

6.10 Routines Built-in or in the Run Time System

In this section we describe the routines and other declarations that are built into the compiler or part of the Run Time System, sorted by topics.

6.10.1 File Routines

Extended Pascal treats files quite differently from Borland Pascal. GPC supports both forms, even in mixed ways, and provides many extensions.

`@@` A lot missing here

- An example of getting the size of a file (though a '`FileSize`' function is already built-in).

  ```
  function FileSize (FileName : String) : LongInt;
  var
    f: bindable file [0 .. MaxInt] of Char;
    b: BindingType;
  begin
    Unbind (f);
    b := Binding (f);
    b.Name := FileName;
    Bind(f, b);
    b := Binding(f);
    SeekRead (f, 0);
    if Empty (f) then
      FileSize := 0
    else
      FileSize := LastPosition (f) + 1;
    Unbind(f);
  end;
  ```

  Prospero's Extended Pascal has a bug in this case. Replace the MaxInt in the type definition of f by a sufficiently large integer. GNU Pascal works correct in this case.
- GPC implements *lazy* text file I/O, i.e. does a `Put` as soon as possible and a `Get` as late as possible. This should avoid most of the problems sometimes considered to be the most stupid feature of Pascal. When passing a file buffer as parameter the buffer is validated when the parameter is passed.
- GPC supports direct access files. E.g., declaring a type for a file that contains 100 integers.

  ```
  program DirectAccessFileDemo;
  type
    DFile = file [1 .. 100] of Integer;
  var
    F: DFile;
  ```

```
    P, N: 1 .. 100;
  begin
    Rewrite (F);
    P := 42;
    N := 17;
    SeekWrite (F, P);
    Write (F, N)
  end.
```

The following direct access routines may be applied to a direct access file:

`SeekRead (F, N); { Open file in inspection mode, seek to record N }`

`SeekWrite (F, N); { Open file in generation mode, seek to record N }`

`SeekUpdate (F, N); { Open file in update mode, seek to record N }`

`Update (F); { Writes F^, position not changed. F^ kept. }`

`p := Position (F); { Yield the current record number }`

`p := LastPosition (F); { Yield the last record number in file }`

If the file is open for inspection or update, `Get` may be applied. If the file is open for generation or update, `Put` may be applied.

- In BP, you can associate file variables with files using the '`Assign`' procedure which GPC supports.

```
program AssignTextDemo;
var
  t: Text;
  Line: String (4096);
begin
  Assign (t, 'mytext.txt');
  Reset (t);
  while not EOF (t) do
    begin
      ReadLn (t, Line);
      WriteLn (Line)
    end
end.
```

- In Extended Pascal, files are considered entities external to your program. External entities, which don't need to be files, need to be bound to a variable your program. Any variable to which external entities can be bound needs to be declared '`bindable`'. Extended Pascal has the '`Bind`' function that binds a variable to an external entity as well as '`Unbind`' to undo a binding and the function '`Binding`' to get the current binding of a variable.

  GPC supports these routines when applied to files. The compiler will reject binding of other object types.

  Only the fields '`Bound`' and '`Name`' of the predefined record type '`BindingType`' are required by Extended Pascal. Additionally, GPC implements some extensions. For the full definition of '`BindingType`', see [BindingType], page 277.

  The following is an example of binding:

```
program BindingDemo (Input, Output, f);

var
  f: bindable Text;
  b: BindingType;
```

```
procedure BindFile (var f: Text);
var
  b: BindingType;
begin
  Unbind (f);
  b := Binding (f);
  repeat
    Write ('Enter a file name: ');
    ReadLn (b.Name);
    Bind (f, b);
    b := Binding (f);
    if not b.Bound then
      WriteLn ('File not bound -- try again.')
  until b.Bound
end;

begin
  BindFile (f);
  { Now the file f is bound to an external file. We can use the
    implementation defined fields of BindingType to check if the
    file exists and is readable, writable or executable. }
  b := Binding (f);
  Write ('The file ');
  if b.Existing then
    WriteLn ('exists.')
  else
    WriteLn ('does not exist.');
  Write ('It is ');
  if not b.Readable then Write ('not ');
  Write ('readable, ');
  if not b.Writable then Write ('not ');
  Write ('writable and ');
  if not b.Executable then Write ('not ');
  WriteLn ('executable.')
end.
```

Note that Prospero's Pascal defaults to creating the file if it does not exists! You need to use Prospero's local addition of setting `b.Existing` to `True` to work-around this. GPC does not behave like this.

6.10.2 String Operations

In the following description, `s1` and `s2` may be arbitrary string expressions, `s` is a variable of string type.

`WriteStr (s, write-parameter-list)`

`ReadStr (s1, read-parameter-list)`
: Write to a string and read from a string. The parameter lists are identical to '`Write`'/'`Read`' from `Text` files. The semantics is closely modeled after file I/O.

`Index (s1, s2)`
: If `s2` is empty, return 1 else if `s1` is empty return 0 else returns the position of `s2` in `s1` (an integer).

`Length (s1)`
: Return the length of `s1` (an integer from `0 .. s1.Capacity`).

`Trim (s1)`
: Returns a new string with spaces stripped of the end of `s`.

`SubStr (s1, i)`
`SubStr (s1, i, j)`
: Return a new substring of `s1` that contains `j` characters starting from `i`. If `j` is missing, return all the characters starting from `i`.

`EQ (s1, s2)`
`NE (s1, s2)`
`LT (s1, s2)`
`LE (s1, s2)`
`GT (s1, s2)`
`GE (s1, s2)`
: Lexicographic comparisons of `s1` and `s2`. Returns a boolean result. Strings are not padded with spaces.

`s1 = s2`
`s1 <> s2`
`s1 < s2`
`s1 <= s2`
`s1 > s2`
`s1 >= s2`
: Lexicographic comparisons of `s1` and `s2`. Returns a boolean result. The shorter string is blank padded to length of the longer one, but only in '`--extended-pascal`' mode.

GPC supports string catenation with the `+` operator or the '`Concat`' function. All string-types are compatible, so you may catenate any chars, fixed length strings and variable length strings.

```
program ConcatDemo (Input, Output);

var
  Ch  : Char;
  Str : String (100);
  Str2: String (50);
  FStr: packed array [1 .. 20] of Char;

begin
   Ch := '$';
   FStr := 'demo';  { padded with blanks }
   Write ('Give me some chars to play with: ');
   ReadLn (Str);
   Str := '^' + 'prefix:' + Str + ':suffix:' + FStr + Ch;
   WriteLn (Concat ('Le', 'ng', 'th'), ' = ', Length (Str));
   WriteLn (Str)
end.
```

Note: The length of strings in GPC is limited only by the range of '`Integer`' (at least 32 bits, i.e., 2 GB, on most platforms), or the available memory, whichever is smaller).

When trying to write programs portable to other EP compilers, it is however safe to assume a limit of about 32 KB. At least Prospero's Extended Pascal compiler limits strings to 32760 bytes. DEC Pascal limits strings to 65535 bytes.

6.10.3 Accessing Command Line Arguments

GPC supports access to the command line arguments with the BP compatible `ParamStr` and `ParamCount` functions.

- `ParamStr[0]` is the program name,
- `ParamStr[1] .. ParamStr[ParamCount]` are the arguments.

The program below accesses the command line arguments.

```
program CommandLineArgumentsDemo (Output);

var
  Counter: Integer;

begin
  WriteLn ('This program displays command line arguments one per line.');
  for Counter := 0 to ParamCount do
    WriteLn ('Command line argument #', Counter, ' is `',
             ParamStr (Counter), '''')
end.
```

6.10.4 Memory Management Routines

Besides the standard 'New' and 'Dispose' routines, GPC also allows BP style dynamic memory management with `GetMem` and `FreeMem`:

```
GetMem (MyPtr, 1024);
FreeMem (MyPtr, 1024);
```

One somehow strange feature of Borland is **not** supported: You cannot free parts of a variable with `FreeMem`, while the rest is still used and can be freed later by another `FreeMem` call:

```
program PartialFreeMemDemo;

type
  Vector = array [0 .. 1023] of Integer;
  VecPtr = ^Vector;

var
  p, q: VecPtr;

begin
  GetMem (p, 1024 * SizeOf (Integer));
  q := VecPtr (@p^[512]);

  { ... }

  FreeMem (p, 512 * SizeOf (Integer));

  { ... }

  FreeMem (q, 512 * SizeOf (Integer));
end.
```

6.10.5 Operations for Integer and Ordinal Types

- Bit manipulations: The BP style bit shift operators `shl` and `shr` exist in GPC as well as bitwise `and`, `or`, `xor` and `not` for integer values.

  ```
  2#100101 and (1 shl 5) = 2#100000
  ```

  GPC also supports `and`, `or`, `xor` and `not` as procedures:

  ```
  program BitOperatorProcedureDemo;
  var x: Integer;
  begin
    x := 7;
    and (x, 14);  { sets x to 6 }
    xor (x, 3);   { sets x to 5 }
  end.
  ```
- Succ, Pred: The standard functions '`Succ`' and '`Pred`' exist in GPC and accept a second parameter.
- Increment, decrement: The BP built-in Procedures `Inc` and `Dec` exist in GPC.

  ```
  program IncDecDemo;
  var
    i: Integer;
    c: Char;
  begin
    Inc (i);     { i := i + 1; }
    Dec (i, 7);  { i := i - 7; }
    Inc (c, 3);  { c := Succ (c, 3); }
  end.
  ```
- `Min`, `Max`: These are a GNU Pascal extension and work for reals as well as for ordinal types. Mixing reals and integers is okay, the result is real then.

6.10.6 Complex Number Operations

`@@` A lot of details missing here

- binary operators `+`, `-`, `*`, `/` and unary `-`, `+`
- exponentiation operators (`pow` and `**`)
- functions (`Sqr`, `SqRt`, `Exp`, `Ln`, `Sin`, `Cos`, `ArcSin`, `ArcCos`, `ArcTan`)
- number info with `Re`, `Im` and `Arg` functions
- numbers constructed by `Cmplx` or `Polar`

The following sample programs illustrates most of the `Complex` type operations.

```
program ComplexOperationsDemo (Output);

var
  z1, z2: Complex;
  Len, Angle: Real;

begin
  z1 := Cmplx (2, 1);
  WriteLn;
  WriteLn ('Complex number z1 is: (', Re (z1) : 1, ',', Im (z1) : 1, ')');
  WriteLn;
  z2 := Conjugate(z1); { GPC extension }
```

```
    WriteLn ('Conjugate of z1 is: (', Re (z2) : 1, ',', Im (z2) : 1, ')');
    WriteLn;
    Len   := Abs (z1);
    Angle := Arg (z1);
    WriteLn ('The polar representation of z1 is: Length=', Len : 1,
             ', Angle=', Angle : 1);
    WriteLn;
    z2 := Polar (Len, Angle);
    WriteLn ('Converting (Length, Angle) back to (x, y) gives: (',
             Re (z2) : 1, ',', Im (z2) : 1, ')');
    WriteLn;
    WriteLn ('The following operations operate on the complex number z1');
    WriteLn;
    z2 := ArcTan (z1);
    WriteLn ('ArcTan (z1) = (', Re (z2), ', ', Im (z2), ')');
    WriteLn;
    z2 := z1 ** 3.141;
    WriteLn ('z1 ** 3.141 =', Re (z2), ', ', Im (z2), ')');
    WriteLn;
    z2 := Sin (z1);
    WriteLn ('Sin (z1) = (', Re (z2), ', ', Im (z2), ')');
    WriteLn ('(Cos, Ln, Exp, SqRt and Sqr exist also.)');
    WriteLn;
    z2 := z1 pow 8;
    WriteLn ('z1 pow 8 = (', Re (z2), ', ', Im (z2), ')');
    WriteLn;
    z2 := z1 pow (-8);
    WriteLn ('z1 pow (-8) = (', Re (z2), ', ', Im (z2), ')');
  end.
```

6.10.7 Set Operations

GPC supports Standard Pascal set operations. In addition it supports the Extended Pascal set operation symmetric difference (`set1 >< set2`) operation whose result consists of those elements which are in exactly one of the operannds.

It also has a function that counts the elements in the set: '`a := Card (set1)`'.

In the following description, S1 and S2 are variables of set type, s is of the base type of the set.

`S1 := S2` Assign a set to a set variable.

`S1 + S2` Union of sets.

`S1 - S2` Difference between two sets.

`S1 * S2` Intersection of two sets.

`S1 >< S2` Symmetric difference

`S1 = S2` Comparison between two sets. Returns boolean result. `True` if `S1` has the same elements as `S2`.

`S1 <> S2` Comparison between two sets. Returns boolean result. `True` if `S1` does not have the same elements as `S2`.

`S1 < S2`

`S2 > S1` Comparison between two sets. Returns boolean result. `True` if `S1` is a strict subset of `S2`.

`S1 <= S2`

`S2 >= S1` Comparison between two sets. Returns boolean result. `True` if `S1` is a subset of (or equal to) `S2`.

`s in S1` Set membership test between an element `s` and a set. Returns boolean result. `True` if `s` is an element of `S1`.

The following example demonstrates some `set` operations. The results of the operations are given in the comments.

```
program SetOpDemo;

type
  TCharSet = set of Char;

var
  S1, S2, S3: TCharSet;
  Result: Boolean;

begin
  S1 := ['a', 'b', 'c'];
  S2 := ['c', 'd', 'e'];
  S3 := S1 + S2;       { S3 = ['a', 'b', 'c', 'd', 'e'] }
  S3 := S1 * S2;       { S3 = ['c'] }
  S3 := S1 - S2;       { S3 = ['a', 'b'] }
  S3 := S1 >< S2;      { S3 = ['a', 'b', 'd', 'e'] }

  S1 := ['c', 'd', 'e'];
  Result := S1 = S2;   { False }
  Result := S1 < S2;   { False }
  Result := S1 <= S2;  { True }

  S1 := ['c', 'd'];
  Result := S1 <> S2;  { True }
  Result := S2 > S1;   { True }
  Result := S2 >= S1   { True }
end.
```

6.10.8 Date And Time Routines

`procedure GetTimeStamp (var t: TimeStamp);`

`function Date (t: TimeStamp): packed array [1 .. DateLength] of Char;`

`function Time (t: TimeStamp): packed array [1 .. TimeLength] of Char;`

`DateLength` and `TimeLength` are implementation dependent constants.

`GetTimeStamp (t)` fills the record '`t`' with values. If they are valid, the Boolean flags are set to True.

`TimeStamp` is a predefined type in the Extended Pascal standard. It may be extended in an implementation, and is indeed extended in GPC. For the full definition of '`TimeStamp`', see [TimeStamp], page 431.

6.11 Interfacing with Other Languages

The standardized GNU compiler back-end makes it relatively easy to share libraries between GNU Pascal and other GNU compilers. On Unix-like platforms (*not* on Dos-like platforms), the GNU compiler back-end usually complies to the standards defined for that system, so communication with other compilers should be easy, too.

In this chapter we discuss how to import libraries written in other languages, and how to import libraries written in GNU Pascal from other languages. While the examples will specialize to compatibility to GNU C, generalization is straightforward if you are familiar with the other language in question.

6.11.1 Importing Libraries from Other Languages

To use a function written in another language, you need to provide an external declaration for it – either in the program, or in the interface part of a unit, or a module.

Let's say you want to use the following C library from Pascal:

```
File 'callc.c':

#include <unistd.h>
#include "callc.h"

int foo = 1;

void bar (void)
{
  sleep (foo);
}
```

```
File 'callc.h':

/* Actually, we wouldn't need this header file, and could instead
   put these prototypes into callc.c, unless we want to use callc.c
   also from other C source files. */

extern int foo;
extern void bar (void);
```

Then your program can look like this:

```
program CallCDemo;

{$L callc.c}  { Or: 'callc.o' if you don't have the source }

var
  MyFoo: CInteger; external name 'foo';

procedure Bar; external name 'bar';

begin
  MyFoo := 42;
  Bar
end.
```

Or, if you want to provide a '`CallCUnit`' unit:

```
unit CallCUnit;

interface

var
  MyFoo: CInteger; external name 'foo';

procedure Bar; external name 'bar';

implementation

{$L callc.c}  { Or: `callc.o' if you don't have the source }

end.
program CallCUDemo;

uses CallCUnit;

begin
  MyFoo := 42;
  Bar
end.
```

You can either link your program manually with 'callc.o' or put a compiler directive '{$L callc.o}' into your program or unit, and then GPC takes care of correct linking. If you have the source of the C library (you always have it if it is Free Software), you can even write '{$L callc.c}' in the program (like above). Then GPC will also link with 'callc.o', but in addition GPC will run the C compiler whenever 'callc.c' has changed if '--automake' is given, too.

While it is often convenient, there is no must to give the C function 'bar' the name 'Bar' in Pascal; you can name it as you like (e.g., the variable 'MyFoo' has a C name of 'foo' in the example above).

If you omit the 'name', the default is the Pascal identifier, converted to lower-case. So, in this example, the 'name' could be omitted for 'Bar', but not for 'MyFoo'.

It is important that data types of both languages are mapped correctly onto each other. C's 'int', for instance, translates to GPC's 'CInteger', and C's 'unsigned long' to 'MedCard'. For a complete list of integer types with their C counterparts, see Section 6.2.3 [Integer Types], page 64.

In some cases it can be reasonable to translate a C pointer parameter to a Pascal 'var' parameter. Since const parameters in GPC can be passed by value *or* by reference internally, possibly depending on the system, 'const foo *' parameters to C functions *cannot* reliably be declared as 'const' in Pascal. However, Extended Pascal's 'protected var' can be used since this guarantees passing by reference.

Some libraries provide a 'main' function and require your program's "main" to be named differently. To achive this with GPC, invoke it with an option '--gpc-main="GPCmain"' (where 'GPCmain' is an example how you might want to name the program). You can also write it into your source as a directive '{$gpc-main="GPCmain"}'.

6.11.2 Exporting GPC Libraries to Other Languages

The '.o' files produced by GPC are in the same format as those of all other GNU compilers, so there is no problem in writing libraries for other languages in Pascal. To use them, you will need to write kind of interface – a header file in C. However there are some things to take into account, especially if your Pascal unit exports objects:

- By default, GPC capitalizes the first letter (only) of each identifier, so '`procedure FooBAR`' must be imported as '`extern void Foobar()`' from C.
- If you want to specify the external name explicitly, use '`attribute`':

```
procedure FooBAR; attribute (name = 'FooBAR');
begin
  WriteLn ('FooBAR')
end;
```

  This one can be imported from C with '`extern void FooBar()`'.
- Objects are "records" internally. They have an implicit '`vmt`' field which contains a pointer to the "virtual method table". This table is another record of the following structure:

```
type
  VMT = record
    ObjectSize: PtrInt;     { Size of object in bytes }
    NegObjectSize: PtrInt;  { Negated size }
    Methods: array [1 .. n] of procedure;
      { Pointers to the virtual methods. The entries are of the
         repective procedure or function types. }
  end;
```

  You can call a virtual method of an object from C if you explicitly declare this '`struct`' and explicitly dereference the '`Fun`' array. The VMT of an object '`FooBAR`' is an external (in C sense) variable '`vmt_Foobar`' internally.
- Methods of objects are named '`Myobject_Mymethod`' (with exactly two capital letters) internally.
- If you want to put a program in a library for some reason, and you want to give the '`main`' program an internal name different from '`main`', call GPC with the command-line option '`--gpc-main="GPCmain"`' (see the previous subsection).

6.12 Notes for Debugging

- The GNU debugger, '`gdb`', does not yet understand Pascal sets, files or subranges. Now '`gdb`' allows you to debug these things, even though it does not yet understand some stabs.
- Forward referencing pointers generate debug info that appears as generic pointers.
- No information of '`with`' statements is currently given to the debugger.
- When debugging, please note that the Initial Letter In Each Identifier Is In Upper Case And The Rest Are In Lower Case, unless explicitly overriden with '`name`' (see [name], page 366). This is to reduce name clashes with libc and other possible libraries.
- All visible GPC Run Time System routines have linker names starting with '`_p_`'.
- The linker name of the main program is '`pascal_main_program`'. This is done because ISO Standard wants to have the program name in a separate name space.

6.13 How to use I18N in own programs

This chapter discusses shortly how to use the Internationalization (I18N) features of GNU Pascal.

Prerequisite

You need to have gettext installed. Try to compile '`demos/gettextdemo.pas`'. Furthermore, you should download a tool named '`pas2po`' from `http://www.gnu-pascal.org/contrib/eike/`.

The source

We would like to translate the messages provided with this simple example different languages (here: German) without touching the source for each language:

```
program Hello1;

begin
  WriteLn ('Hello, World!');
  WriteLn ('The answer of the questions is: ', 42)
end.
```

Preparing the source

To do so, we must prepare the source to use gettext:

```
program Hello2;

uses GPC, Intl;

var s: TString;

begin
  Discard (BindTextDomain ('hello2', '/usr/share/locale/'));
  Discard (TextDomain ('hello2'));
  WriteLn (GetText ('Hello, World!'));
  s := FormatString (GetText ('The answer of the questions is %s'), 42);
  WriteLn (s)
end.
```

'BindTextDomain' sets the path to find our message catalogs in the system. This path is system dependent. 'TextDomain' tells the program to use this catalog. 'GetText' looks up the given string in the catalog and returns a translated string within the current locale settings. 'FormatString' replaces some format specifiers with the following argument. '%s' is the first following argument. After this step is done, we do not need to touch the sourcefile any longer. The output of this program is as follows:

```
Hello, World!
The answer of the questions is 42
```

Getting the translatable strings

There are lots of strings in the above example, but only those surrounded with 'GetText' should be translated. We use 'pas2po hello2.pas -o hello2.po' to extract the messages. The output is:

```
# This file was created by pas2po with 'hello2.pas'.
# Please change this file manually.
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
```

```
"POT-Creation-Date: 2003-04-27 20:48+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#hello2.pas:10
msgid "Hello, World!"
msgstr ""

#hello2.pas:11
msgid "The answer of the questions is %s"
msgstr ""
```

Now we translate the message ids into German language, and set some needful informations at their appropriate places. The following steps must be repeated for each language we would like to support:

```
# This file was created by pas2po with 'hello2.pas'.
# Copyright (C) 2003 Free Software Foundation, Inc.
# Eike Lange  <eike@g-n-u.de>, 2003.
msgid ""
msgstr ""
"Project-Id-Version: Hello2 1.0\n"
"POT-Creation-Date: 2003-04-27 12:00+0200\n"
"PO-Revision-Date: 2003-04-27 12:06+0200\n"
"Last-Translator: Eike Lange <eike@g-n-u.de>\n"
"Language-Team: de <de@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ISO-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"

#hello2.pas:10
msgid "Hello, World!"
msgstr "Hallo, Welt!"

#hello2.pas:11
msgid "The answer of the questions is %s"
msgstr "'%s' lautet die Antwort auf die Frage."
```

Please note that we swapped text and numeric arguments and added some single quotes arround the first argument. We compile the message catalog with '`msgfmt -vv hello2.po -o hello2.mo`' and install the file '`hello2.mo`' at '`/usr/share/locale/de/LC_MESSAGES/`' With a german locale setting, the output should be as follows:

```
Hallo, Welt!
'42' lautet die Antwort auf die Frage.
```

System dependent notes:

The topmost path where message catalogs reside is system dependent:

for DJGPP:
: '`GetEnv ('$DJDIR') + '/share/locale'`'

for Mac OS X:
    '/usr/share/locale' or '/sw/share/locale'

for Linux, *BSD:
    '/usr/share/locale' or '/usr/local/share/locale'

See also

⟨undefined⟩ [Gettext], page ⟨undefined⟩, [FormatString], page 326, Section 6.15.8 [Intl], page 195.

6.14 Pascal declarations for GPC's Run Time System

Below is a Pascal source of the declarations in GPC's Run Time System (RTS). A file 'gpc.pas' with the same contents is included in the GPC distribution in a 'units' subdirectory of the directory containing 'libgcc.a'. (To find out the correct directory for your installation, type 'gpc --print-file-name=units' on the command line.)

```
{ This file was generated automatically by make-gpc-pas.
  DO NOT CHANGE THIS FILE MANUALLY! }

{ Pascal declarations of the GPC Run Time System that are visible to
  each program.

  This unit contains Pascal declarations of many RTS routines which
  are not built into the compiler and can be called from programs.
  Don't copy the declarations from this unit into your programs, but
  rather include this unit with a 'uses' statement. The reason is
  that the internal declarations, e.g. the linker names, may change,
  and this unit will be changed accordingly. @@In the future, this
  unit might be included into every program automatically, so there
  will be no need for a 'uses' statement to make the declarations
  here available.

  Note about 'protected var' parameters:
  Since 'const' parameters in GPC may be passed by value *or* by
  reference internally, possibly depending on the system,
  'const foo *' parameters to C functions *cannot* reliably be
  declared as 'const' in Pascal. However, Extended Pascal's
  'protected var' can be used since this guarantees passing by
  reference.

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Authors: Jukka Virtanen <jtv@hut.fi>
           Peter Gerwinski <peter@gerwinski.de>
           Frank Heckenbach <frank@pascal.gnu.de>
           J.J. v.der Heijden <j.j.vanderheijden@student.utwente.nl>
           Nicola Girardi <nicola@g-n-u.de>
           Prof. Abimbola A. Olowofoyeku <African_Chief@bigfoot.com>
           Emil Jerabek <jerabek@math.cas.cz>
           Maurice Lombardi <Maurice.Lombardi@ujf-grenoble.fr>
           Toby Ewing <ewing@iastate.edu>
```

```
         Mirsad Todorovac <mtodorov_69@yahoo.com>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ <> 20050331}
{$error
Trying to compile gpc.pas with a non-matching GPC version is likely
to cause problems.

In case you are building the RTS separately from GPC, make sure you
install a current GPC version previously. If you are building GPC
now and this message appears, something is wrong -- if you are
overriding the GCC_FOR_TARGET or GPC_FOR_TARGET make variables, this
might be the problem. If you are cross-building GPC, build and
install a current GPC cross-compiler first, sorry. If that's not the
case, please report it as a bug.

If you are not building GPC or the RTS currently, you might have
installed things in the wrong place, so the compiler and RTS
versions do not match.}
{$endif}

{ Command-line options must not change the layout of RTS types
  declared here. }
{$no-pack-struct, maximum-field-alignment 0}

module GPC;
```

```
export
  GPC = all;
  GPC_CP = (ERead { @@ not really, but an empty export doesn't work
  } );
  GPC_EP = (ERead { @@ not really, but an empty export doesn't work
  } );
  GPC_BP = (MaxLongInt, ExitCode, ErrorAddr, FileMode, Pos);
  GPC_Delphi = (MaxLongInt, Int64, InitProc, EConvertError,
                ExitCode, ErrorAddr, FileMode, Pos, SetString,
  StringOfChar,
                TextFile, AssignFile, CloseFile);

type
  GPC_FDR = AnyFile;

{ Pascal declarations of the GPC Run Time System routines that are
  implemented in C, from rtsc.pas }

const
  { Maximum size of a variable }
  MaxVarSize = MaxInt div 8;

{ If set, characters >= #$80 are assumed to be letters even if the
  locale routines don't say so. This is a kludge because some
  systems don't have correct non-English locale tables. }
var
  FakeHighLetters: Boolean; attribute (name = '_p_FakeHighLetters');
  external;

type
  PCStrings = ^TCStrings;
  TCStrings = array [0 .. MaxVarSize div SizeOf (CString) - 1] of
  CString;

  Int64 = Integer attribute (Size = 64);
  UnixTimeType = LongInt;  { This is hard-coded in the compiler. Do
  not change here. }
  MicroSecondTimeType = LongInt;
  FileSizeType = LongInt;
  SignedSizeType = Integer attribute (Size = BitSizeOf (SizeType));
  TSignalHandler = procedure (Signal: CInteger);

  StatFSBuffer = record
    BlockSize, BlocksTotal, BlocksFree: LongInt;
    FilesTotal, FilesFree: CInteger
  end;

  InternalSelectType = record
    Handle: CInteger;
    Read, Write, Exception: Boolean
  end;
```

```
  PString = ^String;

  { ‘Max’ so the range of the array does not become invalid for
    Count = 0 }
  PPStrings = ^TPStrings;
  TPStrings (Count: Cardinal) = array [1 .. Max (Count, 1)] of
  PString;

  GlobBuffer = record
    Result: PPStrings;
    Internal1: Pointer;
    Internal2: PCStrings;
    Internal3: CInteger
  end;

{ Mathematical routines }

function  SinH (x: Real): Real; attribute (const); external
  name '_p_SinH';
function  CosH (x: Real): Real; attribute (const); external
  name '_p_CosH';
function  ArcTan2 (y: Real; x: Real): Real; attribute (const);
  external name '_p_ArcTan2';
function  IsInfinity (x: LongReal): Boolean; attribute (const);
  external name '_p_IsInfinity';
function  IsNotANumber (x: LongReal): Boolean; attribute (const);
  external name '_p_IsNotANumber';
procedure SplitReal (x: LongReal; var Exponent: CInteger; var
  Mantissa: LongReal); external name '_p_SplitReal';

{ Character routines }

{ Convert a character to lower case, according to the current
  locale. }
function  LoCase (ch: Char): Char; attribute (const); external
  name '_p_LoCase';
function  IsUpCase (ch: Char): Boolean; attribute (const); external
  name '_p_IsUpCase';
function  IsLoCase (ch: Char): Boolean; attribute (const); external
  name '_p_IsLoCase';
function  IsAlpha (ch: Char): Boolean; attribute (const); external
  name '_p_IsAlpha';
function  IsAlphaNum (ch: Char): Boolean; attribute (const);
  external name '_p_IsAlphaNum';
function  IsAlphaNumUnderscore (ch: Char): Boolean; attribute
  (const); external name '_p_IsAlphaNumUnderscore';
function  IsSpace (ch: Char): Boolean; attribute (const); external
  name '_p_IsSpace';
function  IsPrintable (ch: Char): Boolean; attribute (const);
  external name '_p_IsPrintable';
```

```
{ Time routines }

{ Sleep for a given number of seconds. }
procedure Sleep (Seconds: CInteger); external name '_p_Sleep';

{ Sleep for a given number of microseconds. }
procedure SleepMicroSeconds (MicroSeconds: CInteger); external
  name '_p_SleepMicroSeconds';

{ Set an alarm timer. }
function  Alarm (Seconds: CInteger): CInteger; external
  name '_p_Alarm';

{ Convert a Unix time value to broken-down local time.
  All parameters except Time may be Null. }
procedure UnixTimeToTime (Time: UnixTimeType; var Year: CInteger;
  var Month: CInteger; var Day: CInteger; var Hour: CInteger; var
  Minute: CInteger; var Second: CInteger;
                              var TimeZone: CInteger; var DST:
  Boolean; var TZName1: CString; var TZName2: CString); external
  name '_p_UnixTimeToTime';

{ Convert broken-down local time to a Unix time value. }
function  TimeToUnixTime (Year: CInteger; Month: CInteger; Day:
  CInteger; Hour: CInteger; Minute: CInteger; Second: CInteger):
  UnixTimeType; external name '_p_TimeToUnixTime';

{ Get the real time. MicroSecond can be Null and is ignored then. }
function  GetUnixTime (var MicroSecond: CInteger): UnixTimeType;
  external name '_p_GetUnixTime';

{ Get the CPU time used. MicroSecond can be Null and is ignored
  then. }
function  GetCPUTime (var MicroSecond: CInteger): CInteger; external
  name '_p_GetCPUTime';

{ Signal and process routines }

{ Extract information from the status returned by PWait }
function  StatusExited (Status: CInteger): Boolean; attribute
  (const); external name '_p_StatusExited';
function  StatusExitCode (Status: CInteger): CInteger; attribute
  (const); external name '_p_StatusExitCode';
function  StatusSignaled (Status: CInteger): Boolean; attribute
  (const); external name '_p_StatusSignaled';
function  StatusTermSignal (Status: CInteger): CInteger; attribute
  (const); external name '_p_StatusTermSignal';
function  StatusStopped (Status: CInteger): Boolean; attribute
  (const); external name '_p_StatusStopped';
function  StatusStopSignal (Status: CInteger): CInteger; attribute
```

```
  (const); external name '_p_StatusStopSignal';

{ Install a signal handler and optionally return the previous
  handler. OldHandler and OldRestart may be Null. }
function  InstallSignalHandler (Signal: CInteger; Handler:
  TSignalHandler; Restart: Boolean; UnlessIgnored: Boolean;
  var OldHandler: TSignalHandler; var OldRestart: Boolean): Boolean;
  external name '_p_InstallSignalHandler';

{ Block or unblock a signal. }
procedure BlockSignal (Signal: CInteger; Block: Boolean); external
  name '_p_BlockSignal';

{ Test whether a signal is blocked. }
function  SignalBlocked (Signal: CInteger): Boolean; external
  name '_p_SignalBlocked';

{ Sends a signal to a process. Returns True if successful. If Signal
  is 0, it doesn't send a signal, but still checks whether it would
  be possible to send a signal to the given process. }
function  Kill (PID: CInteger; Signal: CInteger): Boolean; external
  name '_p_Kill';

{ Constant for WaitPID }
const
  AnyChild = -1;

{ Waits for a child process with the given PID (or any child process
  if PID = AnyChild) to terminate or be stopped. Returns the PID of
  the process. WStatus will contain the status and can be evaluated
  with StatusExited etc.. If nothing happened, and Block is False,
  the function will return 0, and WStatus will be 0. If an error
  occurred (especially on single tasking systems where WaitPID is
  not possible), the function will return a negative value, and
  WStatus will be 0. }
function  WaitPID (PID: CInteger; var WStatus: CInteger; Block:
  Boolean): CInteger; external name '_p_WaitPID';

{ Returns the process ID. }
function  ProcessID: CInteger; external name '_p_ProcessID';

{ Returns the process group. }
function  ProcessGroup: CInteger; external name '_p_ProcessGroup';

{ Returns the real or effective user ID of the process. }
function  UserID (Effective: Boolean): CInteger; external
  name '_p_UserID';

{ Tries to change the real and/or effective user ID. }
function  SetUserID (Real: CInteger; Effective: CInteger): Boolean;
  external name '_p_SetUserID';
```

```
{ Returns the real or effective group ID of the process. }
function  GroupID (Effective: Boolean): CInteger; external
  name '_p_GroupID';

{ Tries to change the real and/or effective group ID. }
function  SetGroupID (Real: CInteger; Effective: CInteger): Boolean;
  external name '_p_SetGroupID';

{ Low-level file routines. Mostly for internal use. }

{ Get information about a file system. }
function  StatFS (Path: CString; var Buf: StatFSBuffer): Boolean;
  external name '_p_StatFS';
function  CStringOpenDir (DirName: CString): Pointer; external
  name '_p_CStringOpenDir';
function  CStringReadDir (Dir: Pointer): CString; external
  name '_p_CStringReadDir';
procedure CStringCloseDir (Dir: Pointer); external
  name '_p_CStringCloseDir';

{ Returns the value of the symlink FileName in a CString allocated
  from the heap. Returns nil if it is no symlink or the function
  is not supported. }
function  ReadLink (FileName: CString): CString; external
  name '_p_ReadLink';

{ Returns a pointer to a *static* buffer! }
function  CStringRealPath (Path: CString): CString; external
  name '_p_CStringRealPath';

{ File mode constants that are ORed for BindingType.Mode, ChMod,
  CStringChMod and Stat. The values below are valid for all OSs
  (as far as supported). If the OS uses different values, they're
  converted internally. }
const
  fm_SetUID           = 8#4000;
  fm_SetGID           = 8#2000;
  fm_Sticky           = 8#1000;
  fm_UserReadable     = 8#400;
  fm_UserWritable     = 8#200;
  fm_UserExecutable   = 8#100;
  fm_GroupReadable    = 8#40;
  fm_GroupWritable    = 8#20;
  fm_GroupExecutable  = 8#10;
  fm_OthersReadable   = 8#4;
  fm_OthersWritable   = 8#2;
  fm_OthersExecutable = 8#1;

{ Constants for Access and OpenHandle }
const
```

```
  MODE_EXEC     = 1 shl 0;
  MODE_WRITE    = 1 shl 1;
  MODE_READ     = 1 shl 2;
  MODE_FILE     = 1 shl 3;
  MODE_CREATE   = 1 shl 4;
  MODE_EXCL     = 1 shl 5;
  MODE_TRUNCATE = 1 shl 6;
  MODE_APPEND   = 1 shl 7;
  MODE_BINARY   = 1 shl 8;

{ Check if a file name is accessible. }
function  Access (FileName: CString; Request: CInteger): CInteger;
  external name '_p_Access';

{ Get information about a file. Any argument except FileName can
  be Null. }
function  Stat (FileName: CString; var Size: FileSizeType;
  var ATime: UnixTimeType; var MTime: UnixTimeType; var CTime:
  UnixTimeType;
  var User: CInteger; var Group: CInteger; var Mode: CInteger; var
  Device: CInteger; var INode: CInteger; var Links: CInteger;
  var SymLink: Boolean; var Dir: Boolean; var Special: Boolean):
  CInteger; external name '_p_Stat';
function  OpenHandle (FileName: CString; Mode: CInteger): CInteger;
  external name '_p_OpenHandle';
function  ReadHandle (Handle: CInteger; Buffer: Pointer; Size:
  SizeType): SignedSizeType; external name '_p_ReadHandle';
function  WriteHandle (Handle: CInteger; Buffer: Pointer; Size:
  SizeType): SignedSizeType; external name '_p_WriteHandle';
function  CloseHandle (Handle: CInteger): CInteger; external
  name '_p_CloseHandle';
procedure FlushHandle (Handle: CInteger); external
  name '_p_FlushHandle';
function  DupHandle (Src: CInteger; Dest: CInteger): CInteger;
  external name '_p_DupHandle';
function  SetFileMode (Handle: CInteger; Mode: CInteger; On:
  Boolean): CInteger; attribute (ignorable); external
  name '_p_SetFileMode';
function  CStringRename (OldName: CString; NewName: CString):
  CInteger; external name '_p_CStringRename';
function  CStringUnlink (FileName: CString): CInteger; external
  name '_p_CStringUnlink';
function  CStringChDir (FileName: CString): CInteger; external
  name '_p_CStringChDir';
function  CStringMkDir (FileName: CString): CInteger; external
  name '_p_CStringMkDir';
function  CStringRmDir (FileName: CString): CInteger; external
  name '_p_CStringRmDir';
function  UMask (Mask: CInteger): CInteger; attribute (ignorable);
  external name '_p_UMask';
function  CStringChMod (FileName: CString; Mode: CInteger):
```

```
  CInteger; external name '_p_CStringChMod';
function  CStringChOwn (FileName: CString; Owner: CInteger; Group:
  CInteger): CInteger; external name '_p_CStringChOwn';
function  CStringUTime (FileName: CString; AccessTime: UnixTimeType;
  ModificationTime: UnixTimeType): CInteger; external
  name '_p_CStringUTime';

{ Constants for SeekHandle }
const
  SeekAbsolute = 0;
  SeekRelative = 1;
  SeekFileEnd  = 2;

{ Seek to a position on a file handle. }
function  SeekHandle (Handle: CInteger; Offset: FileSizeType;
  Whence: CInteger): FileSizeType; external name '_p_SeekHandle';
function  TruncateHandle (Handle: CInteger; Size: FileSizeType):
  CInteger; external name '_p_TruncateHandle';
function  LockHandle (Handle: CInteger; WriteLock: Boolean; Block:
  Boolean): Boolean; external name '_p_LockHandle';
function  UnlockHandle (Handle: CInteger): Boolean; external
  name '_p_UnlockHandle';
function  SelectHandle (Count: CInteger; var Events:
  InternalSelectType; MicroSeconds: MicroSecondTimeType): CInteger;
  external name '_p_SelectHandle';

{ Constants for MMapHandle and MemoryMap }
const
  mm_Readable   = 1;
  mm_Writable   = 2;
  mm_Executable = 4;

{ Try to map (a part of) a file to memory. }
function  MMapHandle (Start: Pointer; Length: SizeType; Access:
  CInteger; Shared: Boolean; Handle: CInteger; Offset:
  FileSizeType): Pointer; external name '_p_MMapHandle';

{ Unmap a previous memory mapping. }
function  MUnMapHandle (Start: Pointer; Length: SizeType): CInteger;
  external name '_p_MUnMapHandle';

{ Returns the file name of the terminal device that is open on
  Handle. Returns nil if (and only if) Handle is not open or not
  connected to a terminal. If NeedName is False, it doesn't bother
  to search for the real name and just returns DefaultName if it
  is a terminal and nil otherwise. DefaultName is also returned if
  NeedName is True, Handle is connected to a terminal, but the
  system does not provide information about the real file name. }
function  GetTerminalNameHandle (Handle: CInteger; NeedName:
  Boolean; DefaultName: CString): CString; external
  name '_p_GetTerminalNameHandle';
```

```
{ System routines }

{ Sets the process group of Process (or the current one if Process
  is 0) to ProcessGroup (or its PID if ProcessGroup is 0). Returns
  True if successful. }
function  SetProcessGroup (Process: CInteger; ProcessGroup:
  CInteger): Boolean; external name '_p_SetProcessGroup';

{ Sets the process group of a terminal given by Terminal (as a file
  handle) to ProcessGroup. ProcessGroup must be the ID of a process
  group in the same session. Returns True if successful. }
function  SetTerminalProcessGroup (Handle: CInteger; ProcessGroup:
  CInteger): Boolean; external name '_p_SetTerminalProcessGroup';

{ Returns the process group of a terminal given by Terminal (as a
  file handle), or -1 on error. }
function  GetTerminalProcessGroup (Handle: CInteger): CInteger;
  external name '_p_GetTerminalProcessGroup';

{ Set the standard input's signal generation, if it is a terminal. }
procedure SetInputSignals (Signals: Boolean); external
  name '_p_SetInputSignals';

{ Get the standard input's signal generation, if it is a terminal. }
function  GetInputSignals: Boolean; external
  name '_p_GetInputSignals';

{ Internal routines }

{ Returns system information if available. Fields not available will
  be set to nil. }
procedure CStringSystemInfo (var SysName: CString; var NodeName:
  CString; var Release: CString; var Version: CString; var Machine:
  CString; var DomainName: CString); external
  name '_p_CStringSystemInfo';

{ Returns the path of the running executable *if possible*. }
function  CStringExecutablePath (Buffer: CString): CString; external
  name '_p_CStringExecutablePath';

{ Sets ErrNo to the value of `errno' and returns the description
  for this error. May return nil if not supported! ErrNo may be
  Null (then only the description is returned). }
function  CStringStrError (var ErrNo: CInteger): CString; external
  name '_p_CStringStrError';

{ Mathematical routines, from math.pas }

function Ln1Plus  (x: Real) = y: Real; attribute (const, name
  = '_p_Ln1Plus'); external;
```

```
{ String handling routines (lower level), from string1.pas }

{ TString is a string type that is used for function results and
  local variables, as long as undiscriminated strings are not
  allowed there. The default size of 2048 characters should be
  enough for file names on any system, but can be changed when
  necessary. It should be at least as big as MAXPATHLEN. }

const
  MaxLongInt = High (LongInt);

  TStringSize = 2048;
  SpaceCharacters = [' ', #9];
  NewLine = "\n";  { the separator of lines within a string }
  LineBreak = {$if defined (__OS_DOS__) and not defined (__CYGWIN__)
  and not defined (__MSYS__)}
              "\r\n"
              {$else}
              "\n"
              {$endif};  { the separator of lines within a file }

type
  TString    = String (TStringSize);
  TStringBuf = packed array [0 .. TStringSize] of Char;
  CharSet    = set of Char;
  Str64      = String (64);
  TInteger2StringBase = 2 .. 36;
  TInteger2StringWidth = 0 .. High (TString);

var
  NumericBaseDigits: array [0 .. 35] of Char; attribute (const, name
  = '_p_NumericBaseDigits'); external;
  NumericBaseDigitsUpper: array [0 .. 35] of Char; attribute (const,
  name = '_p_NumericBaseDigitsUpper'); external;

  CParamCount: Integer; attribute (name = '_p_CParamCount');
  external;
  CParameters: PCStrings; attribute (name = '_p_CParameters');
  external;

function  MemCmp      (const s1, s2; Size: SizeType): CInteger;
  external name 'memcmp';
function  MemComp     (const s1, s2; Size: SizeType): CInteger;
  external name 'memcmp';
function  MemCompCase (const s1, s2; Size: SizeType): Boolean;
  attribute (name = '_p_MemCompCase'); external;

procedure UpCaseString    (var s: String); attribute (name
  = '_p_UpCaseString'); external;
procedure LoCaseString    (var s: String); attribute (name
```

```
  = '_p_LoCaseString'); external;
function  UpCaseStr       (const s: String) = Result: TString;
  attribute (name = '_p_UpCaseStr'); external;
function  LoCaseStr       (const s: String) = Result: TString;
  attribute (name = '_p_LoCaseStr'); external;

function  StrEqualCase    (const s1, s2: String): Boolean; attribute
  (name = '_p_StrEqualCase'); external;

function  Pos             (const SubString, s: String): Integer;
  attribute (name = '_p_Pos'); external;
function  PosChar         (const ch: Char; const s: String):
  Integer; attribute (name = '_p_PosChar'); external;
function  LastPos         (const SubString, s: String): Integer;
  attribute (name = '_p_LastPos'); external;
function  PosCase         (const SubString, s: String): Integer;
  attribute (name = '_p_PosCase'); external;
function  LastPosCase     (const SubString, s: String): Integer;
  attribute (name = '_p_LastPosCase'); external;
function  CharPos         (const Chars: CharSet; const s: String):
  Integer; attribute (name = '_p_CharPos'); external;
function  LastCharPos     (const Chars: CharSet; const s: String):
  Integer; attribute (name = '_p_LastCharPos'); external;

function  PosFrom         (const SubString, s: String; From:
  Integer): Integer; attribute (name = '_p_PosFrom'); external;
function  LastPosTill     (const SubString, s: String; Till:
  Integer): Integer; attribute (name = '_p_LastPosTill'); external;
function  PosFromCase     (const SubString, s: String; From:
  Integer): Integer; attribute (name = '_p_PosFromCase'); external;
function  LastPosTillCase (const SubString, s: String; Till:
  Integer): Integer; attribute (name = '_p_LastPosTillCase');
  external;
function  CharPosFrom     (const Chars: CharSet; const s: String;
  From: Integer): Integer; attribute (name = '_p_CharPosFrom');
  external;
function  LastCharPosTill (const Chars: CharSet; const s: String;
  Till: Integer): Integer; attribute (name = '_p_LastCharPosTill');
  external;

function  IsPrefix        (const Prefix, s: String): Boolean;
  attribute (name = '_p_IsPrefix'); external;
function  IsSuffix        (const Suffix, s: String): Boolean;
  attribute (name = '_p_IsSuffix'); external;
function  IsPrefixCase    (const Prefix, s: String): Boolean;
  attribute (name = '_p_IsPrefixCase'); external;
function  IsSuffixCase    (const Suffix, s: String): Boolean;
  attribute (name = '_p_IsSuffixCase'); external;

function  CStringLength      (Src: CString): SizeType; attribute
  (inline, name = '_p_CStringLength'); external;
```

```
function  CStringEnd        (Src: CString): CString; attribute
  (inline, name = '_p_CStringEnd'); external;
function  CStringNew        (Src: CString): CString; attribute
  (name = '_p_CStringNew'); external;
function  CStringComp       (s1, s2: CString): Integer; attribute
  (name = '_p_CStringComp'); external;
function  CStringCaseComp   (s1, s2: CString): Integer; attribute
  (name = '_p_CStringCaseComp'); external;
function  CStringLComp      (s1, s2: CString; MaxLen: SizeType):
  Integer; attribute (name = '_p_CStringLComp'); external;
function  CStringLCaseComp  (s1, s2: CString; MaxLen: SizeType):
  Integer; attribute (name = '_p_CStringLCaseComp'); external;
function  CStringCopy       (Dest, Source: CString): CString;
  attribute (ignorable, name = '_p_CStringCopy'); external;
function  CStringCopyEnd    (Dest, Source: CString): CString;
  attribute (ignorable, name = '_p_CStringCopyEnd'); external;
function  CStringLCopy      (Dest, Source: CString; MaxLen:
  SizeType): CString; attribute (ignorable, name
  = '_p_CStringLCopy'); external;
function  CStringMove       (Dest, Source: CString; Count:
  SizeType): CString; attribute (ignorable, name
  = '_p_CStringMove'); external;
function  CStringCat        (Dest, Source: CString): CString;
  attribute (ignorable, name = '_p_CStringCat'); external;
function  CStringLCat       (Dest, Source: CString; MaxLen:
  SizeType): CString; attribute (ignorable, name
  = '_p_CStringLCat'); external;
function  CStringChPos      (Src: CString; ch: Char): CString;
  attribute (inline, name = '_p_CStringChPos'); external;
function  CStringLastChPos  (Src: CString; ch: Char): CString;
  attribute (inline, name = '_p_CStringLastChPos'); external;
function  CStringPos        (s, SubString: CString): CString;
  attribute (name = '_p_CStringPos'); external;
function  CStringLastPos    (s, SubString: CString): CString;
  attribute (name = '_p_CStringLastPos'); external;
function  CStringCasePos    (s, SubString: CString): CString;
  attribute (name = '_p_CStringCasePos'); external;
function  CStringLastCasePos (s, SubString: CString): CString;
  attribute (name = '_p_CStringLastCasePos'); external;
function  CStringUpCase     (s: CString): CString; attribute (name
  = '_p_CStringUpCase'); external;
function  CStringLoCase     (s: CString): CString; attribute (name
  = '_p_CStringLoCase'); external;
function  CStringIsEmpty    (s: CString): Boolean; attribute (name
  = '_p_CStringIsEmpty'); external;
function  NewCString        (const Source: String): CString;
  attribute (name = '_p_NewCString'); external;
function  CStringCopyString (Dest: CString; const Source: String):
  CString; attribute (name = '_p_CStringCopyString'); external;
procedure CopyCString       (Source: CString; var Dest: String);
  attribute (name = '_p_CopyCString'); external;
```

```
function  NewString       (const s: String) = Result: PString;
  attribute (name = '_p_NewString'); external;
procedure DisposeString   (p: PString); external name '_p_Dispose';

procedure SetString       (var s: String; Buffer: PChar; Count:
  Integer); attribute (name = '_p_SetString'); external;
function  StringOfChar    (ch: Char; Count: Integer) = s: TString;
  attribute (name = '_p_StringOfChar'); external;

procedure TrimLeft        (var s: String); attribute (name
  = '_p_TrimLeft'); external;
procedure TrimRight       (var s: String); attribute (name
  = '_p_TrimRight'); external;
procedure TrimBoth        (var s: String); attribute (name
  = '_p_TrimBoth'); external;
function  TrimLeftStr     (const s: String) = Result: TString;
  attribute (name = '_p_TrimLeftStr'); external;
function  TrimRightStr    (const s: String) = Result: TString;
  attribute (name = '_p_TrimRightStr'); external;
function  TrimBothStr     (const s: String) = Result: TString;
  attribute (name = '_p_TrimBothStr'); external;
function  LTrim           (const s: String) = Result: TString;
  external name '_p_TrimLeftStr';

function  GetStringCapacity (const s: String): Integer; attribute
  (name = '_p_GetStringCapacity'); external;

{ A shortcut for a common use of WriteStr as a function }
function  Integer2String (i: Integer) = s: Str64; attribute (name
  = '_p_Integer2String'); external;

{ Convert integer n to string in base Base. }
function  Integer2StringBase (n: LongestInt; Base:
  TInteger2StringBase): TString; attribute (name
  = '_p_Integer2StringBase'); external;

{ Convert integer n to string in base Base, with sign, optionally in
  uppercase representation and with printed base, padded with
  leading zeroes between `[<Sign>]<Base>#' and the actual digits to
  specified Width. }
function  Integer2StringBaseExt (n: LongestInt; Base:
  TInteger2StringBase; Width: TInteger2StringWidth; Upper: Boolean;
  PrintBase: Boolean): TString; attribute (name
  = '_p_Integer2StringBaseExt'); external;

{ String handling routines (higher level), from string2.pas }

type
  PChars0 = ^TChars0;
  TChars0 = array [0 .. MaxVarSize div SizeOf (Char) - 1] of Char;
```

```
  PPChars0 = ^TPChars0;
  TPChars0 = array [0 .. MaxVarSize div SizeOf (PChars0) - 1] of
  PChars0;

  PChars = ^TChars;
  TChars = packed array [1 .. MaxVarSize div SizeOf (Char)] of Char;

  { Under development. Interface subject to change.
    Use with caution. }
  { When a const or var AnyString parameter is passed, internally
    these records are passed as const parameters. Value AnyString
    parameters are passed like value string parameters. }
  ConstAnyString = record
    Length: Integer;
    Chars: PChars
  end;

  { Capacity is the allocated space (used internally). Count is the
    actual number of environment strings. The CStrings array
    contains the environment strings, terminated by a nil pointer,
    which is not counted in Count. @CStrings can be passed to libc
    routines like execve which expect an environment (see
    GetCEnvironment). }
  PEnvironment = ^TEnvironment;
  TEnvironment (Capacity: Integer) = record
    Count: Integer;
    CStrings: array [1 .. Capacity + 1] of CString
  end;

var
  Environment: PEnvironment; attribute (name = '_p_Environment');
  external;

{ Get an environment variable. If it does not exist, GetEnv returns
  the empty string, which can't be distinguished from a variable
  with an empty value, while CStringGetEnv returns nil then. Note,
  Dos doesn't know empty environment variables, but treats them as
  non-existing, and does not distinguish case in the names of
  environment variables. However, even under Dos, empty environment
  variables and variable names with different case can now be set
  and used within GPC programs. }
function  GetEnv (const EnvVar: String): TString; attribute (name
  = '_p_GetEnv'); external;
function  CStringGetEnv (EnvVar: CString): CString; attribute (name
  = '_p_CStringGetEnv'); external;

{ Sets an environment variable with the name given in VarName to the
  value Value. A previous value, if any, is overwritten. }
procedure SetEnv (const VarName, Value: String); attribute (name
  = '_p_SetEnv'); external;
```

```
{ Un-sets an environment variable with the name given in VarName. }
procedure UnSetEnv (const VarName: String); attribute (name
  = '_p_UnSetEnv'); external;

{ Returns @Environment^.CStrings, converted to PCStrings, to be
  passed to libc routines like execve which expect an environment. }
function  GetCEnvironment: PCStrings; attribute (name
  = '_p_GetCEnvironment'); external;

type
  FormatStringTransformType = ^function (const Format: String):
  TString;

var
  FormatStringTransformPtr: FormatStringTransformType; attribute
  (name = '_p_FormatStringTransformPtr'); external;

{ Runtime error and signal handling routines, from error.pas }

const
  EAssert = 306;
  EAssertString = 307;
  EOpen = 405;
  EMMap = 408;
  ERead = 413;
  EWrite = 414;
  EWriteReadOnly = 422;
  ENonExistentFile = 436;
  EOpenRead = 442;
  EOpenWrite = 443;
  EOpenUpdate = 444;
  EReading = 464;
  EWriting = 466;
  ECannotWriteAll = 467;
  ECannotFork = 600;
  ECannotSpawn = 601;
  EProgramNotFound = 602;
  EProgramNotExecutable = 603;
  EPipe = 604;
  EPrinterRead = 610;
  EIOCtl = 630;
  EConvertError = 875;
  ELibraryFunction = 952;
  EExitReturned = 953;

  RuntimeErrorExitValue = 42;

var
  { Error number (after runtime error) or exit status (after Halt)
    or 0 (during program run and after succesful termination). }
```

```
  ExitCode: Integer; attribute (name = '_p_ExitCode'); external;

  { Contains the address of the code where a runtime occurred, nil
    if no runtime error occurred. }
  ErrorAddr: Pointer; attribute (name = '_p_ErrorAddr'); external;

  { Error message }
  ErrorMessageString: TString; attribute (name
  = '_p_ErrorMessageString'); external;

  { String parameter to some error messages, *not* the text of the
    error message (the latter can be obtained with
    GetErrorMessage). }
  InOutResString: PString; attribute (name = '_p_InOutResString');
  external;

  { Optional libc error string to some error messages. }
  InOutResCErrorString: PString; attribute (name
  = '_p_InOutResCErrorString'); external;

  RTSErrorFD: Integer; attribute (name = '_p_ErrorFD'); external;
  RTSErrorFileName: PString; attribute (name = '_p_ErrorFileName');
  external;

function  GetErrorMessage                 (n: Integer): CString;
  attribute (name = '_p_GetErrorMessage'); external;
procedure RuntimeError                    (n: Integer); attribute
  (noreturn, name = '_p_RuntimeError'); external;
procedure RuntimeErrorErrNo               (n: Integer); attribute
  (noreturn, name = '_p_RuntimeErrorErrNo'); external;
procedure RuntimeErrorInteger             (n: Integer; i: MedInt);
  attribute (noreturn, name = '_p_RuntimeErrorInteger'); external;
procedure RuntimeErrorCString             (n: Integer; s: CString);
  attribute (noreturn, name = '_p_RuntimeErrorCString'); external;
procedure InternalError                   (n: Integer); attribute
  (noreturn, name = '_p_InternalError'); external;
procedure InternalErrorInteger            (n: Integer; i: MedInt);
  attribute (noreturn, name = '_p_InternalErrorInteger'); external;
procedure InternalErrorCString            (n: Integer; s: CString);
  attribute (noreturn, name = '_p_InternalErrorCString'); external;
procedure RuntimeWarning                  (Message: CString);
  attribute (name = '_p_RuntimeWarning'); external;
procedure RuntimeWarningInteger           (Message: CString; i:
  MedInt); attribute (name = '_p_RuntimeWarningInteger'); external;
procedure RuntimeWarningCString           (Message: CString; s:
  CString); attribute (name = '_p_RuntimeWarningCString'); external;

procedure IOError                         (n: Integer; ErrNoFlag:
  Boolean); attribute (iocritical, name = '_p_IOError'); external;
procedure IOErrorInteger                  (n: Integer; i: MedInt;
  ErrNoFlag: Boolean); attribute (iocritical, name
```

```
  = '_p_IOErrorInteger'); external;
procedure IOErrorCString                  (n: Integer; s: CString;
  ErrNoFlag: Boolean); attribute (iocritical, name
  = '_p_IOErrorCString'); external;

function  GetIOErrorMessage = Res: TString; attribute (name
  = '_p_GetIOErrorMessage'); external;
procedure CheckInOutRes; attribute (name = '_p_CheckInOutRes');
  external;

{ Registers a procedure to be called to restore the terminal for
  another process that accesses the terminal, or back for the
  program itself. Used e.g. by the CRT unit. The procedures must
  allow for being called multiple times in any order, even at the
  end of the program (see the comment for RestoreTerminal). }
procedure RegisterRestoreTerminal (ForAnotherProcess: Boolean;
  procedure Proc); attribute (name = '_p_RegisterRestoreTerminal');
  external;

{ Unregisters a procedure registered with RegisterRestoreTerminal.
  Returns False if the procedure had not been registered, and True
  if it had been registered and was unregistered successfully. }
function  UnregisterRestoreTerminal (ForAnotherProcess: Boolean;
  procedure Proc): Boolean; attribute (name
  = '_p_UnregisterRestoreTerminal'); external;

{ Calls the procedures registered by RegisterRestoreTerminal. When
  restoring the terminal for another process, the procedures are
  called in the opposite order of registration. When restoring back
  for the program, they are called in the order of registration.

  `RestoreTerminal (True)' will also be called at the end of the
  program, before outputting any runtime error message. It can also
  be used if you want to write an error message and exit the program
  (especially when using e.g. the CRT unit). For this purpose, to
  avoid side effects, call RestoreTerminal immediately before
  writing the error message (to StdErr, not to Output!), and then
  exit the program (e.g. with Halt). }
procedure RestoreTerminal (ForAnotherProcess: Boolean); attribute
  (name = '_p_RestoreTerminal'); external;

procedure AtExit (procedure Proc); attribute (name = '_p_AtExit');
  external;

function  ReturnAddr2Hex (p: Pointer) = s: TString; attribute (name
  = '_p_ReturnAddr2Hex'); external;

{ This function is used to write error messages etc. It does not use
  the Pascal I/O system here because it is usually called at the
  very end of a program after the Pascal I/O system has been shut
  down. }
```

```
function  WriteErrorMessage (const s: String; StdErrFlag: Boolean):
  Boolean; attribute (name = '_p_WriteErrorMessage'); external;

procedure SetReturnAddress (Address: Pointer); attribute (name
  = '_p_SetReturnAddress'); external;
procedure RestoreReturnAddress; attribute (name
  = '_p_RestoreReturnAddress'); external;

{ Returns a description for a signal }
function  StrSignal (Signal: Integer) = Res: TString; attribute
  (name = '_p_StrSignal'); external;

{ Installs some signal handlers that cause runtime errors on certain
  signals. This procedure runs only once, and returns immediately
  when called again (so you can't use it to set the signals again if
  you changed them meanwhile). @@Does not work on all systems (since
  the handler might have too little stack space). }
procedure InstallDefaultSignalHandlers; attribute (name
  = '_p_InstallDefaultSignalHandlers'); external;

var
  { Signal actions }
  SignalDefault: TSignalHandler; attribute (const); external
  name '_p_SIG_DFL';
  SignalIgnore : TSignalHandler; attribute (const); external
  name '_p_SIG_IGN';
  SignalError  : TSignalHandler; attribute (const); external
  name '_p_SIG_ERR';

  { Signals. The constants are set to the signal numbers, and
    are 0 for signals not defined. }
  { POSIX signals }
  SigHUp   : Integer; attribute (const); external name '_p_SIGHUP';
  SigInt   : Integer; attribute (const); external name '_p_SIGINT';
  SigQuit  : Integer; attribute (const); external name '_p_SIGQUIT';
  SigIll   : Integer; attribute (const); external name '_p_SIGILL';
  SigAbrt  : Integer; attribute (const); external name '_p_SIGABRT';
  SigFPE   : Integer; attribute (const); external name '_p_SIGFPE';
  SigKill  : Integer; attribute (const); external name '_p_SIGKILL';
  SigSegV  : Integer; attribute (const); external name '_p_SIGSEGV';
  SigPipe  : Integer; attribute (const); external name '_p_SIGPIPE';
  SigAlrm  : Integer; attribute (const); external name '_p_SIGALRM';
  SigTerm  : Integer; attribute (const); external name '_p_SIGTERM';
  SigUsr1  : Integer; attribute (const); external name '_p_SIGUSR1';
  SigUsr2  : Integer; attribute (const); external name '_p_SIGUSR2';
  SigChld  : Integer; attribute (const); external name '_p_SIGCHLD';
  SigCont  : Integer; attribute (const); external name '_p_SIGCONT';
  SigStop  : Integer; attribute (const); external name '_p_SIGSTOP';
  SigTStp  : Integer; attribute (const); external name '_p_SIGTSTP';
  SigTTIn  : Integer; attribute (const); external name '_p_SIGTTIN';
  SigTTOu  : Integer; attribute (const); external name '_p_SIGTTOU';
```

```
  { Non-POSIX signals }
  SigTrap  : Integer; attribute (const); external name '_p_SIGTRAP';
  SigIOT   : Integer; attribute (const); external name '_p_SIGIOT';
  SigEMT   : Integer; attribute (const); external name '_p_SIGEMT';
  SigBus   : Integer; attribute (const); external name '_p_SIGBUS';
  SigSys   : Integer; attribute (const); external name '_p_SIGSYS';
  SigStkFlt: Integer; attribute (const); external
  name '_p_SIGSTKFLT';
  SigUrg   : Integer; attribute (const); external name '_p_SIGURG';
  SigIO    : Integer; attribute (const); external name '_p_SIGIO';
  SigPoll  : Integer; attribute (const); external name '_p_SIGPOLL';
  SigXCPU  : Integer; attribute (const); external name '_p_SIGXCPU';
  SigXFSz  : Integer; attribute (const); external name '_p_SIGXFSZ';
  SigVTAlrm: Integer; attribute (const); external
  name '_p_SIGVTALRM';
  SigProf  : Integer; attribute (const); external name '_p_SIGPROF';
  SigPwr   : Integer; attribute (const); external name '_p_SIGPWR';
  SigInfo  : Integer; attribute (const); external name '_p_SIGINFO';
  SigLost  : Integer; attribute (const); external name '_p_SIGLOST';
  SigWinCh : Integer; attribute (const); external
  name '_p_SIGWINCH';

  { Signal subcodes (only used on some systems, -1 if not used) }
  FPEIntegerOverflow      : Integer; attribute (const); external
  name '_p_FPE_INTOVF_TRAP';
  FPEIntegerDivisionByZero: Integer; attribute (const); external
  name '_p_FPE_INTDIV_TRAP';
  FPESubscriptRange       : Integer; attribute (const); external
  name '_p_FPE_SUBRNG_TRAP';
  FPERealOverflow         : Integer; attribute (const); external
  name '_p_FPE_FLTOVF_TRAP';
  FPERealDivisionByZero   : Integer; attribute (const); external
  name '_p_FPE_FLTDIV_TRAP';
  FPERealUnderflow        : Integer; attribute (const); external
  name '_p_FPE_FLTUND_TRAP';
  FPEDecimalOverflow      : Integer; attribute (const); external
  name '_p_FPE_DECOVF_TRAP';

{ Routines called implicitly by the compiler. }
procedure GPC_Assert (Condition: Boolean; const Message: String);
  attribute (name = '_p_Assert'); external;
function  ObjectTypeIs (Left, Right: PObjectType): Boolean;
  attribute (const, name = '_p_ObjectTypeIs'); external;
procedure ObjectTypeAsError; attribute (noreturn, name
  = '_p_ObjectTypeAsError'); external;
procedure DisposeNilError; attribute (noreturn, name
  = '_p_DisposeNilError'); external;
procedure CaseNoMatchError; attribute (noreturn, name
  = '_p_CaseNoMatchError'); external;
procedure DiscriminantsMismatchError; attribute (noreturn, name
```

```
  = '_p_DiscriminantsMismatchError'); external;
procedure NilPointerError; attribute (noreturn, name
  = '_p_NilPointerError'); external;
procedure InvalidPointerError (p: Pointer); attribute (noreturn,
  name = '_p_InvalidPointerError'); external;
procedure InvalidObjectError; attribute (noreturn, name
  = '_p_InvalidObjectError'); external;
procedure RangeCheckError; attribute (noreturn, name
  = '_p_RangeCheckError'); external;
procedure IORangeCheckError; attribute (name
  = '_p_IORangeCheckError'); external;
procedure SubrangeError; attribute (noreturn, name
  = '_p_SubrangeError'); external;
procedure ModRangeError; attribute (noreturn, name
  = '_p_ModRangeError'); external;

{ Pointer checking with '--pointer-checking-user-defined' }

procedure DefaultValidatePointer (p: Pointer); attribute (name
  = '_p_DefaultValidatePointer'); external;

type
  ValidatePointerType = ^procedure (p: Pointer);

var
  ValidatePointerPtr: ValidatePointerType; attribute (name
  = '_p_ValidatePointerPtr'); external;

{ Time and date routines, from time.pas }

const
  InvalidYear = -MaxInt;

var
  { DayOfWeekName is a constant and therefore does not respect the
    locale. Therefore, it's recommended to use FormatTime instead. }
  DayOfWeekName: array [0 .. 6] of String [9]; attribute (const,
  name = '_p_DayOfWeekName'); external;

  { MonthName is a constant and therefore does not respect the
    locale. Therefore, it's recommended to use FormatTime instead. }
  MonthName: array [1 .. 12] of String [9]; attribute (const, name
  = '_p_MonthName'); external;

function  GetDayOfWeek (Day, Month, Year: Integer): Integer;
  attribute (name = '_p_GetDayOfWeek'); external;
function  GetDayOfYear (Day, Month, Year: Integer): Integer;
  attribute (name = '_p_GetDayOfYear'); external;
function  GetSundayWeekOfYear (Day, Month, Year: Integer): Integer;
  attribute (name = '_p_GetSundayWeekOfYear'); external;
function  GetMondayWeekOfYear (Day, Month, Year: Integer): Integer;
```

```
  attribute (name = '_p_GetMondayWeekOfYear'); external;
procedure GetISOWeekOfYear (Day, Month, Year: Integer; var ISOWeek,
  ISOWeekYear: Integer); attribute (name = '_p_GetISOWeekOfYear');
  external;
procedure UnixTimeToTimeStamp (UnixTime: UnixTimeType; var
  aTimeStamp: TimeStamp); attribute (name
  = '_p_UnixTimeToTimeStamp'); external;
function  TimeStampToUnixTime (protected var aTimeStamp: TimeStamp):
  UnixTimeType; attribute (name = '_p_TimeStampToUnixTime');
  external;
function  GetMicroSecondTime: MicroSecondTimeType; attribute (name
  = '_p_GetMicroSecondTime'); external;

{ Is the year a leap year? }
function  IsLeapYear (Year: Integer): Boolean; attribute (name
  = '_p_IsLeapYear'); external;

{ Returns the length of the month, taking leap years into account. }
function  MonthLength (Month, Year: Integer): Integer; attribute
  (name = '_p_MonthLength'); external;

{ Formats a TimeStamp value according to a Format string. The format
  string can contain date/time items consisting of `%', followed by
  the specifiers listed below. All characters outside of these items
  are copied to the result unmodified. The specifiers correspond to
  those of the C function strftime(), including POSIX.2 and glibc
  extensions and some more extensions. The extensions are also
  available on systems whose strftime() doesn't support them.

  The following modifiers may appear after the `%':

  `_'  The item is left padded with spaces to the given or default
       width.

  `-'  The item is not padded at all.

  `0'  The item is left padded with zeros to the given or default
       width.

  `/'  The item is right trimmed if it is longer than the given
       width.

  `^'  The item is converted to upper case.

  `~'  The item is converted to lower case.

  After zero or more of these flags, an optional width may be
  specified for padding and trimming. It must be given as a decimal
  number (not starting with `0' since `0' has a meaning of its own,
  see above).
```

Afterwards, the following optional modifiers may follow. Their meaning is locale-dependent, and many systems and locales just ignore them.

'E' Use the locale's alternate representation for date and time. In a Japanese locale, for example, '%Ex' might yield a date format based on the Japanese Emperors' reigns.

'O' Use the locale's alternate numeric symbols for numbers. This modifier applies only to numeric format specifiers.

Finally, exactly one of the following specifiers must appear. The padding rules listed here are the defaults that can be overriden with the modifiers listed above.

'a' The abbreviated weekday name according to the current locale.

'A' The full weekday name according to the current locale.

'b' The abbreviated month name according to the current locale.

'B' The full month name according to the current locale.

'c' The preferred date and time representation for the current locale.

'C' The century of the year. This is equivalent to the greatest integer not greater than the year divided by 100.

'd' The day of the month as a decimal number ('01' .. '31').

'D' The date using the format '%m/%d/%y'. NOTE: Don't use this format if it can be avoided. Things like this caused Y2K bugs!

'e' The day of the month like with '%d', but padded with blanks (' 1' .. '31').

'F' The date using the format '%Y-%m-%d'. This is the form specified in the ISO 8601 standard and is the preferred form for all uses.

'g' The year corresponding to the ISO week number, but without the century ('00' .. '99'). This has the same format and value as 'y', except that if the ISO week number (see 'V') belongs to the previous or next year, that year is used instead. NOTE: Don't use this format if it can be avoided. Things like this caused Y2K bugs!

'G' The year corresponding to the ISO week number. This has the same format and value as 'Y', except that if the ISO week number (see ‘V’) belongs to the previous or next year, that year is used instead.

‘h’ The abbreviated month name according to the current locale. This is the same as ‘b’.

‘H’ The hour as a decimal number, using a 24-hour clock (‘00’ .. ‘23’).

‘I’ The hour as a decimal number, using a 12-hour clock (‘01’ .. ‘12’).

‘j’ The day of the year as a decimal number (‘001’ .. ‘366’).

‘k’ The hour as a decimal number, using a 24-hour clock like ‘H’, but padded with blanks (‘ 0’ .. ‘23’).

‘l’ The hour as a decimal number, using a 12-hour clock like ‘I’, but padded with blanks (‘ 1’ .. ‘12’).

‘m’ The month as a decimal number (‘01’ .. ‘12’).

‘M’ The minute as a decimal number (‘00’ .. ‘59’).

‘n’ A single newline character.

‘p’ Either ‘AM’ or ‘PM’, according to the given time value; or the corresponding strings for the current locale. Noon is treated as ‘PM’ and midnight as ‘AM’.

‘P’ Either ‘am’ or ‘pm’, according to the given time value; or the corresponding strings for the current locale, printed in lowercase characters. Noon is treated as ‘pm’ and midnight as ‘am’.

‘Q’ The fractional part of the second. This format has special effects on the modifiers. The width, if given, determines the number of digits to output. Therefore, no actual clipping or trimming is done. However, if padding with spaces is specified, any trailing (i.e., right!) zeros are converted to spaces, and if "no padding" is specified, they are removed. The default is "padding with zeros", i.e. trailing zeros are left unchanged. The digits are cut when necessary without rounding (otherwise, the value would not be consistent with the seconds given by ‘S’ and ‘s’). Note that GPC’s TimeStamp currently provides for microsecond resolution, so there are at most 6 valid digits (which is also the default width), any further digits will be 0 (but if TimeStamp will ever change, this format will be adjusted). However, the actual resolution provided by the operating system via GetTimeStamp etc. may be far lower (e.g., ~1/18s under Dos).

'r' The complete time using the AM/PM format of the current locale.

'R' The hour and minute in decimal numbers using the format '%H:%M'.

's' Unix time, i.e. the number of seconds since the epoch, i.e., since 1970-01-01 00:00:00 UTC. Leap seconds are not counted unless leap second support is available.

'S' The seconds as a decimal number ('00' .. '60').

't' A single tab character.

'T' The time using decimal numbers using the format '%H:%M:%S'.

'u' The day of the week as a decimal number ('1' .. '7'), Monday being '1'.

'U' The week number of the current year as a decimal number ('00' .. '53'), starting with the first Sunday as the first day of the first week. Days preceding the first Sunday in the year are considered to be in week '00'.

'V' The ISO 8601:1988 week number as a decimal number ('01' .. '53'). ISO weeks start with Monday and end with Sunday. Week '01' of a year is the first week which has the majority of its days in that year; this is equivalent to the week containing the year's first Thursday, and it is also equivalent to the week containing January 4. Week '01' of a year can contain days from the previous year. The week before week '01' of a year is the last week ('52' or '53') of the previous year even if it contains days from the new year.

'w' The day of the week as a decimal number ('0' .. '6'), Sunday being '0'.

'W' The week number of the current year as a decimal number ('00' .. '53'), starting with the first Monday as the first day of the first week. All days preceding the first Monday in the year are considered to be in week '00'.

'x' The preferred date representation for the current locale, but without the time.

'X' The preferred time representation for the current locale, but with no date.

'y' The year without a century as a decimal number ('00' .. '99'). This is equivalent to the year modulo 100.

```
     NOTE: Don't use this format if it can be avoided. Things like
     this caused Y2K bugs!

  `Y'  The year as a decimal number, using the Gregorian calendar.
     Years before the year `1' are numbered `0', `-1', and so on.

  `z'  RFC 822/ISO 8601:1988 style numeric time zone (e.g., `-0600'
     or `+0100'), or nothing if no time zone is determinable.

  `Z'  The time zone abbreviation (empty if the time zone can't be
     determined).

  `%'  (i.e., an item `%%') A literal `%' character. }
function  FormatTime (const Time: TimeStamp; const Format: String) =
  Res: TString; attribute (name = '_p_FormatTime'); external;

{ Pseudo random number generator, from random.pas }

type
  RandomSeedType = Cardinal attribute (Size = 32);
  RandomizeType  = ^procedure;
  SeedRandomType = ^procedure (Seed: RandomSeedType);
  RandRealType   = ^function: LongestReal;
  RandIntType    = ^function (MaxValue: LongestCard): LongestCard;

procedure SeedRandom (Seed: RandomSeedType); attribute (name
  = '_p_SeedRandom'); external;

var
  RandomizePtr : RandomizeType; attribute (name
  = '_p_RandomizePtr'); external;
  SeedRandomPtr: SeedRandomType; attribute (name
  = '_p_SeedRandomPtr'); external;
  RandRealPtr  : RandRealType; attribute (name = '_p_RandRealPtr');
  external;
  RandIntPtr   : RandIntType; attribute (name = '_p_RandIntPtr');
  external;

{ File name routines, from fname.pas }

{ Define constants for different systems:

  OSDosFlag:         flag to indicate whether the target system is
                     Dos

  QuotingCharacter:  the character used to quote wild cards and
                     other special characters (#0 if not available)

  PathSeparator:     the separator of multiple paths, e.g. in the
                     PATH environment variable
```

```
DirSeparator:      the separator of the directories within a full
                   file name

DirSeparators:     a set of all possible directory and drive name
                   separators

ExtSeparator:      the separator of a file name extension

DirRoot:           the name of the root directory

DirSelf:           the name of a directory in itself

DirParent:         the name of the parent directory

MaskNoStdDir:      a file name mask that matches all names except
                   the standard directories DirSelf and DirParent

NullDeviceName:    the full file name of the null device

TtyDeviceName:     the full file name of the current Tty

ConsoleDeviceName: the full file name of the system console. On
                   Dos systems, this is the same as the Tty, but
                   on systems that allow remote login, this is a
                   different thing and may reach a completely
                   different user than the one running the
                   program, so use it with care.

EnvVarCharsFirst:  the characters accepted at the beginning of the
                   name of an environment variable without quoting

EnvVarChars:       the characters accepted in the name of an
                   environment variable without quoting

PathEnvVar:        the name of the environment variable which
                   (usually) contains the executable search path

ShellEnvVar:       the name of the environment variable which
                   (usually) contains the path of the shell
                   executable (see GetShellPath)

ShellExecCommand:  the option to the (default) shell to execute
                   the command specified in the following argument
                   (see GetShellPath)

ConfigFileMask:    a mask for the option file name as returned by
                   ConfigFileName

FileNamesCaseSensitive:
                   flag to indicate whether file names are case
                   sensitive }
```

```
const
  UnixShellEnvVar        = 'SHELL';
  UnixShellExecCommand   = '-c';

{$ifdef __OS_DOS__}

{$if defined (__CYGWIN__) or defined(__MSYS__)}
  {$define __POSIX_WIN32__}
{$endif}

const
  OSDosFlag              = True;
  QuotingCharacter       = #0;
  PathSeparator          = {$ifdef __POSIX_WIN32__} ':' {$else} ';'
  {$endif};
  DirSeparator           = '\';
  DirSeparators          = [':', '\', '/'];
  ExtSeparator           = '.';
  DirRoot                = '\';
  DirSelf                = '.';
  DirParent              = '..';
  MaskNoStdDir           = '{*,.[^.]*,..?*}';
  NullDeviceName         = 'nul';
  TtyDeviceName          = 'con';
  ConsoleDeviceName      = 'con';
  EnvVarCharsFirst       = ['A' .. 'Z', 'a' .. 'z', '_'];
  EnvVarChars            = EnvVarCharsFirst + ['0' .. '9'];
  PathEnvVar             = 'PATH';
  ShellEnvVar            = 'COMSPEC';
  ShellExecCommand       = '/c';
  ConfigFileMask         = '*.cfg';
  FileNamesCaseSensitive = False;

{$else}

const
  OSDosFlag              = False;
  QuotingCharacter       = '\';
  PathSeparator          = ':';
  DirSeparator           = '/';
  DirSeparators          = ['/'];
  ExtSeparator           = '.';
  DirRoot                = '/';
  DirSelf                = '.';
  DirParent              = '..';
  MaskNoStdDir           = '{*,.[^.]*,..?*}';
  NullDeviceName         = '/dev/null';
  TtyDeviceName          = '/dev/tty';
  ConsoleDeviceName      = '/dev/console';
  EnvVarCharsFirst       = ['A' .. 'Z', 'a' .. 'z', '_'];
```

```
  EnvVarChars            = EnvVarCharsFirst + ['0' .. '9'];
  PathEnvVar             = 'PATH';
  ShellEnvVar            = UnixShellEnvVar;
  ShellExecCommand       = UnixShellExecCommand;
  ConfigFileMask         = '.*';
  FileNamesCaseSensitive = True;

{$endif}

const
  WildCardChars = ['*', '?', '[', ']'];
  FileNameSpecialChars = (WildCardChars + SpaceCharacters +
  ['{', '}', '$', QuotingCharacter]) - DirSeparators;

type
  DirPtr = Pointer;

{ Convert ch to lower case if FileNamesCaseSensitive is False, leave
  it unchanged otherwise. }
function  FileNameLoCase (ch: Char): Char; attribute (name
  = '_p_FileNameLoCase'); external;

{ Change a file name to use the OS dependent directory separator }
function  Slash2OSDirSeparator (const s: String) = Result: TString;
  attribute (name = '_p_Slash2OSDirSeparator'); external;

{ Change a file name to use '/' as directory separator }
function  OSDirSeparator2Slash (const s: String) = Result: TString;
  attribute (name = '_p_OSDirSeparator2Slash'); external;

{ Like Slash2OSDirSeparator for CStrings. *Note*: overwrites the
  CString }
function  Slash2OSDirSeparator_CString (s: CString): CString;
  attribute (ignorable, name = '_p_Slash2OSDirSeparator_CString');
  external;

{ Like OSDirSeparator2Slash for CStrings. *Note*: overwrites the
  CString }
function  OSDirSeparator2Slash_CString (s: CString): CString;
  attribute (ignorable, name = '_p_OSDirSeparator2Slash_CString');
  external;

{ Add a DirSeparator to the end of s, if there is not already one
  and s denotes an existing directory }
function  AddDirSeparator (const s: String) = Result: TString;
  attribute (name = '_p_AddDirSeparator'); external;

{ Like AddDirSeparator, but also if the directory does not exist }
function  ForceAddDirSeparator (const s: String) = Result: TString;
  attribute (name = '_p_ForceAddDirSeparator'); external;
```

```
{ Remove all trailing DirSeparators from s, if there are any, as
  long as removing them doesn't change the meaning (i.e., they don't
  denote the root directory. }
function  RemoveDirSeparator (const s: String) = Result: TString;
  attribute (name = '_p_RemoveDirSeparator'); external;

{ Returns the current directory using OS dependent directory
  separators }
function  GetCurrentDirectory: TString; attribute (name
  = '_p_GetCurrentDirectory'); external;

{ Returns a directory suitable for storing temporary files using OS
  dependent directory separators. If found, the result always ends
  in DirSeparator. If no suitable directory is found, an empty
  string is returned. }
function  GetTempDirectory: TString; attribute (name
  = '_p_GetTempDirectory'); external;

{ Returns a non-existing file name in the directory given. If the
  directory doesn't exist or the Directory name is empty, an I/O
  error is raised, and GetTempFileNameInDirectory returns the empty
  string. }
function  GetTempFileNameInDirectory (const Directory: String) =
  Result: TString; attribute (iocritical, name
  = '_p_GetTempFileNameInDirectory'); external;

{ Returns a non-existing file name in GetTempDirectory. If no temp
  directory is found, i.e. GetTempDirectory returns the empty
  string, an I/O error is raised, and GetTempFileName returns the
  empty string as well. }
function  GetTempFileName: TString; attribute (iocritical, name
  = '_p_GetTempFileName'); external;

{ The same as GetTempFileName, but returns a CString allocated from
  the heap. }
function  GetTempFileName_CString: CString; attribute (iocritical,
  name = '_p_GetTempFileName_CString'); external;

{ Returns True if the given file name is an existing plain file }
function  FileExists      (const aFileName: String): Boolean;
  attribute (name = '_p_FileExists'); external;

{ Returns True if the given file name is an existing directory }
function  DirectoryExists (const aFileName: String): Boolean;
  attribute (name = '_p_DirectoryExists'); external;

{ Returns True if the given file name is an existing file, directory
  or special file (device, pipe, socket, etc.) }
function  PathExists      (const aFileName: String): Boolean;
  attribute (name = '_p_PathExists'); external;
```

```
{ If a file of the given name exists in one of the directories given
  in DirList (separated by PathSeparator), returns the full path,
  otherwise returns an empty string. If aFileName already contains
  an element of DirSeparators, returns Slash2OSDirSeparator
  (aFileName) if it exists. }
function  FSearch (const aFileName, DirList: String): TString;
  attribute (name = '_p_FSearch'); external;

{ Like FSearch, but only find executable files. Under Dos, if not
  found, the function tries appending '.com', '.exe', '.bat' and
  `.cmd' (the last one only if $COMSPEC points to a `cmd.exe'), so
  you don't have to specify these extensions in aFileName (and with
  respect to portability, it might be preferable not to do so). }
function  FSearchExecutable (const aFileName, DirList: String) =
  Result: TString; attribute (name = '_p_FSearchExecutable');
  external;

{ Replaces all occurrences of `$FOO' and `~' in s by the value of
  the environment variables FOO or HOME, respectively. If a variable
  is not defined, the function returns False, and s contains the
  name of the undefined variable (or the empty string if the
  variable name is invalid, i.e., doesn't start with a character
  from EnvVarCharsFirst). Otherwise, if all variables are found, s
  contains the replaced string, and True is returned. }
function  ExpandEnvironment (var s: String): Boolean; attribute
  (name = '_p_ExpandEnvironment'); external;

{ Expands the given path name to a full path name. Relative paths
  are expanded using the current directory, and occurrences of
  DirSelf and DirParent are resolved. Under Dos, the result is
  converted to lower case and a trailing ExtSeparator (except in a
  trailing DirSelf or DirParent) is removed, like Dos does. If the
  directory, i.e. the path without the file name, is invalid, the
  empty string is returned. }
function  FExpand       (const Path: String): TString; attribute
  (name = '_p_FExpand'); external;

{ Like FExpand, but unquotes the directory before expanding it, and
  quotes WildCardChars again afterwards. Does not check if the
  directory is valid (because it may contain wild card characters).
  Symlinks are expanded only in the directory part, not the file
  name. }
function  FExpandQuoted (const Path: String): TString; attribute
  (name = '_p_FExpandQuoted'); external;

{ FExpands Path, and then removes the current directory from it, if
  it is a prefix of it. If OnlyCurDir is set, the current directory
  will be removed only if Path denotes a file in, not below, it. }
function  RelativePath (const Path: String; OnlyCurDir, Quoted:
  Boolean) = Result: TString; attribute (name = '_p_RelativePath');
  external;
```

```
{ Is aFileName a UNC filename? (Always returns False on non-Dos
  systems.) }
function  IsUNC (const aFileName: String): Boolean; attribute (name
  = '_p_IsUNC'); external;

{ Splits a file name into directory, name and extension. Each of
  Dir, BaseName and Ext may be Null. }
procedure FSplit (const Path: String; var Dir, BaseName, Ext:
  String); attribute (name = '_p_FSplit'); external;

{ Functions that extract one or two of the parts from FSplit.
  DirFromPath returns DirSelf + DirSeparator if the path contains no
  directory. }
function  DirFromPath     (const Path: String) = Dir: TString;
  attribute (name = '_p_DirFromPath'); external;
function  NameFromPath    (const Path: String) = BaseName: TString;
  attribute (name = '_p_NameFromPath'); external;
function  ExtFromPath     (const Path: String) = Ext: TString;
  attribute (name = '_p_ExtFromPath'); external;
function  NameExtFromPath (const Path: String): TString; attribute
  (name = '_p_NameExtFromPath'); external;

{ Start reading a directory. If successful, a pointer is returned
  that can be used for subsequent calls to ReadDir and finally
  CloseDir. On failure, an I/O error is raised and (in case it is
  ignored) nil is returned. }
function  OpenDir  (const DirName: String) = Res: DirPtr; attribute
  (iocritical, name = '_p_OpenDir'); external;

{ Reads one entry from the directory Dir, and returns the file name.
  On errors or end of directory, the empty string is returned. }
function  ReadDir  (Dir: DirPtr): TString; attribute (name
  = '_p_ReadDir'); external;

{ Closes a directory opened with OpenDir. }
procedure CloseDir (Dir: DirPtr); attribute (name = '_p_CloseDir');
  external;

{ Returns the first position of a non-quoted character of CharSet in
  s, or 0 if no such character exists. }
function  FindNonQuotedChar (Chars: CharSet; const s: String; From:
  Integer): Integer; attribute (name = '_p_FindNonQuotedChar');
  external;

{ Returns the first occurence of SubString in s that is not quoted
  at the beginning, or 0 if no such occurence exists. }
function  FindNonQuotedStr (const SubString, s: String; From:
  Integer): Integer; attribute (name = '_p_FindNonQuotedStr');
  external;
```

```
{ Does a string contain non-quoted wildcard characters? }
function  HasWildCards (const s: String): Boolean; attribute (name
  = '_p_HasWildCards'); external;

{ Does a string contain non-quoted wildcard characters, braces or
  spaces? }
function  HasWildCardsOrBraces (const s: String): Boolean; attribute
  (name = '_p_HasWildCardsOrBraces'); external;

{ Insert QuotingCharacter into s before any special characters }
function  QuoteFileName (const s: String; const SpecialCharacters:
  CharSet) = Result: TString; attribute (name = '_p_QuoteFileName');
  external;

{ Remove QuotingCharacter from s }
function  UnQuoteFileName (const s: String) = Result: TString;
  attribute (name = '_p_UnQuoteFileName'); external;

{ Splits s at non-quoted spaces and expands non-quoted braces like
  bash does. The result and its entries should be disposed after
  usage, e.g. with DisposePPStrings. }
function  BraceExpand (const s: String) = Result: PPStrings;
  attribute (name = '_p_BraceExpand'); external;

{ Dispose of a PPStrings array as well as the strings it contains.
  If you want to keep the strings (by assigning them to other string
  pointers), you should instead free the PPStrings array with
  `Dispose'. }
procedure DisposePPStrings (Strings: PPStrings); attribute (name
  = '_p_DisposePPStrings'); external;

{ Tests if a file name matches a shell wildcard pattern (?, *, []) }
function  FileNameMatch (const Pattern, FileName: String): Boolean;
  attribute (name = '_p_FileNameMatch'); external;

{ FileNameMatch with BraceExpand }
function  MultiFileNameMatch (const Pattern, FileName: String):
  Boolean; attribute (name = '_p_MultiFileNameMatch'); external;

{ File name globbing }
{ GlobInit is implied by Glob and MultiGlob, not by GlobOn and
  MultiGlobOn. GlobOn and MultiGlobOn must be called after GlobInit,
  Glob or MultiGlob. MultiGlob and MultiGlobOn do brace expansion,
  Glob and GlobOn do not. GlobFree frees the memory allocated by the
  globbing functions and invalidates the results in Buf. It should
  be called after globbing. }
procedure GlobInit    (var Buf: GlobBuffer); attribute (name
  = '_p_GlobInit'); external;
procedure Glob        (var Buf: GlobBuffer; const Pattern: String);
  attribute (name = '_p_Glob'); external;
procedure GlobOn      (var Buf: GlobBuffer; const Pattern: String);
```

```
  attribute (name = '_p_GlobOn'); external;
procedure MultiGlob   (var Buf: GlobBuffer; const Pattern: String);
  attribute (name = '_p_MultiGlob'); external;
procedure MultiGlobOn (var Buf: GlobBuffer; const Pattern: String);
  attribute (name = '_p_MultiGlobOn'); external;
procedure GlobFree    (var Buf: GlobBuffer); attribute (name
  = '_p_GlobFree'); external;

type
  TPasswordEntry = record
    UserName, RealName, Password, HomeDirectory, Shell: PString;
    UID, GID: Integer
  end;

  PPasswordEntries = ^TPasswordEntries;
  TPasswordEntries (Count: Integer) = array [1 .. Max (1, Count)] of
  TPasswordEntry;

{ Finds a password entry by user name. Returns True if found, False
  otherwise. }
function  GetPasswordEntryByName (const UserName: String; var Entry:
  TPasswordEntry) = Res: Boolean; attribute (name
  = '_p_GetPasswordEntryByName'); external;

{ Finds a password entry by UID. Returns True if found, False
  otherwise. }
function  GetPasswordEntryByUID (UID: Integer; var Entry:
  TPasswordEntry) = Res: Boolean; attribute (name
  = '_p_GetPasswordEntryByUID'); external;

{ Returns all password entries, or nil if none found. }
function  GetPasswordEntries = Res: PPasswordEntries; attribute
  (name = '_p_GetPasswordEntries'); external;

{ Dispose of a TPasswordEntry. }
procedure DisposePasswordEntry (Entry: TPasswordEntry); attribute
  (name = '_p_DisposePasswordEntry'); external;

{ Dispose of a PPasswordEntries. }
procedure DisposePasswordEntries (Entries: PPasswordEntries);
  attribute (name = '_p_DisposePasswordEntries'); external;

{ Returns the mount point (Unix) or drive (Dos) which is part of the
  given path. If the path does not contain any (i.e., is a relative
  path), an empty string is returned. Therefore, if you want to get
  the mount point or drive in any case, apply `FExpand' or
  `RealPath' to the argument. }
function  GetMountPoint (const Path: String) = Result: TString;
  attribute (name = '_p_GetMountPoint'); external;

type
```

```
  TSystemInfo = record
    OSName,
    OSRelease,
    OSVersion,
    MachineType,
    HostName,
    DomainName: TString
  end;

{ Returns system information if available. Fields not available will
  be empty. }
function  SystemInfo = Res: TSystemInfo; attribute (name
  = '_p_SystemInfo'); external;

{ Returns the path to the shell (as the result) and the option that
  makes it execute the command specified in the following argument
  (in `Option'). Usually these are the environment value of
  ShellEnvVar, and ShellExecCommand, but on Dos systems, the
  function will first try UnixShellEnvVar, and UnixShellExecCommand
  because ShellEnvVar will usually point to command.com, but
  UnixShellEnvVar can point to bash which is usually a better choice
  when present. If UnixShellEnvVar is not set, or the shell given
  does not exist, it will use ShellEnvVar, and ShellExecCommand.
  Option may be Null (in case you want to invoke the shell
  interactively). }
function  GetShellPath (var Option: String) = Res: TString;
  attribute (name = '_p_GetShellPath'); external;

{ Returns the path of the running executable. *Note*: On most
  systems, this is *not* guaranteed to be the full path, but often
  just the same as `ParamStr (0)' which usually is the name given on
  the command line. Only on some systems with special support, it
  returns the full path when `ParamStr (0)' doesn't. }
function  ExecutablePath: TString; attribute (name
  = '_p_ExecutablePath'); external;

{ Returns a file name suitable for a global (system-wide) or local
  (user-specific) configuration file, depending on the Global
  parameter. The function does not guarantee that the file name
  returned exists or is readable or writable.

  In the following table, the base name `<base>' is given with the
  BaseName parameter. If it is empty, the base name is the name of
  the running program (as returned by ExecutablePath, without
  directory and extension. `<prefix>' (Unix only) stands for the
  value of the Prefix parameter (usual values include '', '/usr' and
  '/usr/local'). `<dir>' (Dos only) stands for the directory where
  the running program resides. `$foo' stands for the value of the
  environment variable `foo'.

          Global                    Local
```

```
  Unix:   <prefix>/etc/<base>.conf  $HOME/.<base>

  DJGPP:  $DJDIR\etc\<base>.ini     $HOME\<base>.cfg
          <dir>\<base>.ini          <dir>\<base>.cfg

  Other   $HOME\<base>.ini          $HOME\<base>.cfg
    Dos:  <dir>\<base>.ini          <dir>\<base>.cfg

  As you see, there are two possibilities under Dos. If the first
  file exists, it is returned. Otherwise, if the second file exists,
  that is returned. If none of them exists (but the program might
  want to create a file), if the environment variable (DJDIR or
  HOME, respectively) is set, the first file name is returned,
  otherwise the second one. This rather complicated scheme should
  give the most reasonable results for systems with or without DJGPP
  installed, and with or without already existing config files. Note
  that DJDIR is always set on systems with DJGPP installed, while
  HOME is not. However, it is easy for users to set it if they want
  their config files in a certain directory rather than with the
  executables. }
function  ConfigFileName (const Prefix, BaseName: String; Global:
  Boolean): TString; attribute (name = '_p_ConfigFileName');
  external;

{ Returns a directory name suitable for global, machine-independent
  data. The function garantees that the name returned ends with a
  DirSeparator, but does not guarantee that it exists or is
  readable or writable.

  Note: If the prefix is empty, it is assumed to be '/usr'. (If you
  really want /share, you could pass '/' as the prefix, but that's
  very uncommon.)

  Unix:   <prefix>/share/<base>/

  DJGPP:  $DJDIR\share\<base>\
          <dir>\

  Other   $HOME\<base>\
    Dos:  <dir>\

  About the symbols used above, and the two possibilities under Dos,
  see the comments for ConfigFileName. }
function  DataDirectoryName (const Prefix, BaseName: String):
  TString; attribute (name = '_p_DataDirectoryName'); external;

{ Executes a command line. Reports execution errors via the IOResult
  mechanism and returns the exit status of the executed program.
  Execute calls RestoreTerminal with the argument True before and
  False after executing the process, ExecuteNoTerminal does not. }
function  Execute (const CmdLine: String): Integer; attribute
```

```
  (iocritical, name = '_p_Execute'); external;
function  ExecuteNoTerminal (const CmdLine: String): Integer;
  attribute (iocritical, name = '_p_ExecuteNoTerminal'); external;

{ File handling routines, from files.pas }

type
  TextFile = Text;
  TOpenMode = (fo_None, fo_Reset, fo_Rewrite, fo_Append,
  fo_SeekRead, fo_SeekWrite, fo_SeekUpdate);
  PAnyFile = ^AnyFile;

  TOpenProc   = procedure (var PrivateData; Mode: TOpenMode);
  TSelectFunc = function  (var PrivateData; Writing: Boolean):
  Integer;  { called before SelectHandle, must return a file handle
  }
  TSelectProc = procedure (var PrivateData; var ReadSelect,
  WriteSelect, ExceptSelect: Boolean);  { called before and after
  SelectHandle }
  TReadFunc   = function  (var PrivateData; var   Buffer; Size:
  SizeType): SizeType;
  TWriteFunc  = function  (var PrivateData; const Buffer; Size:
  SizeType): SizeType;
  TFileProc   = procedure (var PrivateData);
  TFlushProc  = TFileProc;
  TCloseProc  = TFileProc;
  TDoneProc   = TFileProc;

{ Flags that can be `or'ed into FileMode. The default value of
  FileMode is FileMode_Reset_ReadWrite. The somewhat confusing
  numeric values are meant to be compatible to BP (as far as
  BP supports them). }
const
  { Allow writing to binary files opened with Reset }
  FileMode_Reset_ReadWrite      = 2;

  { Do not allow reading from files opened with Rewrite }
  FileMode_Rewrite_WriteOnly    = 4;

  { Do not allow reading from files opened with Extend }
  FileMode_Extend_WriteOnly     = 8;

  { Allow writing to text files opened with Reset }
  FileMode_Text_Reset_ReadWrite = $100;

var
  FileMode: Integer; attribute (name = '_p_FileMode'); external;

{ Get the external name of a file }
function  FileName (protected var f: GPC_FDR): TString; attribute
  (name = '_p_FileName'); external;
```

```
procedure IOErrorFile (n: Integer; protected var f: GPC_FDR;
  ErrNoFlag: Boolean); attribute (iocritical, name
  = '_p_IOErrorFile'); external;

procedure GetBinding (protected var f: GPC_FDR; var b: BindingType);
  attribute (name = '_p_GetBinding'); external;
procedure ClearBinding (var b: BindingType); attribute (name
  = '_p_ClearBinding'); external;

{ TFDD interface @@ Subject to change! Use with caution! }
procedure AssignTFDD (var f: GPC_FDR;
                      aOpenProc:    TOpenProc;
                      aSelectFunc:  TSelectFunc;
                      aSelectProc:  TSelectProc;
                      aReadFunc:    TReadFunc;
                      aWriteFunc:   TWriteFunc;
                      aFlushProc:   TFlushProc;
                      aCloseProc:   TCloseProc;
                      aDoneProc:    TDoneProc;
                      aPrivateData: Pointer); attribute (name
  = '_p_AssignTFDD'); external;

procedure SetTFDD    (var f: GPC_FDR;
                      aOpenProc:    TOpenProc;
                      aSelectFunc:  TSelectFunc;
                      aSelectProc:  TSelectProc;
                      aReadFunc:    TReadFunc;
                      aWriteFunc:   TWriteFunc;
                      aFlushProc:   TFlushProc;
                      aCloseProc:   TCloseProc;
                      aDoneProc:    TDoneProc;
                      aPrivateData: Pointer); attribute (name
  = '_p_SetTFDD'); external;

{ Any parameter except f may be Null }
procedure GetTFDD    (var f: GPC_FDR;
                      var aOpenProc:    TOpenProc;
                      var aSelectFunc:  TSelectFunc;
                      var aSelectProc:  TSelectProc;
                      var aReadFunc:    TReadFunc;
                      var aWriteFunc:   TWriteFunc;
                      var aFlushProc:   TFlushProc;
                      var aCloseProc:   TCloseProc;
                      var aDoneProc:    TDoneProc;
                      var aPrivateData: Pointer); attribute (name
  = '_p_GetTFDD'); external;

procedure FileMove (var f: GPC_FDR; NewName: CString; Overwrite:
  Boolean); attribute (iocritical, name = '_p_FileMove'); external;
```

```
const
  NoChange = -1;  { can be passed to ChOwn for Owner and/or Group to
  not change that value }

procedure CloseFile (var f: GPC_FDR); attribute (name
  = '_p_CloseFile'); external;
procedure ChMod (var f: GPC_FDR; Mode: Integer); attribute
  (iocritical, name = '_p_ChMod'); external;
procedure ChOwn (var f: GPC_FDR; Owner, Group: Integer); attribute
  (iocritical, name = '_p_ChOwn'); external;

{ Checks if data are available to be read from f. This is
  similar to `not EOF (f)', but does not block on "files" that
  can grow, like Ttys or pipes. }
function  CanRead (var f: GPC_FDR): Boolean; attribute (name
  = '_p_CanRead'); external;

{ Checks if data can be written to f. }
function  CanWrite (var f: GPC_FDR): Boolean; attribute (name
  = '_p_CanWrite'); external;

{ Get the file handle. }
function  FileHandle (protected var f: GPC_FDR): Integer; attribute
  (name = '_p_FileHandle'); external;

{ Lock/unlock a file. }
function  FileLock (var f: GPC_FDR; WriteLock, Block: Boolean):
  Boolean; attribute (name = '_p_FileLock'); external;
function  FileUnlock (var f: GPC_FDR): Boolean; attribute (name
  = '_p_FileUnlock'); external;

{ Try to map (a part of) a file to memory. }
function  MemoryMap (Start: Pointer; Length: SizeType; Access:
  Integer; Shared: Boolean;
                     var f: GPC_FDR; Offset: FileSizeType): Pointer;
  attribute (name = '_p_MemoryMap'); external;

{ Unmap a previous memory mapping. }
procedure MemoryUnMap (Start: Pointer; Length: SizeType); attribute
  (name = '_p_MemoryUnMap'); external;

type
  Natural = 1 .. MaxInt;
  IOSelectEvents = (SelectReadOrEOF, SelectRead, SelectEOF,
  SelectWrite, SelectException, SelectAlways);

type
  IOSelectType = record
    f: PAnyFile;
    Wanted: set of IOSelectEvents;
    Occurred: set of Low (IOSelectEvents) .. Pred (SelectAlways)
```

```
  end;

{ Waits for one of several events to happen. Returns when one or
  more of the wanted events for one of the files occur. If they have
  already occurred before calling the function, it returns
  immediately. MicroSeconds can specify a timeout. If it is 0, the
  function will return immediately, whether or not an event has
  occurred. If it is negative, the function will wait forever until
  an event occurs. The Events parameter can be Null, in which case
  the function only waits for the timeout. If any of the file
  pointers (f) in Events are nil or the files pointed to are closed,
  they are simply ignored for convenience.

  It returns the index of one of the files for which any event has
  occurred. If events have occurred for several files, is it
  undefined which of these file's index is returned. If no event
  occurs until the timeout, 0 is returned. If an error occurs or the
  target system does not have a `select' system call and Events is
  not Null, a negative value is returned. In the Occurred field of
  the elements of Events, events that have occurred are set. The
  state of events not wanted is undefined.

  The possible events are:
  SelectReadOrEOF: the file is at EOF or data can be read now.
  SelectRead:      data can be read now.
  SelectEOF:       the file is at EOF.
  SelectWrite:     data can be written now.
  SelectException: an exception occurred on the file.
  SelectAlways:    if this is set, *all* requested events will be
                   checked for this file in any case. Otherwise,
                   checks may be skipped if already another event
                   for this or another file was found.

  Notes:
  Checking for EOF requires some reading ahead internally (just like
  the EOF function) which can be avoided by setting SelectReadOrEOF
  instead of SelectRead and SelectEOF. If this is followed by, e.g.,
  a BlockRead with 4 parameters, the last parameter will be 0 if and
  only the file is at EOF, and otherwise, data will be read directly
  from the file without reading ahead and buffering.

  SelectAlways should be set for files whose events are considered
  to be of higher priority than others. Otherwise, if one is
  interested in just any event, not setting SelectAlways may be a
  little faster. }
function  IOSelect (var Events: array [m .. n: Natural] of
  IOSelectType; MicroSeconds: MicroSecondTimeType): Integer;
  attribute (name = '_p_IOSelect'); external;

{ A simpler interface to SelectIO for the most common use. Waits for
  SelectReadOrEOF on all files and returns an index. }
```

```
function  IOSelectRead (const Files: array [m .. n: Natural] of
  PAnyFile; MicroSeconds: MicroSecondTimeType): Integer; attribute
  (name = '_p_IOSelectRead'); external;

{ Bind a filename to an external file }
procedure AssignFile   (var t: AnyFile; const FileName: String);
  attribute (name = '_p_AssignFile'); external;
procedure AssignBinary (var t: Text; const FileName: String);
  attribute (name = '_p_AssignBinary'); external;
procedure AssignHandle (var t: AnyFile; Handle: Integer; CloseFlag:
  Boolean); attribute (name = '_p_AssignHandle'); external;

{ Under development }
procedure AnyStringTFDD_Reset (var f: GPC_FDR; var Buf:
  ConstAnyString); attribute (name = '_p_AnyStringTFDD_Reset');
  external;
{ @@ procedure AnyStringTFDD_Rewrite (var f: GPC_FDR; var Buf:
  VarAnyString); attribute (name = '_p_AnyStringTFDD_Rewrite'); }
procedure StringTFDD_Reset (var f: GPC_FDR; var Buf: ConstAnyString;
  const s: String); attribute (name = '_p_StringTFDD_Reset');
  external;
{ @@ procedure StringTFDD_Rewrite (var f: GPC_FDR; var Buf:
  VarAnyString; var s: String); attribute (name
  = '_p_StringTFDD_Rewrite'); }

{ Returns True is a terminal device is open on the file f, False if
  f is not open or not connected to a terminal. }
function  IsTerminal (protected var f: GPC_FDR): Boolean; attribute
  (name = '_p_IsTerminal'); external;

{ Returns the file name of the terminal device that is open on the
  file f. Returns the empty string if (and only if) f is not open or
  not connected to a terminal. }
function  GetTerminalName (protected var f: GPC_FDR): TString;
  attribute (name = '_p_GetTerminalName'); external;

{ Command line option parsing, from getopt.pas }

const
  EndOfOptions      = #255;
  NoOption          = #1;
  UnknownOption     = '?';
  LongOption        = #0;
  UnknownLongOption = '?';

var
  FirstNonOption        : Integer; attribute (name
  = '_p_FirstNonOption'); external;
  HasOptionArgument     : Boolean; attribute (name
  = '_p_HasOptionArgument'); external;
  OptionArgument        : TString; attribute (name
```

```
  = '_p_OptionArgument'); external;
  UnknownOptionCharacter: Char; attribute (name
  = '_p_UnknownOptionCharacter'); external;
  GetOptErrorFlag       : Boolean; attribute (name
  = '_p_GetOptErrorFlag'); external;

{ Parses command line arguments for options and returns the next
  one.

  If a command line argument starts with `-', and is not exactly `-'
  or `--', then it is an option element. The characters of this
  element (aside from the initial `-') are option characters. If
  `GetOpt' is called repeatedly, it returns successively each of the
  option characters from each of the option elements.

  If `GetOpt' finds another option character, it returns that
  character, updating `FirstNonOption' and internal variables so
  that the next call to `GetOpt' can resume the scan with the
  following option character or command line argument.

  If there are no more option characters, `GetOpt' returns
  EndOfOptions. Then `FirstNonOption' is the index of the first
  command line argument that is not an option. (The command line
  arguments have been permuted so that those that are not options
  now come last.)

  OptString must be of the form `[+|-]abcd:e:f:g::h::i::'.

  a, b, c are options without arguments
  d, e, f are options with required arguments
  g, h, i are options with optional arguments

  Arguments are text following the option character in the same
  command line argument, or the text of the following command line
  argument. They are returned in OptionArgument. If an option has no
  argument, OptionArgument is empty. The variable HasOptionArgument
  tells whether an option has an argument. This is mostly useful for
  options with optional arguments, if one wants to distinguish an
  empty argument from no argument.

  If the first character of OptString is `+', GetOpt stops at the
  first non-option argument.

  If it is `-', GetOpt treats non-option arguments as options and
  return NoOption for them.

  Otherwise, GetOpt permutes arguments and handles all options,
  leaving all non-options at the end. However, if the environment
  variable POSIXLY_CORRECT is set, the default behaviour is to stop
  at the first non-option argument, as with `+'.
```

```
  The special argument ‘--’ forces an end of option-scanning
  regardless of the first character of OptString. In the case of
  ‘-’, only ‘--’ can cause GetOpt to return EndOfOptions with
  FirstNonOption <= ParamCount.

  If an option character is seen that is not listed in OptString,
  UnknownOption is returned. The unrecognized option character is
  stored in UnknownOptionCharacter. Unless GetOptErrorFlag is set to
  False, an error message is printed to StdErr automatically. }
function  GetOpt (const OptString: String): Char; attribute (name
  = '_p_GetOpt'); external;

type
  OptArgType = (NoArgument, RequiredArgument, OptionalArgument);

  OptionType = record
    OptionName: CString;
    Argument  : OptArgType;
    Flag      : ^Char;  { if nil, v is returned. Otherwise, Flag^ is
  ... }
    v         : Char    { ... set to v, and LongOption is returned }
  end;

{ Recognize short options, described by OptString as above, and long
  options, described by LongOptions.

  Long-named options begin with ‘--’ instead of ‘-’. Their names may
  be abbreviated as long as the abbreviation is unique or is an
  exact match for some defined option. If they have an argument, it
  follows the option name in the same argument, separated from the
  option name by a ‘=’, or else the in next argument. When GetOpt
  finds a long-named option, it returns LongOption if that option's
  ‘Flag’ field is non-nil, and the value of the option's ‘v’ field
  if the ‘Flag’ field is nil.

  LongIndex, if not Null, returns the index in LongOptions of the
  long-named option found. It is only valid when a long-named option
  has been found by the most recent call.

  If LongOnly is set, ‘-’ as well as ‘--’ can indicate a long
  option. If an option that starts with ‘-’ (not ‘--’) doesn't match
  a long option, but does match a short option, it is parsed as a
  short option instead. If an argument has the form ‘-f’, where f is
  a valid short option, don't consider it an abbreviated form of a
  long option that starts with ‘f’. Otherwise there would be no way
  to give the ‘-f’ short option. On the other hand, if there's a
  long option ‘fubar’ and the argument is ‘-fu’, do consider that an
  abbreviation of the long option, just like ‘--fu’, and not ‘-f’
  with argument ‘u’. This distinction seems to be the most useful
  approach.
```

```
  As an additional feature (not present in the C counterpart), if
  the last character of OptString is `-' (after a possible starting
  `+' or `-' character), or OptString is empty, all long options
  with a nil `Flag' field will automatically be recognized as short
  options with the character given by the `v' field. This means, in
  the common (and recommended) case that all short options have long
  equivalents, you can simply pass an empty OptString (or pass `+-'
  or `--' as OptString if you want this behaviour, see the comment
  for GetOpt), and you will only have to maintain the LongOptions
  array when you add or change options. }
function  GetOptLong (const OptString: String; const LongOptions:
  array [m .. n: Integer] of OptionType { can be Null };
                    var LongIndex: Integer { can be Null };
  LongOnly: Boolean): Char; attribute (name = '_p_GetOptLong');
  external;

{ Reset GetOpt's state and make the next GetOpt or GetOptLong start
  (again) with the StartArgument'th argument (may be 1). This is
  useful for special purposes only. It is *necessary* to do this
  after altering the contents of CParamCount/CParameters (which is
  not usually done, either). }
procedure ResetGetOpt (StartArgument: Integer); attribute (name
  = '_p_ResetGetOpt'); external;

{ Set operations, from sets.pas }

{ All set operations are built-in identifiers and not declared in
  gpc.pas. }

{ Heap management routines, from heap.pas }

{ GPC implements both Mark/Release and Dispose. Both can be mixed
  freely in the same program. Dispose should be preferred, since
  it's faster. }

{ C heap management routines. NOTE: if Release is used anywhere in
  the program, CFreeMem and CReAllocMem may not be used for pointers
  that were not allocated with CGetMem. }
function  CGetMem     (Size: SizeType): Pointer; external
  name 'malloc';
procedure CFreeMem    (aPointer: Pointer); external name 'free';
function  CReAllocMem (aPointer: Pointer; NewSize: SizeType):
  Pointer; external name 'realloc';

type
  GetMemType     = ^function (Size: SizeType): Pointer;
  FreeMemType    = ^procedure (aPointer: Pointer);
  ReAllocMemType = ^function (aPointer: Pointer; NewSize: SizeType):
  Pointer;

{ These variables can be set to user-defined routines for memory
```

```
  allocation/deallocation. GetMemPtr may return nil when
  insufficient memory is available. GetMem/New will produce a
  runtime error then. }
var
  GetMemPtr    : GetMemType; attribute (name = '_p_GetMemPtr');
  external;
  FreeMemPtr   : FreeMemType; attribute (name = '_p_FreeMemPtr');
  external;
  ReAllocMemPtr: ReAllocMemType; attribute (name
  = '_p_ReAllocMemPtr'); external;

  { Address of the lowest byte of heap used }
  HeapLow: PtrCard; attribute (name = '_p_HeapLow'); external;

  { Address of the highest byte of heap used }
  HeapHigh: PtrCard; attribute (name = '_p_HeapHigh'); external;

  { If set to true, `Dispose' etc. will raise a runtime error if
    given an invalid pointer. }
  HeapChecking: Boolean; attribute (name = '_p_HeapChecking');
  external;

const
  UndocumentedReturnNil = Pointer (-1);

function  SuspendMark: Pointer; attribute (name = '_p_SuspendMark');
  external;
procedure ResumeMark (p: Pointer); attribute (name
  = '_p_ResumeMark'); external;

{ Calls the procedure Proc for each block that would be released
  with `Release (aMark)'. aMark must have been marked with Mark. For
  an example of its usage, see the HeapMon unit. }
procedure ForEachMarkedBlock (aMark: Pointer; procedure Proc
  (aPointer: Pointer; aSize: SizeType; aCaller: Pointer)); attribute
  (name = '_p_ForEachMarkedBlock'); external;

procedure ReAllocMem (var aPointer: Pointer; NewSize: SizeType);
  attribute (name = '_p_ReAllocMem'); external;

{ Memory transfer procedures, from move.pas }

{ The move operations are built-in identifiers and not declared in
  gpc.pas. }

{ Routines to handle endianness, from endian.pas }

{ Boolean constants about endianness and alignment }

const
  BitsBigEndian  = {$ifdef __BITS_LITTLE_ENDIAN__}
```

```
                   False
                   {$elif defined (__BITS_BIG_ENDIAN__)}
                   True
                   {$else}
                   {$error Bit endianness is not defined!}
                   {$endif};

  BytesBigEndian = {$ifdef __BYTES_LITTLE_ENDIAN__}
                   False
                   {$elif defined (__BYTES_BIG_ENDIAN__)}
                   True
                   {$else}
                   {$error Byte endianness is not defined!}
                   {$endif};

  WordsBigEndian = {$ifdef __WORDS_LITTLE_ENDIAN__}
                   False
                   {$elif defined (__WORDS_BIG_ENDIAN__)}
                   True
                   {$else}
                   {$error Word endianness is not defined!}
                   {$endif};

  NeedAlignment  = {$ifdef __NEED_ALIGNMENT__}
                   True
                   {$elif defined (__NEED_NO_ALIGNMENT__)}
                   False
                   {$else}
                   {$error Alignment is not defined!}
                   {$endif};

{ Convert single variables from or to little or big endian format.
  This only works for a single variable or a plain array of a simple
  type. For more complicated structures, this has to be done for
  each component separately! Currently, ConvertFromFooEndian and
  ConvertToFooEndian are the same, but this might not be the case on
  middle-endian machines. Therefore, we provide different names. }
procedure ReverseBytes            (var Buf; ElementSize, Count:
  SizeType); attribute (name = '_p_ReverseBytes'); external;
procedure ConvertFromLittleEndian (var Buf; ElementSize, Count:
  SizeType); attribute (name = '_p_ConvertLittleEndian'); external;
procedure ConvertFromBigEndian    (var Buf; ElementSize, Count:
  SizeType); attribute (name = '_p_ConvertBigEndian'); external;
procedure ConvertToLittleEndian   (var Buf; ElementSize, Count:
  SizeType); external name '_p_ConvertLittleEndian';
procedure ConvertToBigEndian      (var Buf; ElementSize, Count:
  SizeType); external name '_p_ConvertBigEndian';

{ Read a block from a file and convert it from little or
  big endian format. This only works for a single variable or a
  plain array of a simple type, note the comment for
```

```
  'ConvertFromLittleEndian' and 'ConvertFromBigEndian'. }
procedure BlockReadLittleEndian   (var aFile: File; var   Buf;
  ElementSize, Count: SizeType); attribute (iocritical, name
  = '_p_BlockRead_LittleEndian'); external;
procedure BlockReadBigEndian      (var aFile: File; var   Buf;
  ElementSize, Count: SizeType); attribute (iocritical, name
  = '_p_BlockRead_BigEndian'); external;

{ Write a block variable to a file and convert it to little or big
  endian format before. This only works for a single variable or a
  plain array of a simple type. Apart from this, note the comment
  for 'ConvertToLittleEndian' and 'ConvertToBigEndian'. }
procedure BlockWriteLittleEndian  (var aFile: File; const Buf;
  ElementSize, Count: SizeType); attribute (iocritical, name
  = '_p_BlockWrite_LittleEndian'); external;
procedure BlockWriteBigEndian     (var aFile: File; const Buf;
  ElementSize, Count: SizeType); attribute (iocritical, name
  = '_p_BlockWrite_BigEndian'); external;

{ Read and write strings from/to binary files, where the length is
  stored in the given endianness and with a fixed size (64 bits),
  and therefore is independent of the system. }
procedure ReadStringLittleEndian  (var f: File; var s: String);
  attribute (iocritical, name = '_p_ReadStringLittleEndian');
  external;
procedure ReadStringBigEndian     (var f: File; var s: String);
  attribute (iocritical, name = '_p_ReadStringBigEndian'); external;
procedure WriteStringLittleEndian (var f: File; const s: String);
  attribute (iocritical, name = '_p_WriteStringLittleEndian');
  external;
procedure WriteStringBigEndian    (var f: File; const s: String);
  attribute (iocritical, name = '_p_WriteStringBigEndian');
  external;

{ Initialization, from init.pas }

var
  InitProc: ^procedure; attribute (name = '_p_InitProc'); external;
```

6.15 Units included with GPC

GPC distributions now include a number of useful Pascal units and a complete set of BP compatibility units – except for the '`Graph`' unit (which is currently distributed separately due to its license) and the OOP stuff. The main use of these units is to provide a way to port BP programs to GPC as easily as possible. Some of the units also implement functionaliy not available otherwise.

Most of the BP compatibility units – except '`CRT`' and '`Printer`' – are merely meant to let programs written for BP compile with GPC as easily as possible. They should not be used in newly written code, and for code ported from BP to GPC, it is suggested to replace them successively with the more powerful – and often easier to use – alternatives that GPC's Run Time System (see Section 6.14 [Run Time System], page 105) offers.

The following sections describe all units included with GPC (besides the 'GPC' module which describes the interface to the Run Time System, Section 6.14 [Run Time System], page 105).

6.15.1 BP compatibility: CRT & WinCRT, portable, with many extensions

The following listing contains the interface of the CRT unit.

'CRT' is a 'curses' based unit for text screen handling. It is compatible to BP's 'CRT' unit, even in a lot of minor details like the values of function key codes and includes some routines for compatibility with TP5's 'Win' unit as well as BP's 'WinCRT' and Turbo Power's 'TPCrt' units, and some extensions.

The unit has been extended by many functions that were lacking in BP's unit and required assembler code or direct memory/port access to be implemented under BP. The GPC version is now fully suited for portable, real-world programming without any dirty tricks.

The unit is also available as 'WinCRT', completely identical to 'CRT'. The only purpose of this "feature" is to let programs written for TPW or BP, with a 'uses WinCRT' directive, compile without changes. Unlike TPW/BP's 'WinCRT' unit, GPC's unit is not crippled, compared to 'CRT'.

To use this unit, you will need the 'ncurses' (version 5.0 or newer) or 'PDCurses' library which can be found in `http://www.gnu-pascal.de/libs/`.

```
{ CRT (Crt Replacement Tool)
  Portable BP compatible CRT unit for GPC with many extensions

  This unit is aware of terminal types. This means programs using
  this unit will work whether run locally or while being logged in
  remotely from a system with a completely different terminal type
  (as long as the appropriate terminfo entry is present on the
  system where the program is run).

  NOTES:

  - The CRT unit needs the ncurses and panel libraries which should
    be available for almost any system. For Dos systems, where
    ncurses is not available, it is configured to use the PDCurses
    and its panel library instead. On Unix systems with X11, it can
    also use PDCurses (xcurses) and xpanel to produce X11 programs.
    The advantage is that the program won't need an xterm with a
    valid terminfo entry, the output may look a little nicer and
    function keys work better than in an xterm, but the disadvantage
    is that it will only run under X. The ncurses and PDCurses
    libraries (including panel and xpanel, resp.) can be found in
    http://www.gnu-pascal.de/libs/
    (Note that ncurses is already installed on many Unix systems.)
    For ncurses, version 5.0 or newer is required.

    When an X11 version under Unix is wanted, give '-DX11' when
    compiling crt.pas and crtc.c (or when compiling crt.pas or a
    program that uses CRT with '--automake'). On pre-X11R6 systems,
    give '-DNOX11R6' additionally. You might also have to give the
    path to the X11 libraries with '-L', e.g. '-L /usr/X11/lib'.
```

```
- A few features cannot be implemented in a portable way and are
  only available on some systems:

    Sound, NoSound 1)                 -----------------------.
    GetShiftState                     ------------------.    |
    TextMode etc. 2)                  -------------.    |    |
    CRTSavePreviousScreen             --------.    |    |    |
    Interrupt signal (Ctrl-C) handling ---.    |    |    |    |
                                         |    |    |    |    |
  Linux/IA32 3) (terminal)               X    X 4) X 5) X 6) X 6)
  Other Unix (terminal)                  X    X 7) X 5) -    -
  Unix (X11 version)                     X    X    -    X    -
  Dos (DJGPP)                            X    X    X    X    X
  MS-Windows (Cygwin, mingw, MSYS)       X    -    X 8) X    -

  Notes:

  1) If you define NO_CRT_DUMMY_SOUND while compiling CRT, you
     will get linking errors when your program tries to use
     Sound/NoSound on a platform where it's not supported (which
     is useful to detect at compile time if playing sound is a
     major task of your program). Otherwise, Sound/NoSound will
     simply do nothing (which is usually acceptable if the program
     uses these routines just for an occasional beep).

  2) Changing to monochrome modes works on all platforms. Changing
     the screen size only works on those indicated. However, even
     on the platforms not supported, the program will react to
     screen size changes by external means (e.g. changing the
     window size with the mouse if running in a GUI window or
     resizing a console or virtual terminal).

  3) Probably also on other processors, but I've had no chance to
     test this yet.

  4) Only on a local console with access permissions to the
     corresponding virtual console memory device or using the
     `crtscreen' utility (see crtscreen.c in the demos directory).

  5) Only if supported by an external command (e.g., in xterms and
     on local Linux consoles). The command to be called can be
     defined in the environment variable `RESIZETERM' (where the
     variables `columns' and `lines' in the command are set to the
     size wanted). If not set, the code will try `resize -s' in an
     xterm and otherwise `SVGATextMode' and `setfont'. For this to
     work, these utilities need to be present in the PATH or
     `/usr/sbin' or `/usr/local/sbin'. Furthermore, SVGATextMode
     and setfont require root permissions, either to the
     executable of the program compiled with CRT or to resizecons
     (called by setfont) or SVGATextMode. To allow the latter, do
     "chmod u+s `which resizecons`" and/or
```

```
     "chmod u+s `which SVGATextMode`", as root once, but only if
     you really want each user to be allowed to change the text
     mode.

  6) Only on local consoles.

  7) Some terminals only. Most xterms etc. support it as well as
     other terminals that support an "alternate screen" in the
     smcup/rmcup terminal capabilities.

  8) Only with PDCurses, not with ncurses. Changing the number of
     screen *columns* doesn't work in a full-screen session.

- When CRT is initialized (automatically or explicitly; see the
  comments for CRTInit), the screen is cleared, and at the end of
  the program, the cursor is placed at the bottom of the screen
  (curses behaviour).

- All the other things (including most details like color and
  function key constants) are compatible with BP's CRT unit, and
  there are many extensions that BP's unit does not have.

- When the screen size is changed by an external event (e.g.,
  resizing an xterm or changing the screen size from another VC
  under Linux), the virtual "function key" kbScreenSizeChanged is
  returned. Applications can use the virtual key to resize their
  windows. kbScreenSizeChanged will not be returned if the screen
  size change was initiated by the program itself (by using
  TextMode or SetScreenSize). Note that TextMode sets the current
  panel to the full screen size, sets the text attribute to the
  default and clears the window (BP compatibility), while
  SetScreenSize does not.

- After the screen size has been changed, whether by using
  TextMode, SetScreenSize or by an external event, ScreenSize will
  return the new screen size. The current window and all panels
  will have been adjusted to the new screen size. This means, if
  their right or lower ends are outside the new screen size, the
  windows are moved to the left and/or top as far as necessary. If
  this is not enough, i.e., if they are wider/higher than the new
  screen size, they are shrinked to the total screen width/height.
  When the screen size is enlarged, window sizes are not changed,
  with one exception: Windows that extend through the whole screen
  width/height are enlarged to the whole new screen width/height
  (in particular, full-screen windows remain full-screen). This
  behaviour might not be optimal for all purposes, but you can
  always resize your windows in your application after the screen
  size change.

- (ncurses only) The environment variable `ESCDELAY' specifies the
  number of milliseconds allowed between an `Esc' character and
```

```
  the rest of an escape sequence (default 1000). Setting it to a
  value too small can cause problems with programs not recognizing
  escape sequences such as function keys, especially over slow
  network connections. Setting it to a value too large can delay
  the recognition of an `ESC' key press notably. On local Linux
  consoles, e.g., 10 seems to be a good value.

- When trying to write portable programs, don't rely on exactly
  the same look of your output and the availability of all the key
  combinations. Some kinds of terminals support only some of the
  display attributes and special characters, and usually not all
  of the keys declared are really available. Therefore, it's safer
  to provide the same function on different key combinations and
  to not use the more exotic ones.

- CRT supports an additional modifier key (if present), called
  `Extra'. On DJGPP, it's the <Scroll Lock> key, under X11 it's
  the modifier #4, and on a local Linux console, it's the `CtrlL'
  modifier (value 64) which is unused on many keytabs and can be
  mapped to any key(s), e.g. to those keys on new keyboards with
  these ugly symbols waiting to be replaced by penguins (keycodes
  125 and 127) by inserting the following two lines into your
  /etc/default.keytab and reloading the keytab with `loadkeys'
  (you usually have to do this as root):

  keycode 125 = CtrlL
  keycode 127 = CtrlL

Copyright (C) 1998-2005 Free Software Foundation, Inc.

Author: Frank Heckenbach <frank@pascal.gnu.de>

This file is part of GNU Pascal.

GNU Pascal is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published
by the Free Software Foundation; either version 2, or (at your
option) any later version.

GNU Pascal is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU Pascal; see the file COPYING. If not, write to the
Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

As a special exception, if you link this file with files compiled
with a GNU compiler to produce an executable, this does not cause
```

```
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License.

  Please also note the license of the curses library used. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030722}
{$error This unit requires GPC release 20030722 or newer.}
{$endif}

unit {$ifdef THIS_IS_WINCRT} WinCRT {$else} CRT {$endif};

interface

uses GPC;

const
  { CRT modes }
  BW40          = 0;            { 40x25 Black/White }
  CO40          = 1;            { 40x25 Color }
  BW80          = 2;            { 80x25 Black/White }
  CO80          = 3;            { 80x25 Color }
  Mono          = 7;            { 80x25 Black/White }
  Font8x8       = 256;          { Add-in for 80x43 or 80x50 mode }

  { Mode constants for Turbo Pascal 3.0 compatibility }
  C40           = CO40;
  C80           = CO80;

  { Foreground and background color constants }
  Black         = 0;
  Blue          = 1;
  Green         = 2;
  Cyan          = 3;
  Red           = 4;
  Magenta       = 5;
  Brown         = 6;
  LightGray     = 7;

  { Foreground color constants }
  DarkGray      = 8;
  LightBlue     = 9;
  LightGreen    = 10;
  LightCyan     = 11;
  LightRed      = 12;
  LightMagenta  = 13;
  Yellow        = 14;
  White         = 15;
```

```
  { Add-in for blinking }
  Blink         = 128;

type
  TTextAttr = Byte;

var
  { If False (default: True), catch interrupt signals (SIGINT;
    Ctrl-C), and other flow control characters as well as SIGTERM,
    SIGHUP and perhaps other signals }
  CheckBreak: Boolean = True; attribute (name = 'crt_CheckBreak');

  { If True (default : False), replace Ctrl-Z by #0 in input }
  CheckEOF: Boolean = False; attribute (name = 'crt_CheckEOF');

  { Ignored -- meaningless here }
  DirectVideo: Boolean = True;

  { Ignored -- curses or the terminal driver will take care of that
    when necessary }
  CheckSnow: Boolean = False;

  { Current (sic!) text mode }
  LastMode: CCardinal = 3; attribute (name = 'crt_LastMode');

  { Current text attribute }
  TextAttr: TTextAttr = 7; attribute (name = 'crt_TextAttr');

  { Window upper left coordinates. *Obsolete*! Please see WindowMin
    below. }
  WindMin: CCardinal = High (CCardinal); attribute (name
  = 'crt_WindMin');

  { Window lower right coordinates. *Obsolete*! Please see WindowMax
    below. }
  WindMax: CCardinal = High (CCardinal); attribute (name
  = 'crt_WindMax');

procedure AssignCRT (var f: Text);
function  KeyPressed: Boolean; external name 'crt_KeyPressed';
function  ReadKey: Char; external name 'crt_ReadKey';

{ Not effective on all platforms, see above. See also SetScreenSize
  and SetMonochrome. }
procedure TextMode (Mode: Integer);

procedure Window (x1, y1, x2, y2: CInteger); external
  name 'crt_Window';
procedure GotoXY (x, y: CInteger); external name 'crt_GotoXY';
function  WhereX: CInteger; external name 'crt_WhereX';
function  WhereY: CInteger; external name 'crt_WhereY';
```

```
procedure ClrScr; external name 'crt_ClrScr';
procedure ClrEOL; external name 'crt_ClrEOL';
procedure InsLine; external name 'crt_InsLine';
procedure DelLine; external name 'crt_DelLine';
procedure TextColor (Color: TTextAttr);
procedure TextBackground (Color: TTextAttr);
procedure LowVideo;
procedure HighVideo;
procedure NormVideo;
procedure Delay (MS: CCardinal); external name 'crt_Delay';

{ Not available on all platforms, see above }
procedure Sound (Hz: CCardinal); external name 'crt_Sound';
procedure NoSound; external name 'crt_NoSound';

{ =================== Extensions over BP's CRT =================== }

{ Initializes the CRT unit. Should be called before using any of
  CRT's routines.

  Note: For BP compatibility, CRT is initizalized automatically when
  (almost) any of its routines are used for the first time. In this
  case, some defaults are set to match BP more closely. In
  particular, the PC charset (see SetPCCharSet) is enabled then
  (disabled otherwise), and the update level (see SetCRTUpdate) is
  set to UpdateRegularly (UpdateWaitInput otherwise). This feature
  is meant for BP compatibility *only*. Don't rely on it when
  writing a new program. Use CRTInit then, and set the defaults to
  the values you want explicitly.

  SetCRTUpdate is one of those few routines which will not cause CRT
  to be initialized immediately, and a value set with it will
  survive both automatic and explicit initialization, so you can use
  it to set the update level without caring which way CRT will be
  initialized. (This does not apply to SetPCCharSet. Since it works
  on a per-panel basis, it has to initialize CRT first, so there is
  a panel to start with.)

  If you terminate the program before calling CRTInit or any routine
  that causes automatic initialization, curses will never be
  initialized, so e.g., the screen won't be cleared. This can be
  useful, e.g., to check the command line arguments (or anything
  else) and if there's a problem, write an error and abort. Just be
  sure to write the error to StdErr, not Output (because Output will
  be assigned to CRT, and therefore writing to Output will cause CRT
  to be initialized, and because errors belong to StdErr, anyway),
  and to call `RestoreTerminal (True)' before (just to be sure, in
  case some code -- perhaps added later, or hidden in the
  initialization of some unit -- does initialize CRT). }
procedure CRTInit; external name 'crt_Init';
```

```
{ Changes the input and output file and the terminal description CRT
  uses. Only effective with ncurses, and only if called before CRT
  is initialized (automatically or explicitly; see the comments for
  CRTInit). If TerminalType is nil, the default will be used. If
  InputFile and/or OutputFile are Null, they remain unchanged. }
procedure CRTSetTerminal (TerminalType: CString; var InputFile,
  OutputFile: AnyFile); attribute (name = 'crt_SetTerminal');

{ If called with an argument True, it causes CRT to save the
  previous screen contents if possible (see the comments at the
  beginning of the unit), and restore them when calling
  RestoreTerminal (True). After RestoreTerminal (False), they're
  saved again, and at the end of the program, they're restored. If
  called with an argument False, it will prohibit this behaviour.
  The default, if this procedure is not called, depends on the
  terminal (generally it is active on most xterms and similar and
  not active on most other terminals).

  This procedure should be called before initializing CRT (using
  CRTInit or automatically), otherwise the previous screen contents
  may already have been overwritten. It has no effect under XCurses,
  because the program uses its own window, anyway. }
procedure CRTSavePreviousScreen (On: Boolean); external
  name 'crt_SavePreviousScreen';

{ Returns True if CRTSavePreviousScreen was called with argument
  True and the functionality is really available. Note that the
  result is not reliable until CRT is initialized, while
  CRTSavePreviousScreen should be called before CRT is initialized.
  That's why they are two separate routines. }
function  CRTSavePreviousScreenWorks: Boolean; external
  name 'crt_SavePreviousScreenWorks';

{ If CRT is initialized automatically, not via CRTInit, and
  CRTAutoInitProc is not nil, it will be called before actually
  initializing CRT. }
var
  CRTAutoInitProc: procedure = nil; attribute (name
  = 'crt_AutoInitProc');

{ Aborts with a runtime error saying that CRT was not initialized.
  If you set CRTAutoInitProc to this procedure, you can effectively
  disable CRT's automatic initialization. }
procedure CRTNotInitialized; attribute (name
  = 'crt_NotInitialized');

{ Set terminal to shell or curses mode. An internal procedure
  registered by CRT via RegisterRestoreTerminal does this as well,
  so CRTSetCursesMode has to be called only in unusual situations,
  e.g. after executing a process that changes terminal modes, but
  does not restore them (e.g. because it crashed or was killed), and
```

```
  the process was not executed with the Execute routine, and
  RestoreTerminal was not called otherwise. If you set it to False
  temporarily, be sure to set it back to True before doing any
  further CRT operations, otherwise the result may be strange. }
procedure CRTSetCursesMode (On: Boolean); external
  name 'crt_SetCursesMode';

{ Do the same as `RestoreTerminal (True)', but also clear the screen
  after restoring the terminal (except for XCurses, because the
  program uses its own window, anyway). Does not restore and save
  again the previous screen contents if CRTSavePreviousScreen was
  called. }
procedure RestoreTerminalClearCRT; attribute (name
  = 'crt_RestoreTerminalClearCRT');

{ Keyboard and character graphics constants -- BP compatible! =:-}
{$i crt.inc}

var
  { Tells whether the XCurses version of CRT is used }
  XCRT: Boolean = {$ifdef XCURSES} True {$else} False {$endif};
  attribute (name = 'crt_XCRT');

  { If True (default: False), the Beep procedure and writing #7 do a
    Flash instead }
  VisualBell: Boolean = False; attribute (name = 'crt_VisualBell');

  { Cursor shape codes. Only to be used in very special cases. }
  CursorShapeHidden: CInteger = 0; attribute (name
  = 'crt_CursorShapeHidden');
  CursorShapeNormal: CInteger = 1; attribute (name
  = 'crt_CursorShapeNormal');
  CursorShapeFull:   CInteger = 2; attribute (name
  = 'crt_CursorShapeFull');

type
  TKey = CCardinal;

  TCursorShape = (CursorIgnored, CursorHidden, CursorNormal,
  CursorFat, CursorBlock);

  TCRTUpdate = (UpdateNever, UpdateWaitInput, UpdateInput,
                UpdateRegularly, UpdateAlways);

  TPoint = record
    x, y: CInteger
  end;

  PCharAttr = ^TCharAttr;
  TCharAttr = record
    ch       : Char;
```

```
    Attr     : TTextAttr;
    PCCharSet: Boolean
  end;

  PCharAttrs = ^TCharAttrs;
  TCharAttrs = array [1 .. MaxVarSize div SizeOf (TCharAttr)] of
  TCharAttr;

  TWindowXYInternalCard8 = Cardinal attribute (Size = 8);
  TWindowXYInternalFill = Integer attribute (Size = BitSizeOf
  (CCardinal) - 16);
  TWindowXY = packed record
    {$ifdef __BYTES_BIG_ENDIAN__}
    Fill: TWindowXYInternalFill;
    y, x: TWindowXYInternalCard8
    {$elif defined (__BYTES_LITTLE_ENDIAN__)}
    x, y: TWindowXYInternalCard8;
    Fill: TWindowXYInternalFill
    {$else}
    {$error Endianness is not defined!}
    {$endif}
  end;

{ Make sure TWindowXY really has the same size as WindMin and
  WindMax. The value of the constant will always be True, and is of
  no further interest. }
const
  AssertTWindowXYSize = CompilerAssert ((SizeOf (TWindowXY) = SizeOf
  (WindMin)) and
                                        (SizeOf (TWindowXY) = SizeOf
  (WindMax)));

var
  { Window upper and left coordinates. More comfortable to access
    than WindMin, but also *obsolete*. WindMin and WindowMin still
    work, but have the problem that they implicitly limit the window
    size to 255x255 characters. Though that's not really small for a
    text window, it's easily possible to create bigger ones (e.g. in
    an xterm with a small font, on a high resolution screen and/or
    extending over several virutal desktops). When using coordinates
    greater than 254, the corresponding bytes in WindowMin/WindowMax
    will be set to 254, so, e.g., programs which do
    `Inc (WindowMin.x)' will not fail quite as badly (but probably
    still fail). The routines Window and GetWindow use Integer
    coordinates, and don't suffer from any of these problems, so
    they should be used instead. }
  WindowMin: TWindowXY absolute WindMin;

  { Window lower right coordinates. More comfortable to access than
    WindMax, but also *obsolete* (see the comments for WindowMin).
    Use Window and GetWindow instead. }
```

```
  WindowMax: TWindowXY absolute WindMax;

  { The attribute set by NormVideo }
  NormAttr: TTextAttr = 7; attribute (name = 'crt_NormAttr');

  { Tells whether the current mode is monochrome }
  IsMonochrome: Boolean = False; attribute (name
  = 'crt_IsMonochrome');

  { This value can be set to a combination of the shFoo constants
    and will be ORed to the actual shift state returned by
    GetShiftState. This can be used to easily simulate shift keys on
    systems where they can't be accessed. }
  VirtualShiftState: CInteger = 0; attribute (name
  = 'crt_VirtualShiftState');

{ Returns the size of the screen. Note: In BP's WinCRT unit,
  ScreenSize is a variable. But since writing to it from a program
  is pointless, anyway, providing a function here should not cause
  any incompatibility. }
function  ScreenSize: TPoint; attribute (name
  = 'crt_GetScreenSize');

{ Change the screen size if possible. }
procedure SetScreenSize (x, y: CInteger); external
  name 'crt_SetScreenSize';

{ Turns colors off or on. }
procedure SetMonochrome (Monochrome: Boolean); external
  name 'crt_SetMonochrome';

{ Tell which modifier keys are currently pressed. The result is a
  combination of the shFoo constants defined in crt.inc, or 0 on
  systems where this function is not supported -- but note
  VirtualShiftState. If supported, ReadKey automatically converts
  kbIns and kbDel keys to kbShIns and kbShDel, resp., if shift is
  pressed. }
function  GetShiftState: CInteger; external
  name 'crt_GetShiftState';

{ Get the extent of the current window. Use this procedure rather
  than reading WindMin and WindMax or WindowMin and WindowMax, since
  this routine allows for window sizes larger than 255. The
  resulting coordinates are 1-based (like in Window, unlike WindMin,
  WindMax, WindowMin and WindowMax). Any of the parameters may be
  Null in case you're interested in only some of the coordinates. }
procedure GetWindow (var x1, y1, x2, y2: Integer); attribute (name
  = 'crt_GetWindow');

{ Determine when to update the screen. The possible values are the
  following. The given conditions *guarantee* updates. However,
```

```
  updates may occur more frequently (even if the update level is set
  to UpdateNever). About the default value, see the comments for
  CRTInit.

  UpdateNever    : never (unless explicitly requested with
                   CRTUpdate)
  UpdateWaitInput: before Delay and CRT input, unless typeahead is
                   detected
  UpdateInput    : before Delay and CRT input
  UpdateRegularly: before Delay and CRT input and otherwise in
                   regular intervals without causing too much
                   refresh. This uses a timer on some systems
                   (currently, Unix with ncurses). This was created
                   for BP compatibility, but for many applications,
                   a lower value causes less flickering in the
                   output, and additionally, timer signals won't
                   disturb other operations. Under DJGPP, this
                   always updates immediately, but this fact should
                   not mislead DJGPP users into thinking this is
                   always so.
  UpdateAlways   : after each output. This can be very slow. (Not so
                   under DJGPP, but this fact should not mislead
                   DJGPP users ...) }
procedure SetCRTUpdate (UpdateLevel: TCRTUpdate); external
  name 'crt_SetUpdateLevel';

{ Do an update now, independently of the update level }
procedure CRTUpdate; external name 'crt_Update';

{ Do an update now and completely redraw the screen }
procedure CRTRedraw; external name 'crt_Redraw';

{ Return Ord (key) for normal keys and $100 * Ord (fkey) for
  function keys }
function  ReadKeyWord: TKey; external name 'crt_ReadKeyWord';

{ Extract the character and scan code from a TKey value }
function  Key2Char (k: TKey): Char;
function  Key2Scan (k: TKey): Char;

{ Convert a key to upper/lower case if it is a letter, leave it
  unchanged otherwise }
function  UpCaseKey (k: TKey): TKey;
function  LoCaseKey (k: TKey): TKey;

{ Return key codes for the combination of the given key with Ctrl,
  Alt, AltGr or Extra, resp. Returns 0 if the combination is
  unknown. }
function  CtrlKey  (ch: Char): TKey; attribute (name
  = 'crt_CtrlKey');
function  AltKey   (ch: Char): TKey; external name 'crt_AltKey';
```

```
function  AltGrKey (ch: Char): TKey; external name 'crt_AltGrKey';
function  ExtraKey (ch: Char): TKey; external name 'crt_ExtraKey';

{ Check if k is a pseudo key generated by a deadly signal trapped }
function  IsDeadlySignal (k: TKey): Boolean;

{ Produce a beep or a screen flash }
procedure Beep; external name 'crt_Beep';
procedure Flash; external name 'crt_Flash';

{ Get size of current window (calculated using GetWindow) }
function  GetXMax: Integer;
function  GetYMax: Integer;

{ Get/goto an absolute position }
function  WhereXAbs: Integer;
function  WhereYAbs: Integer;
procedure GotoXYAbs (x, y: Integer);

{ Turn scrolling on or off }
procedure SetScroll (State: Boolean); external name 'crt_SetScroll';

{ Read back whether scrolling is enabled }
function  GetScroll: Boolean; external name 'crt_GetScroll';

{ Determine whether to interpret non-ASCII characters as PC ROM
  characters (True), or in a system dependent way (False). About the
  default, see the comments for CRTInit. }
procedure SetPCCharSet (PCCharSet: Boolean); external
  name 'crt_SetPCCharSet';

{ Read back the value set by SetPCCharSet }
function  GetPCCharSet: Boolean; external name 'crt_GetPCCharSet';

{ Determine whether to interpret #7, #8, #10, #13 as control
  characters (True, default), or as graphics characters (False) }
procedure SetControlChars (UseControlChars: Boolean); external
  name 'crt_SetControlChars';

{ Read back the value set by SetControlChars }
function  GetControlChars: Boolean; external
  name 'crt_GetControlChars';

procedure SetCursorShape (Shape: TCursorShape); external
  name 'crt_SetCursorShape';
function  GetCursorShape: TCursorShape; external
  name 'crt_GetCursorShape';

procedure HideCursor;
procedure HiddenCursor;
procedure NormalCursor;
```

```
procedure FatCursor;
procedure BlockCursor;
procedure IgnoreCursor;

{ Simulates a block cursor by writing a block character onto the
  cursor position. The procedure automatically finds the topmost
  visible panel whose shape is not CursorIgnored and places the
  simulated cursor there (just like the hardware cursor), with
  matching attributes, if the cursor shape is CursorFat or
  CursorBlock (otherwise, no simulated cursor is shown).

  Calling this procedure again makes the simulated cursor disappear.
  In particular, to get the effect of a blinking cursor, you have to
  call the procedure repeatedly (say, 8 times a second). CRT will
  not do this for you, since it does not intend to be your main
  event loop. }
procedure SimulateBlockCursor; external
  name 'crt_SimulateBlockCursor';

{ Makes the cursor simulated by SimulateBlockCursor disappear if it
  is active. Does nothing otherwise. You should call this procedure
  after using SimulateBlockCursor before doing any further CRT
  output (though failing to do so should not hurt except for
  possibly leaving the simulated cursor in its old position longer
  than it should). }
procedure SimulateBlockCursorOff; external
  name 'crt_SimulateBlockCursorOff';

function  GetTextColor: Integer;
function  GetTextBackground: Integer;

{ Write string at the given position without moving the cursor.
  Truncated at the right margin. }
procedure WriteStrAt (x, y: Integer; const s: String; Attr:
  TTextAttr);

{ Write (several copies of) a char at then given position without
  moving the cursor. Truncated at the right margin. }
procedure WriteCharAt (x, y, Count: Integer; ch: Char; Attr:
  TTextAttr);

{ Write characters with specified attributes at the given position
  without moving the cursor. Truncated at the right margin. }
procedure WriteCharAttrAt (x, y, Count: CInteger; CharAttr:
  PCharAttrs); external name 'crt_WriteCharAttrAt';

{ Write a char while moving the cursor }
procedure WriteChar (ch: Char);

{ Read a character from a screen position }
procedure ReadChar (x, y: CInteger; var ch: Char; var Attr:
```

```
  TTextAttr); external name 'crt_ReadChar';

{ Change only text attributes, leave characters. Truncated at the
  right margin. }
procedure ChangeTextAttr (x, y, Count: Integer; NewAttr: TTextAttr);

{ Fill current window }
procedure FillWin (ch: Char; Attr: TTextAttr); external
  name 'crt_FillWin';

{ Calculate size of memory required for ReadWin in current window. }
function  WinSize: SizeType; external name 'crt_WinSize';

{ Save window contents. Buf must be WinSize bytes large. }
procedure ReadWin (var Buf); external name 'crt_ReadWin';

{ Restore window contents saved by ReadWin. The size of the current
  window must match the size of the window from which ReadWin was
  used, but the position may be different. }
procedure WriteWin (const Buf); external name 'crt_WriteWin';

type
  WinState = record
    x1, y1, x2, y2, WhereX, WhereY, NewX1, NewY1, NewX2, NewY2:
  Integer;
    TextAttr: TTextAttr;
    CursorShape: TCursorShape;
    ScreenSize: TPoint;
    Buffer: ^Byte
  end;

{ Save window position and size, cursor position, text attribute and
  cursor shape -- *not* the window contents. }
procedure SaveWin (var State: WinState);

{ Make a new window (like Window), and save the contents of the
  screen below the window as well as the position and size, cursor
  position, text attribute and cursor shape of the old window. }
procedure MakeWin (var State: WinState; x1, y1, x2, y2: Integer);

{ Create window in full size, save previous text mode and all values
  that MakeWin does. }
procedure SaveScreen (var State: WinState);

{ Restore the data saved by SaveWin, MakeWin or SaveScreen. }
procedure RestoreWin (var State: WinState);

{ Panels }

type
  TPanel = Pointer;
```

```
function  GetActivePanel: TPanel; external
  name 'crt_GetActivePanel';
procedure PanelNew                 (x1, y1, x2, y2: CInteger;
  BindToBackground: Boolean); external name 'crt_PanelNew';
procedure PanelDelete              (Panel: TPanel); external
  name 'crt_PanelDelete';
procedure PanelBindToBackground    (Panel: TPanel; BindToBackground:
  Boolean); external name 'crt_PanelBindToBackground';
function  PanelIsBoundToBackground (Panel: TPanel): Boolean;
  external name 'crt_PanelIsBoundToBackground';
procedure PanelActivate            (Panel: TPanel); external
  name 'crt_PanelActivate';
procedure PanelHide                (Panel: TPanel); external
  name 'crt_PanelHide';
procedure PanelShow                (Panel: TPanel); external
  name 'crt_PanelShow';
function  PanelHidden              (Panel: TPanel): Boolean;
  external name 'crt_PanelHidden';
procedure PanelTop                 (Panel: TPanel); external
  name 'crt_PanelTop';
procedure PanelBottom              (Panel: TPanel); external
  name 'crt_PanelBottom';
procedure PanelMoveAbove           (Panel, Above: TPanel); external
  name 'crt_PanelMoveAbove';
procedure PanelMoveBelow           (Panel, Below: TPanel); external
  name 'crt_PanelMoveBelow';
function  PanelAbove               (Panel: TPanel): TPanel; external
  name 'crt_PanelAbove';
function  PanelBelow               (Panel: TPanel): TPanel; external
  name 'crt_PanelBelow';

{ TPCRT compatibility }

{ Write a string at the given position without moving the cursor.
  Truncated at the right margin. }
procedure WriteString (const s: String; y, x: Integer);

{ Write a string at the given position with the given attribute
  without moving the cursor. Truncated at the right margin. }
procedure FastWriteWindow (const s: String; y, x: Integer; Attr:
  TTextAttr);

{ Write a string at the given absolute position with the given
 attribute without moving the cursor. Truncated at the right
  margin. }
procedure FastWrite       (const s: String; y, x: Integer; Attr:
  TTextAttr);

{ WinCRT compatibility }
```

```
const
  cw_UseDefault = Integer ($8000);

var
  { Ignored }
  WindowOrg : TPoint = (cw_UseDefault, cw_UseDefault);
  WindowSize: TPoint = (cw_UseDefault, cw_UseDefault);
  Origin    : TPoint = (0, 0);
  InactiveTitle: PChar = '(Inactive %s)';
  AutoTracking: Boolean = True;
  WindowTitle: {$ifdef __BP_TYPE_SIZES__}
               array [0 .. 79] of Char
               {$else}
               TStringBuf
               {$endif};

  { Cursor location, 0-based }
  Cursor    : TPoint = (0, 0); attribute (name = 'crt_Cursor');

procedure InitWinCRT; attribute (name = 'crt_InitWinCRT');

{ Halts the program }
procedure DoneWinCRT; attribute (noreturn, name = 'crt_DoneWinCRT');

procedure WriteBuf (Buffer: PChar; Count: SizeType); attribute (name
  = 'crt_WriteBuf');

function  ReadBuf (Buffer: PChar; Count: SizeType): SizeType;
  attribute (name = 'crt_ReadBuf');

{ 0-based coordinates! }
procedure CursorTo (x, y: Integer); attribute (name
  = 'crt_CursorTo');

{ Dummy }
procedure ScrollTo (x, y: Integer); attribute (name
  = 'crt_ScrollTo');

{ Dummy }
procedure TrackCursor; attribute (name = 'crt_TrackCursor');
```

6.15.2 BP compatibility: Dos

The following listing contains the interface of the Dos unit.

This is a portable implementation of most routines from BP's 'Dos' unit. A few routines that are Dos – or even IA32 real mode – specific, are only available if '__BP_UNPORTABLE_ROUTINES__' is defined, Section 7.2 [BP Incompatibilities], page 237.

The same functionality and much more is available in the Run Time System, Section 6.14 [Run Time System], page 105. In some cases, the RTS routines have the same interface as the routines in this unit (e.g. 'GetEnv', 'FSplit', 'FExpand', 'FSearch'), in other cases, they have different names and/or easier and less limiting interfaces (e.g. 'ReadDir' etc. vs. 'FindFirst' etc.), and are often more efficient.

Therefore, using this unit is not recommended in newly written programs.

```
{ Portable BP compatible Dos unit

  This unit supports most of the routines and declarations of BP's
  Dos unit.

  Notes:

  - The procedures Keep, GetIntVec, SetIntVec are not supported
    since they make only sense for Dos real-mode programs (and GPC
    compiled programs do not run in real-mode, even on IA32 under
    Dos). The procedures Intr and MsDos are only supported under
    DJGPP if '__BP_UNPORTABLE_ROUTINES__' is defined (with the
    '-D__BP_UNPORTABLE_ROUTINES__' option). A few other routines are
    also only supported with this define, but on all platforms (but
    they are crude hacks, that's why they are not supported without
    this define).

  - The internal structure of file variables (FileRec and TextRec)
    is different in GPC. However, as far as TFDDs are concerned,
    there are other ways to achieve the same in GPC, see the GPC
    unit.

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Authors: Frank Heckenbach <frank@pascal.gnu.de>
           Prof. Abimbola A. Olowofoyeku <African_Chief@bigfoot.com>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
```

```
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-,maximum-field-alignment 0}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

module Dos;

{ GPC and this unit use ‘AnyFile’ for different meanings. Export
  renaming helps us to avoid a conflict here. If you use both units,
  the meaning of the latter one will be effective, but you always
  get the built-in meaning by using ‘GPC_AnyFile’. }
export Dos = all (DosAnyFile => AnyFile, FSearch, FExpand, FSplit,
  GetEnv);

import GPC (MaxLongInt => GPC_Orig_MaxLongInt); System;

type
  GPC_AnyFile = AnyFile;
  Byte8 = Cardinal attribute (Size = 8);
  Word16 = Cardinal attribute (Size = 16);
  Word32 = Cardinal attribute (Size = 32);
  TDosAttr = Word;

const
  { File attribute constants }
  ReadOnly   = $01;
  Hidden     = $02;  { set for dot files except '.' and '..' }
  SysFile    = $04;  { not supported }
  VolumeID   = $08;  { not supported }
  Directory  = $10;
  Archive    = $20;  { means: not executable }
  DosAnyFile = $3f;

  { Flag bit masks -- only used by the unportable Dos routines }
  FCarry     = 1;
  FParity    = 4;
  FAuxiliary = $10;
  FZero      = $40;
  FSign      = $80;
  FOverflow  = $800;

  { DosError codes }
  DosError_FileNotFound = 2;
  DosError_PathNotFound = 3;
  DosError_AccessDenied = 5;
  DosError_InvalidMem   = 9;
  DosErorr_InvalidEnv   = 10;
  DosError_NoMoreFiles  = 18;
```

```
  DosError_IOError      = 29;
  DosError_ReadFault    = 30;

type
  { String types. Not used in this unit, but declared for
    compatibility. }
  ComStr  = String [127];  { Command line string }
  PathStr = String [79];   { File pathname string }
  DirStr  = String [67];   { Drive and directory string }
  NameStr = String [8];    { File name string }
  ExtStr  = String [4];    { File extension string }

  TextBuf = array [0 .. 127] of Char;

  { Search record used by FindFirst and FindNext }
  SearchRecFill = packed array [1 .. 21] of Byte8;
  SearchRec = record
    Fill: SearchRecFill;
    Attr: Byte8;
    Time,
    Size: LongInt;
    Name: {$ifdef __BP_TYPE_SIZES__}
          String [12]
          {$else}
          TString
          {$endif}
  end;

  { Date and time record used by PackTime and UnpackTime }
  DateTime = record
    Year, Month, Day, Hour, Min, Sec: Word
  end;

  { 8086 CPU registers -- only used by the unportable Dos routines }
  Registers = record
  case Boolean of
    False: (ax, bx, cx, dx, bp, si, di, ds, es, Flags: Word16);
    True : (al, ah, bl, bh, cl, ch, dl, dh: Byte8)
  end;

var
  { Error status variable }
  DosError: Integer = 0;

procedure GetDate (var Year, Month, Day, DayOfWeek: Word);
procedure GetTime (var Hour, Minute, Second, Sec100: Word);
procedure GetCBreak (var BreakOn: Boolean);
procedure SetCBreak (BreakOn: Boolean);
{ GetVerify and SetVerify are dummies except for DJGPP (in the
  assumption that any real OS knows by itself when and how to verify
  its disks). }
```

```
procedure GetVerify (var VerifyOn: Boolean);
procedure SetVerify (VerifyOn: Boolean);
function  DiskFree (Drive: Byte): LongInt;
function  DiskSize (Drive: Byte): LongInt;
procedure GetFAttr (var f: GPC_AnyFile; var Attr: TDosAttr);
procedure SetFAttr (var f: GPC_AnyFile; Attr: TDosAttr);
procedure GetFTime (var f: GPC_AnyFile; var MTime: LongInt);
procedure SetFTime (var f: GPC_AnyFile; MTime: LongInt);

{ FindFirst and FindNext are quite inefficient since they emulate
  all the brain-dead Dos stuff. If at all possible, the standard
  routines OpenDir, ReadDir and CloseDir (in the GPC unit) should be
  used instead. }
procedure FindFirst (const Path: String; Attr: TDosAttr; var SR:
  SearchRec);
procedure FindNext  (var SR: SearchRec);

procedure FindClose (var SR: SearchRec);
procedure UnpackTime (p: LongInt; var t: DateTime);
procedure PackTime (const t: DateTime; var p: LongInt);
function  EnvCount: Integer;
function  EnvStr (EnvIndex: Integer): TString;
procedure SwapVectors;
{ Exec executes a process via Execute, so RestoreTerminal is called
  with the argument True before and False after executing the
  process. }
procedure Exec (const Path, Params: String);
function  DosExitCode: Word;

{ Unportable Dos-only routines and declarations }

{$ifdef __BP_UNPORTABLE_ROUTINES__}
{$ifdef __GO32__}
{ These are unportable Dos-only declarations and routines, since
  interrupts are Dos and CPU specific (and have no place in a
  high-level program, anyway). }
procedure Intr (IntNo: Byte; var Regs: Registers);
procedure MsDos (var Regs: Registers);
{$endif}

{ Though probably all non-Dos systems have versions numbers as well,
  returning them here would usually not do what is expected, e.g.
  testing if certain Dos features are present by comparing the
  version number. Therefore, this routine always returns 7 (i.e.,
  version 7.0) on non-Dos systems, in the assumption that any real
  OS has at least the features of Dos 7. }
function  DosVersion: Word;

{ Changing the system date and time is a system administration task,
  not allowed to a normal process. On non-Dos systems, these
  routines emulate the changed date/time, but only for GetTime and
```

```
  GetDate (not the RTS date/time routines), and only for this
  process, not for child processes or even the parent process or
  system-wide. }
procedure SetDate (Year, Month, Day: Word);
procedure SetTime (Hour, Minute, Second, Sec100: Word);
{$endif}
```

6.15.3 Overcome some differences between Dos and Unix

The following listing contains the interface of the DosUnix unit.

This unit is there to overcome some of those differences between Dos and Unix systems that are not automatically hidden by GPC and the Run Time System. Currently features translation of bash style input/output redirections ('`foo 2>&1`') into '`redir`' calls for DJGPP ('`redir -eo foo`') and a way to read files with Dos CR/LF pairs on any system.

When necessary, new features will be added to the unit in future releases.

```
{ Some routines to support writing programs portable between Dos and
  Unix. Perhaps it would be a good idea not to put features to make
  Dos programs Unix-compatible (shell redirections) and vice versa
  (reading Dos files from Unix) together into one unit, but rather
  into two units, DosCompat and UnixCompat or so -- let's wait and
  see, perhaps when more routines suited for this/these unit(s) will
  be found, the design will become clearer ...

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }
```

```
{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

unit DosUnix;

interface

uses GPC;

{ This function is meant to be used when you want to invoke a system
  shell command (e.g. via Execute or Exec from the Dos unit) and
  want to specify input/output redirections for the command invoked.
  It caters for the different syntax between DJGPP (with the `redir'
  utility) and other systems.

  To use it, code your redirections in bash style (see the table
  below) in your command line string, pass this string to this
  function, and the function's result to Execute or the other
  routines.

  The function translates the following bash style redirections
  (characters in brackets are optional) into a redir call under Dos
  systems except EMX, and leave them unchanged under other systems.
  Note: `redir' comes with DJGPP, but it should be possible to
  install it on other Dos systems as well. OS/2's shell, however,
  supports bash style redirections, I was told, so we don't
  translate on EMX.

  [0]<     file      redirect standard input from file
  [1]>[|]  file      redirect standard output to file
  [1]>>    file      append standard output to file
  [1]>&2             redirect standard output to standard error
  2>[|]    file      redirect standard error to file
  2>>      file      append standard error to file
  2>&1               redirect standard error to standard output
  &> file            redirect both standard output and standard
                     error to file }
function  TranslateRedirections (const Command: String) = s:
  TString;

{ Under Unix, translates CR/LF pairs to single LF characters when
  reading from f, and back when writing to f. Under Dos, does
  nothing because the run time system alrady does this job. In the
  result, you can read both Dos and Unix files, and files written
  will be Dos. }
procedure AssignDos (var f: AnyFile; const FileName: String);

{ Translates a character from the "OEM" charset used under Dos to
```

```
  the ISO-8859-1 (Latin1) character set. }
function  OEM2Latin1 (ch: Char): Char;
function  OEM2Latin1Str (const s: String) = r: TString;

{ Translates a character from the ISO-8859-1 (Latin1) character set
  to the "OEM" charset used under Dos. }
function  Latin12OEM (ch: Char): Char;
function  Latin12OEMStr (const s: String) = r: TString;
```

6.15.4 Higher level file and directory handling

The following listing contains the interface of the FileUtils unit.

This unit provides some routines for file and directory handling on a higher level than those provided by the RTS.

```
{ Some routines for file and directory handling on a higher level
  than those provided by the RTS.

  Copyright (C) 2000-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}
```

```
unit FileUtils;

interface

uses GPC;

type
  TStringProc = procedure (const s: String);

{ Finds all files matching the given Mask in the given Directory and
  all subdirectories of it. The matching is done using all wildcards
  and brace expansion, like MultiFileNameMatch does. For each file
  found, FileAction is executed. For each directory found (including
  `.' and `..' if they match the Mask!), DirAction is executed. If
  MainDirFirst is True, this happens before processing the files in
  the directory and below, otherwise afterwards. (The former is
  useful, e.g., if this is used to copy a directory tree and
  DirAction does a MkDir, while the latter behaviour is required
  when removing a directory tree and DirAction does a RmDir.) Both
  FileAction and DirAction can be nil in which case nothing is done
  for files or directories found, respectively. (If DirAction is
  nil, the value of DirsFirst does not matter.) Of course,
  FileAction and DirAction may also be identical. The procedure
  leaves InOutRes set in case of any error. If FileAction or
  DirAction return with InOutRes set, FindFiles recognizes this and
  returns immediately. }
procedure FindFiles (const Directory, Mask: String; MainDirFirst:
  Boolean;
                     FileAction, DirAction: TStringProc); attribute
  (iocritical);

{ Creates the directory given by Path and all directories in between
  that are necessary. Does not report an error if the directory
  already exists, but, of course, if it cannot be created because of
  missing permissions or because Path already exists as a file. }
procedure MkDirs (const Path: String); attribute (iocritical);

{ Removes Path if empty as well as any empty parent directories.
  Does not report an error if the directory is not empty. }
procedure RmDirs (const Path: String); attribute (iocritical);

{ Copies the file Source to Dest, overwriting Dest if it exists and
  can be written to. Returns any errors in IOResult. If Mode >= 0,
  it will change the permissions of Dest to Mode immediately after
  creating it and before writing any data to it. That's useful,
  e.g., if Dest is not meant to be world-readable, because if you'd
  do a ChMod after FileCopy, you might leave the data readable
  (depending on the umask) during the copying. If Mode < 0, Dest
  will be set to the same permissions Source has. In any case, Dest
  will be set to the modification time of Source after copying. On
  any error, the destination file is erased. This is to avoid
```

```
    leaving partial files in case of full file systems (one of the
    most common reasons for errors). }
  procedure FileCopy (const Source, Dest: String; Mode: Integer);
    attribute (iocritical);

  { Creates a backup of FileName in the directory BackupDirectory or,
    if BackupDirectory is empty, in the directory of FileName. Errors
    are returned in IOResult (and on any error, no partial backup file
    is left), but if FileName does not exist, this does *not* count as
    an error (i.e., BackupFile will just return without setting
    IOResult then). If OnlyUserReadable is True, the backup file will
    be given only user-read permissions, nothing else.

    The name chosen for the backup depends on the Simple and Short
    parameters. The short names will fit into 8+3 characters (whenever
    possible), while the long ones conform to the conventions used by
    most GNU tools. If Simple is True, a simple backup file name will
    be used, and previous backups under the same name will be
    overwritten (if possible). Otherwise, backups will be numbered,
    where the number is chosen to be larger than all existing backups,
    so it will be unique and increasing in chronological order. In
    particular:

    Simple  Short  Backup name
    True    True   Base name of FileName plus '.bak'
    False   True   Base name of FileName plus '.b' plus a number
    True    False  Base name plus extension of FileName plus '~'
    False   False  Base name plus extension of FileName plus '.~', a
                   number and '~' }
  procedure BackupFile (const FileName, BackupDirectory: String;
    Simple, Short, OnlyUserReadable: Boolean); attribute (iocritical);
```

6.15.5 Arithmetic with unlimited size and precision

The following listing contains the interface of the GMP unit.

This unit provides an interface to the GNU Multiprecision Library to perform arithmetic on integer, rational and real numbers of unlimited size and precision.

To use this unit, you will need the 'gmp' library which can be found in http://www.gnu-pascal.de/libs/.

```
  { Definitions for GNU multiple precision functions: arithmetic with
    integer, rational and real numbers of arbitrary size and
    precision.

    Translation of the C header (gmp.h) of the GMP library. Tested
    with GMP 3.x and 4.x.

    To use the GMP unit, you will need the GMP library which can be
    found in http://www.gnu-pascal.de/libs/

    Copyright (C) 1998-2005 Free Software Foundation, Inc.
```

```
  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License.

  Please also note the license of the GMP library. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}
{$nested-comments}

{ If HAVE_GMP4 is set (the default unless HAVE_GMP3 is set, some
  interface changes made in GMP 4 are taken into account.
  I.e., if this is set wrong, programs might fail. However, this
  only affects a few routines related to random numbers. }
{$if not defined (HAVE_GMP3)}
{$define HAVE_GMP4}
{$endif}

{$undef GMP}  { in case it's set by the user }
unit GMP;

interface

uses GPC;

{$if defined (__mips) and defined (_ABIN32) and defined (HAVE_GMP3)}
```

```
{ Force the use of 64-bit limbs for all 64-bit MIPS CPUs if ABI
  permits. }
{$define _LONG_LONG_LIMB}
{$endif}

type
  {$ifdef _SHORT_LIMB}
  mp_limb_t        = CCardinal;
  mp_limb_signed_t = CInteger;
  {$elif defined (_LONG_LONG_LIMB)}
  mp_limb_t        = LongCard;
  mp_limb_signed_t = LongInt;
  {$else}
  mp_limb_t        = MedCard;
  mp_limb_signed_t = MedInt;
  {$endif}

  mp_ptr           = ^mp_limb_t;

  {$if defined (_CRAY) and not defined (_CRAYMPP)}
  mp_size_t        = CInteger;
  mp_exp_t         = CInteger;
  {$else}
  mp_size_t        = MedInt;
  mp_exp_t         = MedInt;
  {$endif}

  mpz_t = record
    mp_alloc,
    mp_size: CInteger;
    mp_d:    mp_ptr
  end;

  mpz_array_ptr = ^mpz_array;
  mpz_array = array [0 .. MaxVarSize div SizeOf (mpz_t) - 1] of
  mpz_t;

  mpq_t = record
    mp_num,
    mp_den: mpz_t
  end;

  mpf_t = record
    mp_prec,
    mp_size: CInteger;
    mp_exp:  mp_exp_t;
    mp_d:    mp_ptr
  end;

  TAllocFunction    = function (Size: SizeType): Pointer;
  TReAllocFunction  = function (var Dest: Pointer; OldSize, NewSize:
```

```
  SizeType): Pointer;
  TDeAllocProcedure = procedure (Src: Pointer; Size: SizeType);

var
  mp_bits_per_limb: CInteger; attribute (const); external
  name '__gmp_bits_per_limb';

procedure mp_set_memory_functions (AllocFunction: TAllocFunction;
                                   ReAllocFunction:
  TReAllocFunction;
                                   DeAllocProcedure:
  TDeAllocProcedure); external name '__gmp_set_memory_functions';

{ Integer (i.e. Z) routines }

procedure mpz_init             (var Dest: mpz_t); external
  name '__gmpz_init';
procedure mpz_clear            (var Dest: mpz_t); external
  name '__gmpz_clear';
function  mpz_realloc          (var Dest: mpz_t; NewAlloc:
  mp_size_t): Pointer; external name '__gmpz_realloc';
procedure mpz_array_init       (Dest: mpz_array_ptr; ArraySize,
  FixedNumBits: mp_size_t); external name '__gmpz_array_init';

procedure mpz_set              (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_set';
procedure mpz_set_ui           (var Dest: mpz_t; Src: MedCard);
  external name '__gmpz_set_ui';
procedure mpz_set_si           (var Dest: mpz_t; Src: MedInt);
  external name '__gmpz_set_si';
procedure mpz_set_d            (var Dest: mpz_t; Src: Real);
  external name '__gmpz_set_d';
procedure mpz_set_q            (var Dest: mpz_t; Src: mpq_t);
  external name '__gmpz_set_q';
procedure mpz_set_f            (var Dest: mpz_t; Src: mpf_t);
  external name '__gmpz_set_f';
function  mpz_set_str          (var Dest: mpz_t; Src: CString; Base:
  CInteger): CInteger; external name '__gmpz_set_str';

procedure mpz_init_set         (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_init_set';
procedure mpz_init_set_ui      (var Dest: mpz_t; Src: MedCard);
  external name '__gmpz_init_set_ui';
procedure mpz_init_set_si      (var Dest: mpz_t; Src: MedInt);
  external name '__gmpz_init_set_si';
procedure mpz_init_set_d       (var Dest: mpz_t; Src: Real);
  external name '__gmpz_init_set_d';
function  mpz_init_set_str     (var Dest: mpz_t; Src: CString; Base:
  CInteger): CInteger; external name '__gmpz_init_set_str';

function  mpz_get_ui           (protected var Src: mpz_t): MedCard;
```

```
  external name '__gmpz_get_ui';
function  mpz_get_si          (protected var Src: mpz_t): MedInt;
  external name '__gmpz_get_si';
function  mpz_get_d           (protected var Src: mpz_t): Real;
  external name '__gmpz_get_d';
{ Pass nil for Dest to let the function allocate memory for it }
function  mpz_get_str         (Dest: CString; Base: CInteger;
  protected var Src: mpz_t): CString; external
  name '__gmpz_get_str';

procedure mpz_add             (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_add';
procedure mpz_add_ui          (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_add_ui';
procedure mpz_sub             (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_sub';
procedure mpz_sub_ui          (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_sub_ui';
procedure mpz_mul             (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_mul';
procedure mpz_mul_ui          (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_mul_ui';
procedure mpz_mul_2exp        (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_mul_2exp';
procedure mpz_neg             (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_neg';
procedure mpz_abs             (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_abs';
procedure mpz_fac_ui          (var Dest: mpz_t; Src: MedCard);
  external name '__gmpz_fac_ui';

procedure mpz_tdiv_q          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_tdiv_q';
procedure mpz_tdiv_q_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_tdiv_q_ui';
procedure mpz_tdiv_r          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_tdiv_r';
procedure mpz_tdiv_r_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_tdiv_r_ui';
procedure mpz_tdiv_qr         (var DestQ, DestR: mpz_t; protected
  var Src1, Src2: mpz_t); external name '__gmpz_tdiv_qr';
procedure mpz_tdiv_qr_ui      (var DestQ, DestR: mpz_t; protected
  var Src1: mpz_t; Src2: MedCard); external
  name '__gmpz_tdiv_qr_ui';

procedure mpz_fdiv_q          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_fdiv_q';
function  mpz_fdiv_q_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard): MedCard; external name '__gmpz_fdiv_q_ui';
procedure mpz_fdiv_r          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_fdiv_r';
```

```
function  mpz_fdiv_r_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard): MedCard; external name '__gmpz_fdiv_r_ui';
procedure mpz_fdiv_qr         (var DestQ, DestR: mpz_t; protected
  var Src1, Src2: mpz_t); external name '__gmpz_fdiv_qr';
function  mpz_fdiv_qr_ui      (var DestQ, DestR: mpz_t; protected
  var Src1: mpz_t; Src2: MedCard): MedCard; external
  name '__gmpz_fdiv_qr_ui';
function  mpz_fdiv_ui         (protected var Src1: mpz_t; Src2:
  MedCard): MedCard; external name '__gmpz_fdiv_ui';

procedure mpz_cdiv_q          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_cdiv_q';
function  mpz_cdiv_q_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard): MedCard; external name '__gmpz_cdiv_q_ui';
procedure mpz_cdiv_r          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_cdiv_r';
function  mpz_cdiv_r_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard): MedCard; external name '__gmpz_cdiv_r_ui';
procedure mpz_cdiv_qr         (var DestQ, DestR: mpz_t; protected
  var Src1,Src2: mpz_t); external name '__gmpz_cdiv_qr';
function  mpz_cdiv_qr_ui      (var DestQ, DestR: mpz_t; protected
  var Src1: mpz_t; Src2: MedCard): MedCard; external
  name '__gmpz_cdiv_qr_ui';
function  mpz_cdiv_ui         (protected var Src1: mpz_t;
  Src2:MedCard): MedCard; external name '__gmpz_cdiv_ui';

procedure mpz_mod             (var Dest: mpz_t; protected var
  Src1,Src2: mpz_t); external name '__gmpz_mod';
procedure mpz_divexact        (var Dest: mpz_t; protected var
  Src1,Src2: mpz_t); external name '__gmpz_divexact';

procedure mpz_tdiv_q_2exp     (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_tdiv_q_2exp';
procedure mpz_tdiv_r_2exp     (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_tdiv_r_2exp';
procedure mpz_fdiv_q_2exp     (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_fdiv_q_2exp';
procedure mpz_fdiv_r_2exp     (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_fdiv_r_2exp';

procedure mpz_powm            (var Dest: mpz_t; protected var Base,
  Exponent, Modulus: mpz_t); external name '__gmpz_powm';
procedure mpz_powm_ui         (var Dest: mpz_t; protected var Base:
  mpz_t; Exponent: MedCard; protected var Modulus: mpz_t); external
  name '__gmpz_powm_ui';
procedure mpz_pow_ui          (var Dest: mpz_t; protected var Base:
  mpz_t; Exponent: MedCard); external name '__gmpz_pow_ui';
procedure mpz_ui_pow_ui       (var Dest: mpz_t; Base, Exponent:
  MedCard); external name '__gmpz_ui_pow_ui';

procedure mpz_sqrt            (var Dest: mpz_t; protected var Src:
```

```
  mpz_t); external name '__gmpz_sqrt';
procedure mpz_sqrtrem          (var Dest, DestR: mpz_t; protected
  var Src: mpz_t); external name '__gmpz_sqrtrem';
function  mpz_perfect_square_p (protected var Src: mpz_t): CInteger;
  external name '__gmpz_perfect_square_p';

function  mpz_probab_prime_p   (protected var Src: mpz_t;
  Repetitions: CInteger): CInteger; external
  name '__gmpz_probab_prime_p';
procedure mpz_gcd              (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_gcd';
function  mpz_gcd_ui           (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard): MedCard; external name '__gmpz_gcd_ui';
procedure mpz_gcdext           (var Dest, DestA, DestB: mpz_t;
  protected var SrcA, SrcB: mpz_t); external name '__gmpz_gcdext';
function  mpz_invert           (var Dest: mpz_t; protected var Src,
  Modulus: mpz_t): CInteger; external name '__gmpz_invert';
function  mpz_jacobi           (protected var Src1, Src2: mpz_t):
  CInteger; external name '__gmpz_jacobi';

function  mpz_cmp              (protected var Src1, Src2: mpz_t):
  CInteger; external name '__gmpz_cmp';
function  mpz_cmp_ui           (protected var Src1: mpz_t; Src2:
  MedCard): CInteger; external name '__gmpz_cmp_ui';
function  mpz_cmp_si           (protected var Src1: mpz_t; Src2:
  MedInt): CInteger; external name '__gmpz_cmp_si';
function  mpz_sgn              (protected var Src: mpz_t): CInteger;
  attribute (inline);

procedure mpz_and              (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_and';
procedure mpz_ior              (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_ior';
procedure mpz_com              (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_com';
function  mpz_popcount         (protected var Src: mpz_t): MedCard;
  external name '__gmpz_popcount';
function  mpz_hamdist          (protected var Src1, Src2: mpz_t):
  MedCard; external name '__gmpz_hamdist';
function  mpz_scan0            (protected var Src: mpz_t;
  StartingBit: MedCard): MedCard; external name '__gmpz_scan0';
function  mpz_scan1            (protected var Src: mpz_t;
  StartingBit: MedCard): MedCard; external name '__gmpz_scan1';
procedure mpz_setbit           (var Dest: mpz_t; BitIndex: MedCard);
  external name '__gmpz_setbit';
procedure mpz_clrbit           (var Dest: mpz_t; BitIndex: MedCard);
  external name '__gmpz_clrbit';

procedure mpz_random           (var Dest: mpz_t; MaxSize:
  mp_size_t); external name '__gmpz_random';
procedure mpz_random2          (var Dest: mpz_t; MaxSize:
```

```
  mp_size_t); external name '__gmpz_random2';
function  mpz_sizeinbase       (protected var Src: mpz_t; Base:
  CInteger): SizeType; external name '__gmpz_sizeinbase';

{ Rational (i.e. Q) routines }

procedure mpq_canonicalize     (var Dest: mpq_t); external
  name '__gmpq_canonicalize';

procedure mpq_init             (var Dest: mpq_t); external
  name '__gmpq_init';
procedure mpq_clear            (var Dest: mpq_t); external
  name '__gmpq_clear';
procedure mpq_set              (var Dest: mpq_t; protected var Src:
  mpq_t); external name '__gmpq_set';
procedure mpq_set_z            (var Dest: mpq_t; protected var Src:
  mpz_t); external name '__gmpq_set_z';
procedure mpq_set_ui           (var Dest: mpq_t; Nom, Den: MedCard);
  external name '__gmpq_set_ui';
procedure mpq_set_si           (var Dest: mpq_t; Nom: MedInt; Den:
  MedCard); external name '__gmpq_set_si';

procedure mpq_add              (var Dest: mpq_t; protected var Src1,
  Src2: mpq_t); external name '__gmpq_add';
procedure mpq_sub              (var Dest: mpq_t; protected var Src1,
  Src2: mpq_t); external name '__gmpq_sub';
procedure mpq_mul              (var Dest: mpq_t; protected var Src1,
  Src2: mpq_t); external name '__gmpq_mul';
procedure mpq_div              (var Dest: mpq_t; protected var Src1,
  Src2: mpq_t); external name '__gmpq_div';
procedure mpq_neg              (var Dest: mpq_t; protected var Src:
  mpq_t); external name '__gmpq_neg';
procedure mpq_inv              (var Dest: mpq_t; protected var Src:
  mpq_t); external name '__gmpq_inv';

function  mpq_cmp              (protected var Src1, Src2: mpq_t):
  CInteger; external name '__gmpq_cmp';
function  mpq_cmp_ui           (protected var Src1: mpq_t; Nom2,
  Den2: MedCard): CInteger; external name '__gmpq_cmp_ui';
function  mpq_sgn              (protected var Src: mpq_t): CInteger;
  attribute (inline);
function  mpq_equal            (protected var Src1, Src2: mpq_t):
  CInteger; external name '__gmpq_equal';

function  mpq_get_d            (protected var Src: mpq_t): Real;
  external name '__gmpq_get_d';
procedure mpq_set_num          (var Dest: mpq_t; protected var Src:
  mpz_t); external name '__gmpq_set_num';
procedure mpq_set_den          (var Dest: mpq_t; protected var Src:
  mpz_t); external name '__gmpq_set_den';
procedure mpq_get_num          (var Dest: mpz_t; protected var Src:
```

```
  mpq_t); external name '__gmpq_get_num';
procedure mpq_get_den         (var Dest: mpz_t; protected var Src:
  mpq_t); external name '__gmpq_get_den';

{ Floating point (i.e. R) routines }

procedure mpf_set_default_prec (Precision: MedCard); external
  name '__gmpf_set_default_prec';
procedure mpf_init            (var Dest: mpf_t); external
  name '__gmpf_init';
procedure mpf_init2           (var Dest: mpf_t; Precision:
  MedCard); external name '__gmpf_init2';
procedure mpf_clear           (var Dest: mpf_t); external
  name '__gmpf_clear';
procedure mpf_set_prec        (var Dest: mpf_t; Precision:
  MedCard); external name '__gmpf_set_prec';
function  mpf_get_prec        (protected var Src: mpf_t): MedCard;
  external name '__gmpf_get_prec';
procedure mpf_set_prec_raw    (var Dest: mpf_t; Precision:
  MedCard); external name '__gmpf_set_prec_raw';

procedure mpf_set             (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_set';
procedure mpf_set_ui          (var Dest: mpf_t; Src: MedCard);
  external name '__gmpf_set_ui';
procedure mpf_set_si          (var Dest: mpf_t; Src: MedInt);
  external name '__gmpf_set_si';
procedure mpf_set_d           (var Dest: mpf_t; Src: Real);
  external name '__gmpf_set_d';
procedure mpf_set_z           (var Dest: mpf_t; protected var Src:
  mpz_t); external name '__gmpf_set_z';
procedure mpf_set_q           (var Dest: mpf_t; protected var Src:
  mpq_t); external name '__gmpf_set_q';
function  mpf_set_str         (var Dest: mpf_t; Src: CString; Base:
  CInteger): CInteger; external name '__gmpf_set_str';

procedure mpf_init_set        (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_init_set';
procedure mpf_init_set_ui     (var Dest: mpf_t; Src: MedCard);
  external name '__gmpf_init_set_ui';
procedure mpf_init_set_si     (var Dest: mpf_t; Src: MedInt);
  external name '__gmpf_init_set_si';
procedure mpf_init_set_d      (var Dest: mpf_t; Src: Real);
  external name '__gmpf_init_set_d';
function  mpf_init_set_str    (var Dest: mpf_t; Src: CString; Base:
  CInteger): CInteger; external name '__gmpf_init_set_str';

function  mpf_get_d           (protected var Src: mpf_t): Real;
  external name '__gmpf_get_d';
{ Pass nil for Dest to let the function allocate memory for it }
function  mpf_get_str         (Dest: CString; var Exponent:
```

```
  mp_exp_t; Base: CInteger;
                              NumberOfDigits: SizeType; protected
  var Src: mpf_t): CString; external name '__gmpf_get_str';

procedure mpf_add              (var Dest: mpf_t; protected var Src1,
  Src2: mpf_t); external name '__gmpf_add';
procedure mpf_add_ui           (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_add_ui';
procedure mpf_sub              (var Dest: mpf_t; protected var Src1,
  Src2: mpf_t); external name '__gmpf_sub';
procedure mpf_ui_sub           (var Dest: mpf_t; Src1: MedCard;
  protected var Src2: mpf_t); external name '__gmpf_ui_sub';
procedure mpf_sub_ui           (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_sub_ui';
procedure mpf_mul              (var Dest: mpf_t; protected var Src1,
  Src2: mpf_t); external name '__gmpf_mul';
procedure mpf_mul_ui           (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_mul_ui';
procedure mpf_div              (var Dest: mpf_t; protected var Src1,
  Src2: mpf_t); external name '__gmpf_div';
procedure mpf_ui_div           (var Dest: mpf_t; Src1: MedCard;
  protected var Src2: mpf_t); external name '__gmpf_ui_div';
procedure mpf_div_ui           (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_div_ui';
procedure mpf_sqrt             (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_sqrt';
procedure mpf_sqrt_ui          (var Dest: mpf_t; Src: MedCard);
  external name '__gmpf_sqrt_ui';
procedure mpf_neg              (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_neg';
procedure mpf_abs              (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_abs';
procedure mpf_mul_2exp         (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_mul_2exp';
procedure mpf_div_2exp         (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_div_2exp';

function  mpf_cmp              (protected var Src1, Src2: mpf_t):
  CInteger; external name '__gmpf_cmp';
function  mpf_cmp_si           (protected var Src1: mpf_t; Src2:
  MedInt): CInteger; external name '__gmpf_cmp_si';
function  mpf_cmp_ui           (protected var Src1: mpf_t; Src2:
  MedCard): CInteger; external name '__gmpf_cmp_ui';
function  mpf_eq               (protected var Src1, Src2: mpf_t;
  NumberOfBits: MedCard): CInteger; external name '__gmpf_eq';
procedure mpf_reldiff          (var Dest: mpf_t; protected var Src1,
  Src2: mpf_t); external name '__gmpf_reldiff';
function  mpf_sgn              (protected var Src: mpf_t): CInteger;
  attribute (inline);

procedure mpf_random2          (var Dest: mpf_t; MaxSize: mp_size_t;
```

```
  MaxExp: mp_exp_t); external name '__gmpf_random2';

{$if False}  { @@ commented out because they use C file pointers }
function  mpz_inp_str         (var Dest: mpz_t; Src: CFilePtr;
  Base: CInteger): SizeType; external name '__gmpz_inp_str';
function  mpz_inp_raw         (var Dest: mpz_t; Src: CFilePtr):
  SizeType; external name '__gmpz_inp_raw';
function  mpz_out_str         (Dest: CFilePtr; Base: CInteger;
  protected var Src: mpz_t): SizeType; external
  name '__gmpz_out_str';
function  mpz_out_raw         (Dest: CFilePtr; protected var Src:
  mpz_t): SizeType ; external name '__gmpz_out_raw';
{ @@ mpf_out_str has a bug in GMP 2.0.2: it writes a spurious #0
  before the exponent for negative numbers }
function  mpf_out_str         (Dest: CFilePtr; Base: CInteger;
  NumberOfDigits: SizeType; protected var Src: mpf_t): SizeType;
  external name '__gmpf_out_str';
function  mpf_inp_str         (var Dest: mpf_t; Src: CFilePtr;
  Base: CInteger): SizeType; external name '__gmpf_inp_str';
{$endif}

{ Available random number generation algorithms. }
type
  gmp_randalg_t = (GMPRandAlgLC { Linear congruential. });

const
  GMPRandAlgDefault = GMPRandAlgLC;

{ Linear congruential data struct. }
type
  gmp_randata_lc = record
    a: mpz_t;  { Multiplier. }
    c: MedCard;  { Adder. }
    m: mpz_t;  { Modulus (valid only if M2Exp = 0). }
    M2Exp: MedCard;  { If <> 0, modulus is 2 ^ M2Exp. }
  end;

type
  gmp_randstate_t = record
    Seed: mpz_t;  { Current seed. }
    Alg: gmp_randalg_t;  { Algorithm used. }
    AlgData: record  { Algorithm specific data. }
    case gmp_randalg_t of
      GMPRandAlgLC: (lc: ^gmp_randata_lc)  { Linear congruential. }
    end
  end;

procedure gmp_randinit         (var State: gmp_randstate_t; Alg:
  gmp_randalg_t; ...); external name '__gmp_randinit';
procedure gmp_randinit_lc      (var State: gmp_randstate_t; {$ifdef
  HAVE_GMP4} protected var {$endif} a: mpz_t; c: MedCard; {$ifdef
```

```
  HAVE_GMP4} protected var {$endif} m: mpz_t); external
  name '__gmp_randinit_lc';
procedure gmp_randinit_lc_2exp (var State: gmp_randstate_t; {$ifdef
  HAVE_GMP4} protected var {$endif} a: mpz_t; c: MedCard; M2Exp:
  MedCard); external name '__gmp_randinit_lc_2exp';
procedure gmp_randseed        (var State: gmp_randstate_t; Seed:
  mpz_t); external name '__gmp_randseed';
procedure gmp_randseed_ui     (var State: gmp_randstate_t; Seed:
  MedCard); external name '__gmp_randseed_ui';
procedure gmp_randclear       (var State: gmp_randstate_t);
  external name '__gmp_randclear';

procedure mpz_addmul_ui       (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_addmul_ui';
procedure mpz_bin_ui          (var Dest: mpz_t; protected var Src1:
  mpz_t; Src2: MedCard); external name '__gmpz_bin_ui';
procedure mpz_bin_uiui        (var Dest: mpz_t; Src1, Src2:
  MedCard); external name '__gmpz_bin_uiui';
function  mpz_cmpabs          (protected var Src1, Src2: mpz_t):
  CInteger; external name '__gmpz_cmpabs';
function  mpz_cmpabs_ui       (protected var Src1: mpz_t; Src2:
  MedCard): CInteger; external name '__gmpz_cmpabs_ui';
procedure mpz_dump            (protected var Src: mpz_t); external
  name '__gmpz_dump';
procedure mpz_fib_ui          (var Dest: mpz_t; Src: MedCard);
  external name '__gmpz_fib_ui';
function  mpz_fits_sint_p     (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_sint_p';
function  mpz_fits_slong_p    (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_slong_p';
function  mpz_fits_sshort_p   (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_sshort_p';
function  mpz_fits_uint_p     (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_uint_p';
function  mpz_fits_ulong_p    (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_ulong_p';
function  mpz_fits_ushort_p   (protected var Src: mpz_t): CInteger;
  external name '__gmpz_fits_ushort_p';
procedure mpz_lcm             (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_lcm';
procedure mpz_nextprime       (var Dest: mpz_t; protected var Src:
  mpz_t); external name '__gmpz_nextprime';
function  mpz_perfect_power_p (protected var Src: mpz_t): CInteger;
  external name '__gmpz_perfect_power_p';
function  mpz_remove          (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t): MedCard; external name '__gmpz_remove';
function  mpz_root            (var Dest: mpz_t; protected var Src:
  mpz_t; n: MedCard): CInteger; external name '__gmpz_root';
procedure mpz_rrandomb        (var ROP: mpz_t; var State:
  gmp_randstate_t; n: MedCard); external name '__gmpz_rrandomb';
procedure mpz_swap            (var v1, v2: mpz_t); external
```

```
  name '__gmpz_swap';
function  mpz_tdiv_ui          (protected var Src1: mpz_t; Src2:
  MedCard): MedCard; external name '__gmpz_tdiv_ui';
function  mpz_tstbit           (protected var Src1: mpz_t; Src2:
  MedCard): CInteger; external name '__gmpz_tstbit';
procedure mpz_urandomb         ({$ifdef HAVE_GMP4} var {$endif} ROP:
  mpz_t; var State: gmp_randstate_t; n: MedCard); external
  name '__gmpz_urandomb';
procedure mpz_urandomm         ({$ifdef HAVE_GMP4} var {$endif} ROP:
  mpz_t; var State: gmp_randstate_t; {$ifdef HAVE_GMP4} protected
  var {$endif} n: mpz_t); external name '__gmpz_urandomm';
procedure mpz_xor              (var Dest: mpz_t; protected var Src1,
  Src2: mpz_t); external name '__gmpz_xor';

procedure mpq_set_d            (var Dest: mpq_t; Src: Real);
  external name '__gmpq_set_d';

procedure mpf_ceil             (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_ceil';
procedure mpf_floor            (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_floor';
{$ifdef HAVE_GMP4}
function  mpf_get_si           (protected var Src: mpf_t): MedInt;
  external name '__gmpf_get_si';
function  mpf_get_ui           (protected var Src: mpf_t): MedCard;
  external name '__gmpf_get_ui';
function  mpf_get_d_2exp       (var Exp: MedInt; protected var Src:
  mpf_t): Real; external name '__gmpf_get_d_2exp';
{$endif}
procedure mpf_pow_ui           (var Dest: mpf_t; protected var Src1:
  mpf_t; Src2: MedCard); external name '__gmpf_pow_ui';
procedure mpf_trunc            (var Dest: mpf_t; protected var Src:
  mpf_t); external name '__gmpf_trunc';
procedure mpf_urandomb         (ROP: mpf_t; var State:
  gmp_randstate_t; n: MedCard); external name '__gmpf_urandomb';

const
  GMPErrorNone = 0;
  GMPErrorUnsupportedArgument = 1;
  GMPErrorDivisionByZero = 2;
  GMPErrorSqrtOfNegative = 4;
  GMPErrorInvalidArgument = 8;
  GMPErrorAllocate = 16;

var
  gmp_errno: CInteger; external name '__gmp_errno';

{ Extensions to the GMP library, implemented in this unit }

procedure mpf_exp    (var Dest: mpf_t; protected var Src: mpf_t);
procedure mpf_ln     (var Dest: mpf_t; protected var Src: mpf_t);
```

```
procedure mpf_pow    (var Dest: mpf_t; protected var Src1, Src2:
  mpf_t);
procedure mpf_sin    (var Dest: mpf_t; protected var Src: mpf_t);
procedure mpf_cos    (var Dest: mpf_t; protected var Src: mpf_t);
procedure mpf_arctan (var Dest: mpf_t; protected var Src: mpf_t);
procedure mpf_pi     (var Dest: mpf_t);
```

6.15.6 Turbo Power compatibility, etc.

The following listing contains the interface of the GPCUtil unit.

This unit provides some utility routines for compatibility to some units available for BP, like some *Turbo Power* units.

```
{ Some utility routines for compatibility to some units available
  for BP, like some 'Turbo Power' units.

  @@NOTE - SOME OF THE ROUTINES IN THIS UNIT MAY NOT WORK CORRECTLY.
  TEST CAREFULLY AND USE WITH CARE!

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Authors: Prof. Abimbola A. Olowofoyeku <African_Chief@bigfoot.com>
           Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
```

```
{$endif}

module GPCUtil;

export GPCUtil = all
       (
        { Return the current working directory }
        GetCurrentDirectory => ThisDirectory,

        { Does a directory exist? }
        DirectoryExists => IsDirectory,

        { Does file name s exist? }
        FileExists => ExistFile,

        { Return just the directory path of Path. Returns
          DirSelf + DirSeparator if Path contains no directory. }
        DirFromPath => JustPathName,

        { Return just the file name part without extension of Path.
          Empty if Path contains no file name. }
        NameFromPath => JustFileName,

        { Return just the extension of Path. Empty if Path contains
          no extension. }
        ExtFromPath => JustExtension,

        { Return the full pathname of Path }
        FExpand => FullPathName,

        { Add a DirSeparator to the end of s if there is not
          already one. }
        ForceAddDirSeparator => AddBackSlash,

        { Return a string stripped of leading spaces }
        TrimLeftStr => TrimLead,

        { Return a string stripped of trailing spaces }
        TrimRightStr => TrimTrail,

        { Return a string stripped of leading and trailing spaces }
        TrimBothStr => Trim,

        { Convert a string to lowercase }
        LoCaseStr => StLoCase,

        { Convert a string to uppercase }
        UpCaseStr => StUpCase
       );

import GPC;
```

```
{ Replace all occurences of OldC with NewC in s and return the
  result }
function  ReplaceChar (const s: String; OldC, NewC: Char) = Res:
  TString;

{ Break a string into 2 parts, using Ch as a marker }
function  BreakStr (const Src: String; var Dest1, Dest2: String; ch:
  Char): Boolean; attribute (ignorable);

{ Convert a CString to an Integer }
function  PChar2Int (s: CString) = i: Integer;

{ Convert a CString to a LongInt }
function  PChar2Long (s: CString) = i: LongInt;

{ Convert a CString to a Double }
function  PChar2Double (s: CString) = x: Double;

{ Search for s as an executable in the path and return its location
  (full pathname) }
function  PathLocate (const s: String): TString;

{ Copy file Src to Dest and return the number of bytes written }
function  CopyFile (const Src, Dest: String; BufSize: Integer):
  LongInt; attribute (ignorable);

{ Copy file Src to Dest and return the number of bytes written;
  report the number of bytes written versus total size of the source
  file }
function  CopyFileEx (const Src, Dest: String; BufSize: Integer;
  function Report (Reached, Total: LongInt): LongInt) = BytesCopied:
  LongInt; attribute (ignorable);

{ Turbo Power compatibility }

{ Execute the program prog. Dummy1 and Dummy2 are for compatibility
  only; they are ignored. }
function  ExecDos (const Prog: String; Dummy1: Boolean; Dummy2:
  Pointer): Integer; attribute (ignorable);

{ Return whether Src exists in the path as an executable -- if so
  return its full location in Dest }
function  ExistOnPath (const Src: String; var Dest: String) =
  Existing: Boolean;

{ Change the extension of s to Ext (do not include the dot!) }
function  ForceExtension (const s, Ext: String) = Res: TString;

{ Convert Integer to PChar; uses NewCString to allocate memory for
  the result, so you must call StrDispose to free the memory later }
```

```
function  Int2PChar (i: Integer): PChar;

{ Convert Integer to string }
function  Int2Str (i: Integer) = s: TString;

{ Convert string to Integer }
function  Str2Int (const s: String; var i: Integer): Boolean;
  attribute (ignorable);

{ Convert string to LongInt }
function  Str2Long (const s: String; var i: LongInt): Boolean;
  attribute (ignorable);

{ Convert string to Double }
function  Str2Real (const s: String; var i: Double): Boolean;
  attribute (ignorable);

{ Return a string right-padded to length Len with ch }
function  PadCh (const s: String; ch: Char; Len: Integer) = Padded:
  TString;

{ Return a string right-padded to length Len with spaces }
function  Pad (const s: String; Len: Integer): TString;

{ Return a string left-padded to length Len with ch }
function  LeftPadCh (const s: String; ch: Char; Len: Byte) = Padded:
  TString;

{ Return a string left-padded to length Len with blanks }
function  LeftPad (const s: String; Len: Integer): TString;

{ Uniform access to big memory blocks for GPC and BP. Of course, for
  programs that are meant only for GPC, you can use the usual
  New/Dispose routines. But for programs that should compile with
  GPC and BP, you can use the following routines for GPC. In the GPC
  unit for BP (gpc-bp.pas), you can find emulations for BP that try
  to provide access to as much memory as possible, despite the
  limitations of BP. The drawback is that this memory cannot be used
  freely, but only with the following moving routines. }

type
  PBigMem = ^TBigMem;
  TBigMem (MaxNumber: SizeType) = record
    { Public fields }
    Number, BlockSize: SizeType;
    Mappable: Boolean;
    { Private fields }
    Pointers: array [1 .. Max (1, MaxNumber)] of ^Byte
  end;

{ Note: the number of blocks actually allocated may be smaller than
```

```
  WantedNumber. Check the Number field of the result. }
function  AllocateBigMem (WantedNumber, aBlockSize: SizeType;
  WantMappable: Boolean) = p: PBigMem;
procedure DisposeBigMem (p: PBigMem);
procedure MoveToBigMem (var Source; p: PBigMem; BlockNumber:
  SizeType);
procedure MoveFromBigMem (p: PBigMem; BlockNumber: SizeType; var
  Dest);
{ Maps a big memory block into normal addressable memory and returns
  its address. The memory must have been allocated with
  WantMappable = True. The mapping is only valid until the next
  MapBigMem call. }
function  MapBigMem (p: PBigMem; BlockNumber: SizeType): Pointer;
```

6.15.7 Primitive heap checking

The following listing contains the interface of the HeapMon unit.

This unit provide a rather primitive means to watch the heap, i.e. check if all pointers that were allocated are released again. This is meant as a debugging help for avoiding memory leaks.

More extensive heap checking is provided by libraries like 'efence' which can be used in GPC programs without special provisions.

```
{ A unit to watch the heap, i.e. check if all pointers that were
  allocated are released again. It is meant as a debugging help to
  detect memory leaks.

  Use it in the main program before all other units. When, at the
  end of the program, some pointers that were allocated, have not
  been released, the unit writes a report to StdErr or another file
  (see below). Only pointers allocated via the Pascal mechanisms
  (New, GetMem) are tracked, not pointers allocated with direct libc
  calls or from C code. After a runtime error, pointers are not
  checked.

  Note that many units and libraries allocate memory for their own
  purposes and don't always release it at the end. Therefore, the
  usefulness of this unit is rather limited.

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit HeapMon;

interface

uses GPC;

{ This unit is automatically activated when used. The following
  declarations are only needed for special purposes. }

{ The report generated at the end can be redirected to a certain
  file by pointing HeapMonOutput to it. If not set, the report will
  be printed to the error messages file given with `--gpc-rts'
  options if given, and StdErr otherwise. }
var
  HeapMonOutput: ^Text = nil;

{ HeapMonReport can be used to print a report on non-released memory
  blocks at an arbitrary point during a program run to the file f.
  It is invoked automatically at the end, so usually you don't have
  to call it. Returns True if any non-released blocks were found,
  False otherwise. }
function HeapMonReport (var f: Text; DoRestoreTerminal: Boolean) =
  Res: Boolean; attribute (ignorable);
```

6.15.8 Internationalization

The following listing contains the interface of the Intl unit.

This unit provides national language support via locales and '.mo' files.

```
{ Welcome to the wonderful world of
  INTERNATIONALIZATION (i18n).
```

```
  This unit provides the powerful mechanism of national language
  support by accessing `.mo' files and locale information.

  It includes:
    locales (not xlocales) and libintl.

  See documentation for gettext (`info gettext') for details.

  Because GPC can deal with both CStrings and Pascal Strings, there
  is an interface for both types of arguments and function results
  with slightly different names.

  E.g. for Pascal strings:

    function GetText (const MsgId: String): TString;

  And the same as above, but with a C interface:

    function GetTextC (MsgId: CString): CString;

  `PLConv' in Pascal is very different from `struct lconv *' in C.
  Element names do not have underscores and have sometimes different
  sizes. The conversion is done automatically and has correct
  results.

  Furthermore, we have a tool similar to `xgettext' to extract all
  strings out of a Pascal source. It extracts the strings and writes
  a complete `.po' file to a file. See
  http://www.gnu-pascal.de/contrib/eike/
  The filename is pas2po-VERSION.tar.gz.

  Copyright (C) 2001-2005 Free Software Foundation, Inc.

  Author: Eike Lange <eike.lange@uni-essen.de>

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU Library General Public License as
  published by the Free Software Foundation, version 2.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  Library General Public License for more details.

  You should have received a copy of the GNU Library General Public
  License along with this library; see the file COPYING.LIB. If not,
  write to the Free Software Foundation, Inc., 59 Temple Place -
  Suite 330, Boston, MA 02111-1307, USA. }

{$gnu-pascal,I-}
```

```
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit Intl;

interface

uses GPC;

type
  IntlString = String (16);

  { Pascal translation from OrigLConv in intlc.c }
  PLConv = ^TLConv;
  TLConv = record
    { Numeric (non-monetary) information. }

    { Decimal point character. }
    DecimalPoint: IntlString;

    { Thousands separator. }
    ThousandsSep: IntlString;

    { Each element is the number of digits in each group;
      elements with higher indices are farther left.
      An element with value CharMax means that no further grouping
      is done.
      An element with value Chr (0) means that the previous element
      is used for all groups farther left. }
    Grouping: IntlString;

    { Monetary information. }

    { First three chars are a currency symbol from ISO 4217.
      Fourth char is the separator.  Fifth char is Chr (0). }
    IntCurrSymbol: IntlString;

    { Local currency symbol. }
    CurrencySymbol: IntlString;

    { Decimal point character. }
    MonDecimalPoint: IntlString;

    { Thousands separator. }
    MonThousandsSep: IntlString;

    { Like 'Grouping' element (above). }
    MonGrouping: IntlString;

    { Sign for positive values. }
```

```
    PositiveSign: IntlString;

    { Sign for negative values. }
    NegativeSign: IntlString;

    { Int'l fractional digits. }
    IntFracDigits: ByteInt;

    { Local fractional digits. }
    FracDigits: ByteInt;

    { 1 if CurrencySymbol precedes a positive value, 0 if it
      succeeds. }
    PCSPrecedes: ByteInt;

    { 1 iff a space separates CurrencySymbol from a positive
      value. }
    PSepBySpace: ByteInt;

    { 1 if CurrencySymbol precedes a negative value, 0 if it
      succeeds. }
    NCSPrecedes: ByteInt;

    { 1 iff a space separates CurrencySymbol from a negative
      value. }
    NSepBySpace: ByteInt;

    { Positive and negative sign positions:
      0 Parentheses surround the quantity and CurrencySymbol.
      1 The sign string precedes the quantity and CurrencySymbol.
      2 The sign string follows the quantity and CurrencySymbol.
      3 The sign string immediately precedes the CurrencySymbol.
      4 The sign string immediately follows the CurrencySymbol. }
    PSignPosn,
    NSignPosn: ByteInt;
  end;

{ Please do not assign anything to these identifiers! }
var
  LC_CTYPE:    CInteger; external name '_p_LC_CTYPE';
  LC_NUMERIC:  CInteger; external name '_p_LC_NUMERIC';
  LC_TIME:     CInteger; external name '_p_LC_TIME';
  LC_COLLATE:  CInteger; external name '_p_LC_COLLATE';
  LC_MONETARY: CInteger; external name '_p_LC_MONETARY';
  LC_MESSAGES: CInteger; external name '_p_LC_MESSAGES';
  LC_ALL:      CInteger; external name '_p_LC_ALL';
  CharMax:     Char; external name '_p_CHAR_MAX';

{@section Locales }

{ Set and/or return the current locale. }
```

```
function SetLocale (Category: Integer; const Locale: String):
  TString; attribute (ignorable);

{ Set and/or return the current locale. Same as above, but returns
  a CString. }
function SetLocaleC (Category: Integer; const Locale: String):
  CString; attribute (ignorable);

{ Return the numeric/monetary information for the current locale.
  The result is allocated from the heap. You can Dispose it when
  you don't need it anymore. }
function LocaleConv: PLConv;

{@section GetText }

{ Look up MsgId in the current default message catalog for the
  current LC_MESSAGES locale.  If not found, returns MsgId itself
  (the default text). }
function GetText (const MsgId: String): TString;

{ Same as above, but with a C interface }
function GetTextC (MsgId: CString): CString;

{ Look up MsgId in the DomainName message catalog for the current
  LC_MESSAGES locale. }
function DGetText (const DomainName, MsgId: String): TString;

{ Same as above, but with a C interface }
function DGetTextC (DomainName, MsgId: CString): CString;

{ Look up MsgId in the DomainName message catalog for the current
  Category locale. }
function DCGetText (const DomainName, MsgId: String; Category:
  Integer): TString;

{ Same as above, but with a C interface }
function DCGetTextC (DomainName, MsgId: CString; Category: Integer):
  CString;

{ Set the current default message catalog to DomainName.
  If DomainName is empty, reset to the default of 'messages'. }
function TextDomain (const DomainName: String): TString; attribute
  (ignorable);

{ Same as above, but with a C interface.
  If DomainName is nil, return the current default. }
function TextDomainC (DomainName: CString): CString; attribute
  (ignorable);

{ Specify that the DomainName message catalog will be found
  in DirName rather than in the system locale data base. }
```

```
function BindTextDomain (const DomainName, DirName: String):
  TString; attribute (ignorable);

{ Same as above, but with a C interface }
function BindTextDomainC (DomainName, DirName: CString): CString;
  attribute (ignorable);
```

6.15.9 'MD5' Message Digests

The following listing contains the interface of the MD5 unit.

This unit provides functions to compute 'MD5' message digest of files or memory blocks, according to the definition of 'MD5' in *RFC 1321* from April 1992.

```
{ Functions to compute MD5 message digest of files or memory blocks,
  according to the definition of MD5 in RFC 1321 from April 1992.

  Copyright (C) 1995, 1996, 2000-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  Based on the C code written by Ulrich Drepper
  <drepper@gnu.ai.mit.edu>, 1995 as part of glibc.

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU Library General Public License as
  published by the Free Software Foundation, version 2.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  Library General Public License for more details.

  You should have received a copy of the GNU Library General Public
  License along with this library; see the file COPYING.LIB. If not,
  write to the Free Software Foundation, Inc., 59 Temple Place -
  Suite 330, Boston, MA 02111-1307, USA. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit MD5;

interface

uses GPC;

{ Representation of a MD5 value. It is always in little endian byte
```

```
  order and therefore portable. }
type
  Card8 = Cardinal attribute (Size = 8);
  TMD5 = array [1 .. 16] of Card8;

const
  MD5StrLength = 2 * High (TMD5);

type
  MD5String = String (MD5StrLength);

{ Computes MD5 message digest for Length bytes in Buffer. }
procedure MD5Buffer (const Buffer; Length: SizeType; var MD5: TMD5);

{ Computes MD5 message digest for the contents of the file f. }
procedure MD5File (var f: File; var MD5: TMD5); attribute
  (iocritical);

{ Initializes a MD5 value with zeros. }
procedure MD5Clear (var MD5: TMD5);

{ Compares two MD5 values for equality. }
function MD5Compare (const Value1, Value2: TMD5): Boolean;

{ Converts an MD5 value to a string. }
function MD5Str (const MD5: TMD5) = s: MD5String;

{ Converts a string to an MD5 value. Returns True if successful. }
function MD5Val (const s: String; var MD5: TMD5): Boolean;

{ Composes two MD5 values to a single one. }
function MD5Compose (const Value1, Value2: TMD5) = Dest: TMD5;
```

6.15.10 BP compatibility: Overlay

The following listing contains the interface of the Overlay unit.

This is just a dummy replacement for BP's '`Overlay`' unit, since GPC doesn't need overlays.

```
{ Dummy BP compatible overlay unit for GPC

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.
```

```
  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

unit Overlay;

interface

const
  OvrOK = 0;
  OvrError = -1;
  OvrNotFound = -2;
  OvrNoMemory = -3;
  OvrIOError = -4;
  OvrNoEMSDriver = -5;
  OvrNoEMSMemory = -6;

const
  OvrEmsPages: Word = 0;
  OvrTrapCount: Word = 0;
  OvrLoadCount: Word = 0;
  OvrFileMode: Byte = 0;

type
  OvrReadFunc = function (OvrSeg: Word): Integer;

var
  OvrReadBuf: OvrReadFunc;
  OvrResult: Integer = 0;

procedure OvrInit (aFileName: String);
procedure OvrInitEMS;
procedure OvrSetBuf (Size: LongInt);
```

```
function  OvrGetBuf: LongInt;
procedure OvrSetRetry (Size: LongInt);
function  OvrGetRetry: LongInt;
procedure OvrClearBuf;
```

6.15.11 Start a child process, connected with pipes, also on Dos

The following listing contains the interface of the Pipes unit.

This unit provides routines to start a child process and write to/read from its Input/Output/StdErr via pipes. All of this is emulated transparently under Dos as far as possible.

```
{ Piping data from and to processes

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

{ Keep this consistent with the one in pipesc.c }
{$if defined (MSDOS) or defined (__MINGW32__)}
{$define NOFORK}
{$endif}
```

```
unit Pipes;

interface

uses GPC;

const
  PipeForking = {$ifdef NOFORK} False {$else} True {$endif};

type
  TProcedure = procedure;

  PWaitPIDResult = ^TWaitPIDResult;
  TWaitPIDResult = (PIDNothing, PIDExited, PIDSignaled, PIDStopped,
  PIDUnknown);

  PPipeProcess = ^TPipeProcess;
  TPipeProcess = record
    PID      : Integer;         { Process ID of process forked }
    SignalPID: Integer;         { Process ID to send the signal to.
                                  Equals PID by default }
    OpenPipes: Integer;         { Number of pipes to/from the
                                  process, for internal use }
    Signal   : Integer;         { Send this signal (if not 0) to the
                                  process after all pipes have been
                                  closed after some time }
    Seconds  : Integer;         { Wait so many seconds before
                                  sending the signal if the process
                                  has not terminated by itself }
    Wait     : Boolean;         { Wait for the process, even longer
                                  than Seconds seconds, after
                                  sending the signal (if any) }
    Result   : PWaitPIDResult;  { Default nil. If a pointer to a
                                  variable is stored here, its
                                  destination will contain the
                                  information whether the process
                                  terminated by itself, or was
                                  terminated or stopped by a signal,
                                  when waiting after closing the
                                  pipes }
    Status   : ^Integer;        { Default nil. If a pointer to a
                                  variable is stored here, its
                                  destination will contain the exit
                                  status if the process terminated
                                  by itself, or the number of the
                                  signal otherwise, when waiting
                                  after closing the pipes }
  end;

var
```

```
  { Default values for TPipeProcess records created by Pipe }
  DefaultPipeSignal : Integer = 0;
  DefaultPipeSeconds: Integer = 0;
  DefaultPipeWait   : Boolean = True;

{ The procedure Pipe starts a process whose name is given by
  ProcessName, with the given parameters (can be Null if no
  parameters) and environment, and create pipes from and/or to the
  process' standard input/output/error. ProcessName is searched for
  in the PATH with FSearchExecutable. Any of ToInputFile,
  FromOutputFile and FromStdErrFile can be Null if the corresponding
  pipe is not wanted. FromOutputFile and FromStdErrFile may be
  identical, in which case standard output and standard error are
  redirected to the same pipe. The behaviour of other pairs of files
  being identical is undefined, and useless, anyway. The files are
  Assigned and Reset or Rewritten as appropriate. Errors are
  returned in IOResult. If Process is not Null, a pointer to a
  record is stored there, from which the PID of the process created
  can be read, and by writing to which the action after all pipes
  have been closed can be changed. (The record is automatically
  Dispose'd of after all pipes have been closed.) If automatic
  waiting is turned off, the caller should get the PID from the
  record before it's Dispose'd of, and wait for the process sometime
  in order to avoid zombies. If no redirections are performed (i.e.,
  all 3 files are Null), the caller should wait for the process with
  WaitPipeProcess. When an error occurs, Process is not assigned to,
  and the state of the files is undefined, so be sure to check
  IOResult before going on.

  ChildProc, if not nil, is called in the child process after
  forking and redirecting I/O, but before executing the new process.
  It can even be called instead of executing a new process
  (ProcessName can be empty then).

  The procedure even works under Dos, but, of course, in a limited
  sense: if ToInputFile is used, the process will not actually be
  started until ToInputFile is closed. Signal, Seconds and Wait of
  TPipeProcess are ignored, and PID and SignalPID do not contain a
  Process ID, but an internal value without any meaning to the
  caller. Result will always be PIDExited. So, Status is the only
  interesting field (but Result should also be checked). Since there
  is no forking under Dos, ChildProc, if not nil, is called in the
  main process before spawning the program. So, to be portable, it
  should not do any things that would influence the process after
  the return of the Pipe function.

  The only portable way to use "pipes" in both directions is to call
  `Pipe', write all the Input data to ToInputFile, close
  ToInputFile, and then read the Output and StdErr data from
  FromOutputFile and FromStdErrFile. However, since the capacity of
  pipes is limited, one should also check for Data from
```

```
  FromOutputFile and FromStdErrFile (using CanRead, IOSelect or
  IOSelectRead) while writing the Input data (under Dos, there
  simply won't be any data then, but checking for data doesn't do
  any harm). Please see pipedemo.pas for an example. }
procedure Pipe (var ToInputFile, FromOutputFile, FromStdErrFile:
  AnyFile; const ProcessName: String; protected var Parameters:
  TPStrings; ProcessEnvironment: PCStrings; var Process:
  PPipeProcess; ChildProc: TProcedure); attribute (iocritical);

{ Waits for a process created by Pipe as determined in the Process
  record. (Process is Dispose'd of afterwards.) Returns True if
  successful. }
function WaitPipeProcess (Process: PPipeProcess): Boolean; attribute
  (ignorable);

{ Alternative interface from PExecute }

const
  PExecute_First   = 1;
  PExecute_Last    = 2;
  PExecute_One     = PExecute_First or PExecute_Last;
  PExecute_Search  = 4;
  PExecute_Verbose = 8;

{ PExecute: execute a chain of processes.

  Program and Arguments are the arguments to execv/execvp.

  Flags and PExecute_Search is non-zero if $PATH should be searched.
  Flags and PExecute_First is nonzero for the first process in
  chain. Flags and PExecute_Last is nonzero for the last process in
  chain.

  The result is the pid on systems like Unix where we fork/exec and
  on systems like MS-Windows and OS2 where we use spawn. It is up to
  the caller to wait for the child.

  The result is the exit code on systems like MSDOS where we spawn
  and wait for the child here.

  Upon failure, ErrMsg is set to the text of the error message,
  and -1 is returned. `errno' is available to the caller to use.

  PWait: cover function for wait.

  PID is the process id of the task to wait for. Status is the
  `status' argument to wait. Flags is currently unused (allows
  future enhancement without breaking upward compatibility). Pass 0
  for now.

  The result is the process ID of the child reaped, or -1 for
```

```
  failure.

  On systems that don't support waiting for a particular child, PID
  is ignored. On systems like MSDOS that don't really multitask
  PWait is just a mechanism to provide a consistent interface for
  the caller. }
function  PExecute (ProgramName: CString; Arguments: PCStrings; var
  ErrMsg: String; Flags: Integer): Integer; attribute (ignorable);
function  PWait (PID: Integer; var Status: Integer; Flags: Integer):
  Integer; attribute (ignorable);
```

6.15.12 BP compatibility (partly): 'Port', 'PortW' arrays

The following listing contains the interface of the Ports unit.

This unit provides access routines for the hardware ports on the IA32, as a partial replacement for BP's 'Port' and 'PortW' pseudo arrays.

Since port access is platform-specific, this unit cannot be used in code intended to be portable. Even on the IA32, its use can often be avoided – e.g. Linux provides a number of 'ioctl' functions, and DJGPP provides some routines to achieve things that would require port access under BP. Therefore, it is recommended to avoid using this unit whenever possible.

```
{ Access functions for I/O ports for GPC on an IA32 platform. This
  unit is *not* portable. It works only on IA32 platforms (tested
  under Linux and DJGPP). It is provided here only to serve as a
  replacement for BP's Port and PortW pseudo arrays.

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
```

```
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}
{$ifndef __i386__}
{$error The Ports unit is only for the IA32 platform}
{$endif}

unit Ports;

interface

{ Port access functions }
function  InPortB  (PortNumber: ShortWord): Byte;
function  InPortW  (PortNumber: ShortWord): ShortWord;
procedure OutPortB (PortNumber: ShortWord; aValue: Byte);
procedure OutPortW (PortNumber, aValue: ShortWord);

{ libc functions for getting access to the ports -- only for root
  processes, of course -- and to give up root privileges after
  getting access to the ports for setuid root programs. Dummies
  under DJGPP. }
{$ifdef MSDOS}
function  IOPerm (From, Num: MedCard; On: Integer): Integer;
  attribute (name = 'ioperm');
function  IOPL (Level: Integer): Integer; attribute (name = 'iopl');
function  SetEUID (EUID: Integer): Integer; attribute (name
  = 'seteuid');
{$else}
function  IOPerm (From, Num: MedCard; On: Integer): Integer;
  external name 'ioperm';
function  IOPL (Level: Integer): Integer; external name 'iopl';
function  SetEUID (EUID: Integer): Integer; external name 'seteuid';
{$endif}
```

6.15.13 BP compatibility: Printer, portable

The following listing contains the interface of the Printer unit.

This unit provides printer access, compatible to BP's '`Printer`' unit, for Dos (using printer devices) and Unix systems (using printer utilities).

For BP compatibility, the variable '`Lst`' is provided, but for newly written programs, it is recommended to use the '`AssignPrinter`' procedure on a text file, and close the file when done (thereby committing the printer job). This method allows for sending multiple printer jobs in the same program.

```
{ BP compatible printer unit with extensions

  Copyright (C) 1998-2005 Free Software Foundation, Inc.
```

```
  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit Printer;

interface

uses GPC {$ifndef __OS_DOS__}, Pipes {$endif};

var
  { Dos-like systems: writing to a printer device }

  { The file name to write printer output into }
  PrinterDeviceName: PString = @'prn';

  { Unix-like systems: printing via a printer program }

  { The file name of the printer program. If it contains a '/', it
    will be taken as a complete path, otherwise the file name will
    be searched for in the PATH with FSearchExecutable. }
  PrinterCommand: PString = @'lpr';
```

```
  { Optional command line parameters for the printer program.
    Ignored when nil. }
  PrinterArguments: PPStrings = nil;

  { How to deal with the printer spooler after the printer pipe is
    closed, cf. the Pipes unit. }
  PrinterPipeSignal : Integer = 0;
  PrinterPipeSeconds: Integer = 0;
  PrinterPipeWait   : Boolean = True;

{ Text file opened to default printer }
var
  Lst: Text;

{ Assign a file to the printer. Lst will be assigned to the default
  printer at program start, but other files can be assigned to the
  same or other printers (possibly after changing the variables
  above). SpoolerOutput, if not Null, will be redirected from the
  printer spooler's standard output and error. If you use this, note
  that a deadlock might arise when trying to write data to the
  spooler while its output is not being read, though this seems
  quite unlikely, since most printer spoolers don't write so much
  output that could fill a pipe. Under Dos, where no spooler is
  involved, SpoolerOutput, if not Null, will be reset to an empty
  file for compatibility. }
procedure AssignPrinter (var f: AnyFile; var SpoolerOutput:
  AnyFile);
```

6.15.14 Regular Expression matching and substituting

The following listing contains the interface of the RegEx unit.

This unit provides routines to match strings against regular expressions and perform substitutions using matched subexpressions. Regular expressions are strings with some characters having special meanings. They describe (match) a class of strings. They are similar to wild cards used in file name matching, but much more powerful.

To use this unit, you will need the '`rx`' library which can be found in `http://www.gnu-pascal.de/libs/`.

```
{$nested-comments}

{ Regular expression matching and replacement

  The RegEx unit provides routines to match strings against regular
  expressions and perform substitutions using matched
  subexpressions.

  To use the RegEx unit, you will need the rx library which can be
  found in http://www.gnu-pascal.de/libs/

  Regular expressions are strings with some characters having
  special meanings. They describe (match) a class of strings. They
```

```
are similar to wild cards used in file name matching, but much
more powerful.

There are two kinds of regular expressions supported by this unit,
basic and extended regular expressions. The difference between
them is not functionality, but only syntax. The following is a
short overview of regular expressions. For a more thorough
explanation see the literature, or the documentation of the rx
library, or man pages of programs like grep(1) and sed(1).

Basic            Extended         Meaning
‘.’              ‘.’              matches any single character
‘[aei-z]’        ‘[aei-z]’        matches either ‘a’, ‘e’, or any
                                  character from ‘i’ to ‘z’
‘[^aei-z]’       ‘[^aei-z]’       matches any character but ‘a’,
                                  ‘e’, or ‘i’ .. ‘z’
                                  To include in such a list the the
                                  characters ‘]’, ‘^’, or ‘-’, put
                                  them first, anywhere but first, or
                                  first or last, resp.
‘[[:alnum:]]’    ‘[[:alnum:]]’    matches any alphanumeric character
‘[^[:digit:]]’   ‘[^[:digit:]]’   matches anything but a digit
‘[a[:space:]]’   ‘[a[:space:]]’   matches the letter ‘a’ or a space
                                  character (space, tab)
...                               (there are more classes available)
‘\w’             ‘\w’             = [[:alnum:]]
‘\W’             ‘\W’             = [^[:alnum:]]
‘^’              ‘^’              matches the empty string at the
                                  beginning of a line
‘$’              ‘$’              matches the empty string at the
                                  end of a line
‘*’              ‘*’              matches zero or more occurences of
                                  the preceding expression
‘\+’             ‘+’              matches one or more occurences of
                                  the preceding expression
‘\?’             ‘?’              matches zero or one occurence of
                                  the preceding expression
‘\{N\}’          ‘{N}’            matches exactly N occurences of
                                  the preceding expression (N is an
                                  integer number)
‘\{M,N\}’        ‘{M,N}’          matches M to N occurences of the
                                  preceding expression (M and N are
                                  integer numbers, M <= N)
‘AB’             ‘AB’             matches A followed by B (A and B
                                  are regular expressions)
‘A\|B’           ‘A|B’            matches A or B (A and B are
                                  regular expressions)
‘\( \)’          ‘( )’            forms a subexpression, to override
                                  precedence, and for subexpression
                                  references
‘\7’             ‘\7’             matches the 7'th parenthesized
```

```
                                subexpression (counted by their
                                start in the regex), where 7 is a
                                number from 1 to 9 ;-).
                                *Please note:* using this feature
                                can be *very* slow or take very
                                much memory (exponential time and
                                space in the worst case, if you
                                know what that means ...).
‘\’            ‘\’              quotes the following character if
                                it’s special (i.e. listed above)
rest           rest             any other character matches itself

Precedence, from highest to lowest:
* parentheses (‘()’)
* repetition (‘*’, ‘+’, ‘?’, ‘{}’)
* concatenation
* alternation (‘|’)

When performing substitutions using matched subexpressions of a
regular expression (see ‘ReplaceSubExpressionReferences’), the
replacement string can reference the whole matched expression with
‘&’ or ‘\0’, the 7th subexpression with ‘\7’ (just like in the
regex itself, but using it in replacements is not slow), and the
7th subexpression converted to upper/lower case with ‘\u7’ or
‘\l7’, resp. (which also works for the whole matched expression
with ‘\u0’ or ‘\l0’). A verbatim ‘&’ or ‘\’ can be specified with
‘\&’ or ‘\\’, resp.

Copyright (C) 1998-2005 Free Software Foundation, Inc.

Author: Frank Heckenbach <frank@pascal.gnu.de>

This file is part of GNU Pascal.

GNU Pascal is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published
by the Free Software Foundation; either version 2, or (at your
option) any later version.

GNU Pascal is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU Pascal; see the file COPYING. If not, write to the
Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

As a special exception, if you link this file with files compiled
with a GNU compiler to produce an executable, this does not cause
```

```
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License.

  Please also note the license of the rx library. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit RegEx;

interface

uses GPC;

const
  { `BasicRegExSpecialChars' contains all characters that have
    special meanings in basic regular expressions.
    `ExtRegExSpecialChars' contains those that have special meanings
    in extended regular expressions. }
  BasicRegExSpecialChars = ['.', '[', ']', '^', '$', '*', '\'];
  ExtRegExSpecialChars   =
  ['.', '[', ']', '^', '$', '*', '+', '?', '{', '}', '|', '(', ')', '\'];

type
  { The type used by the routines of the `RegEx' unit to store
    regular expressions in an internal format. The fields RegEx,
    RegMatch, ErrorInternal, From and Length are only used
    internally. SubExpressions can be read after `NewRegEx' and will
    contain the number of parenthesized subexpressions. Error should
    be checked after `NewRegEx'. It will be `nil' when it succeeded,
    and contain an error message otherwise. }
  RegExType = record
    RegEx, RegMatch: Pointer;  { Internal }
    ErrorInternal: CString;    { Internal }
    From, Length: CInteger;    { Internal }
    SubExpressions: CInteger;
    Error: PString
  end;

{ Simple interface to regular expression matching. Matches a regular
  expression against a string starting from a specified position.
  Returns the position of the first match, or 0 if it does not
  match, or the regular expression is invalid. }
function  RegExPosFrom (const Expression: String; ExtendedRegEx,
  CaseInsensitive: Boolean; const s: String; From: Integer) =
  MatchPosition: Integer;
```

```
{ Creates the internal format of a regular expression. If
  ExtendedRegEx is True, Expression is assumed to denote an extended
  regular expression, otherwise a basic regular expression.
  CaseInsensitive determines if the case of letters will be ignored
  when matching the expression. If NewLines is True, `NewLine'
  characters in a string matched against the expression will be
  treated as dividing the string in multiple lines, so that `$' can
  match before the NewLine and `^' can match after. Also, `.' and
  `[^...]' will not match a NewLine then. }
procedure NewRegEx (var RegEx: RegExType; const Expression: String;
  ExtendedRegEx, CaseInsensitive, NewLines: Boolean);

{ Disposes of a regular expression created with `NewRegEx'. *Must*
  be used after `NewRegEx' before the RegEx variable becomes invalid
  (i.e., goes out of scope or a pointer pointing to it is Dispose'd
  of). }
procedure DisposeRegEx (var RegEx: RegExType); external
  name '_p_DisposeRegEx';

{ Matches a regular expression created with `NewRegEx' against a
  string. }
function  MatchRegEx (var RegEx: RegExType; const s: String;
  NotBeginningOfLine, NotEndOfLine: Boolean): Boolean;

{ Matches a regular expression created with `NewRegEx' against a
  string, starting from a specified position. }
function  MatchRegExFrom (var RegEx: RegExType; const s: String;
  NotBeginningOfLine, NotEndOfLine: Boolean; From: Integer):
  Boolean;

{ Finds out where the regular expression matched, if `MatchRegEx' or
  `MatchRegExFrom' were successful. If n = 0, it returns the
  position of the whole match, otherwise the position of the n'th
  parenthesized subexpression. MatchPosition and MatchLength will
  contain the position (counted from 1) and length of the match, or
  0 if it didn't match. (Note: MatchLength can also be 0 for a
  successful empty match, so check whether MatchPosition is 0 to
  find out if it matched at all.) MatchPosition or MatchLength may
  be Null and is ignored then. }
procedure GetMatchRegEx (var RegEx: RegExType; n: Integer; var
  MatchPosition, MatchLength: Integer);

{ Checks if the string s contains any quoted characters or
  (sub)expression references to the regular expression RegEx created
  with `NewRegEx'. These are `&' or `\0' for the whole matched
  expression (if OnlySub is not set) and `\1' .. `\9' for the n'th
  parenthesized subexpression. Returns 0 if it does not contain any,
  and the number of references and quoted characters if it does. If
  an invalid reference (i.e. a number bigger than the number of
  subexpressions in RegEx) is found, it returns the negative value
  of the (first) invalid reference. }
```

```
function  FindSubExpressionReferences (var RegEx: RegExType; const
  s: String; OnlySub: Boolean): Integer;

{ Replaces (sub)expression references in ReplaceStr by the actual
  (sub)expressions and unquotes quoted characters. To be used after
  the regular expression RegEx created with 'NewRegEx' was matched
  against s successfully with 'MatchRegEx' or 'MatchRegExFrom'. }
function  ReplaceSubExpressionReferences (var RegEx: RegExType;
  const s, ReplaceStr: String) = Res: TString;

{ Returns the string for a regular expression that matches exactly
  one character out of the given set. It can be combined with the
  usual operators to form more complex expressions. }
function  CharSet2RegEx (const Characters: CharSet) = s: TString;
```

6.15.15 BP compatibility: Strings

The following listing contains the interface of the Strings unit.

This is a compatibility unit to BP's 'Strings' unit to handle C style '#0'-terminated strings.

The same functionality and much more is available in the Run Time System, Section 6.14 [Run Time System], page 105, under clearer names (starting with a 'CString' prefix),

Moreover, the use of '#0'-terminated C-style strings ('PChar' or 'CString') is generally not recommended in GPC, since GPC provides ways to deal with Pascal-style strings of arbitrary and dynamic size in a comfortable way, as well as automatic conversion to C-style strings in order to call external C functions.

Therefore, using this unit is not recommended in newly written programs.

```
{ BP compatible Strings unit

  Copyright (C) 1999-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
```

```
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

module Strings;

export Strings = all (CStringLength => StrLen, CStringEnd => StrEnd,
                      CStringMove => StrMove, CStringCopy =>
  StrCopy,
                      CStringCopyEnd => StrECopy, CStringLCopy =>
  StrLCopy,
                      CStringCopyString => StrPCopy, CStringCat =>
  StrCat,
                      CStringLCat => StrLCat, CStringComp =>
  StrComp,
                      CStringCaseComp => StrIComp, CStringLComp =>
  StrLComp,
                      CStringLCaseComp => StrLIComp, CStringChPos =>
  StrScan,
                      CStringLastChPos => StrRScan, CStringPos =>
  StrPos,
                      CStringLastPos => StrRPos, CStringUpCase =>
  StrUpper,
                      CStringLoCase => StrLower, CStringIsEmpty =>
  StrEmpty,
                      CStringNew => StrNew);

import GPC;

function StrPas (aString: CString): TString;
procedure StrDispose (s: CString); external name '_p_Dispose';
```

6.15.16 Higher level string handling

The following listing contains the interface of the StringUtils unit.

This unit provides some routines for string handling on a higher level than those provided by the RTS.

```
{ Some routines for string handling on a higher level than those
  provided by the RTS.

  Copyright (C) 1999-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>
```

```
  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit StringUtils;

interface

uses GPC;

{ Various routines }

{ Appends Source to s, truncating the result if necessary. }
procedure AppendStr (var s: String; const Source: String);

{ Cuts s to MaxLength characters. If s is already MaxLength
  characters or shorter, it doesn't change anything. }
procedure StrCut (var s: String; MaxLength: Integer);

{ Returns the number of disjoint occurences of SubStr in s. Returns
  0 if SubStr is empty. }
function  StrCount (const SubStr: String; s: String): Integer;

{ Returns s, with all disjoint occurences of Source replaced by
  Dest. }
```

```
function  StrReplace (const s, Source, Dest: String) = Result:
  TString;

{ Sets of characters accepted for ‘True’ and ‘False’ by
  Char2Boolean and StrReadBoolean. }
var
  CharactersTrue : CharSet = ['Y', 'y'];
  CharactersFalse: CharSet = ['N', 'n'];

{ If ch is an element of CharactersTrue, Dest is set to True,
  otherwise if it is an element of CharactersFalse, Dest is set to
  False. In both cases True is returned. If ch is not an element of
  either set, Dest is set to False and False is returned. }
function  Char2Boolean (ch: Char; var Dest: Boolean): Boolean;
  attribute (ignorable);

{ Converts a digit character to its numeric value. Handles every
  base up to 36 (0 .. 9, a .. z, upper and lower case recognized).
  Returns -1 if the character is not a digit at all. If you want to
  use it for a base < 36, you have to check if the result is smaller
  than the base and not equal to -1. }
function  Char2Digit (ch: Char): Integer;

{ Encode a string in a printable format (quoted printable). All
  occurences of EscapeChar within the string are encoded. If
  QuoteHigh is True, all characters above the ASCII range are
  encoded as well (required in "7 bit" environments, as per several
  RFCs). ‘=’ is always encoded, as required for proper decoding, as
  are all characters below space (control characters), so if you
  don’t need an escape char yourself, you can pass #0 for
  EscapeChar. }
function  QuoteStringEscape (const s: String; EscapeChar: Char;
  QuoteHigh: Boolean): TString;

{ Encode a string in a printable format (quoted printable and
  surrounded with ‘"’). All occurences of ‘"’ within the string are
  encoded, so the result string contains exactly two ‘"’ characters
  (at the beginning and ending). This is useful to store arbitrary
  strings in text files while keeping them as readable as possible
  (which is the goal of the quoted printable encoding in general,
  see RFC 1521, section 5.1) and being able to read them back
  losslessly (with UnQuoteString). }
function  QuoteString (const s: String): TString;

{ Encode a string in a printable format suitable for StrReadEnum.
  All occurences of ‘,’ within the string are encoded. }
function  QuoteEnum (const s: String): TString;

{ Decode a string encoded by QuoteString (removing the ‘"’ and
  expanding quoted printable encoded characters). Returns True if
  successful and False if the string has an invalid form. A string
```

```
  returned by QuoteString is always valid. }
function  UnQuoteString (var s: String): Boolean; attribute
  (ignorable);

{ Decode a quoted-printable string (not enclosed in ‘"’, unlike for
  UnQuoteString). Returns True if successful and False if the string
  has an invalid form. In the latter case, it still decodes as much
  as is valid, even after the error position. }
function  UnQPString (var s: String): Boolean; attribute
  (ignorable);

{ Quotes a string as done in shells, i.e. all special characters are
  enclosed in either ‘"’ or ‘'’, where ‘"’, ‘$’ and ‘‘’ are always
  enclosed in ‘'’ and ‘'’ is always enclosed in ‘"’. }
function  ShellQuoteString (const s: String) = Res: TString;

{ Replaces all tab characters in s with the appropriate amount of
  spaces, assuming tab stops at every TabSize columns. Returns True
  if successful and False if the expanded string would exceed the
  capacity of s. In the latter case, some, but not all of the tabs
  in s may have been expanded. }
function  ExpandTabs (var s: String; TabSize: Integer): Boolean;
  attribute (ignorable);

{ Returns s, with all occurences of C style escape sequences (e.g.
  ‘\n’) replaced by the characters they mean. If AllowOctal is True,
  also octal character specifications (e.g. ‘\007’) are replaced. If
  RemoveQuoteChars is True, any other backslashes are removed (e.g.
  ‘\*’ -> ‘*’ and ‘\\’ -> ‘\’), otherwise they are kept, and also
  ‘\\’ is left as two backslashes then. }
function  ExpandCEscapeSequences (const s: String; RemoveQuoteChars,
  AllowOctal: Boolean) = r: TString;

{ Routines for TPStrings }

{ Initialise a TPStrings variable, allocate Size characters for each
  element. This procedure does not dispose of any previously
  allocated storage, so if you use it on a previously used variable
  without freeing the storage yourself, this might cause memory
  leaks. }
procedure AllocateTPStrings (var Strings: TPStrings; Size: Integer);

{ Clear all elements (set them to empty strings), does not free any
  storage. }
procedure ClearTPStrings (var Strings: TPStrings);

{ Divide a string into substrings, using Separators as separator. A
  single trailing separator is ignored. Further trailing separators
  as well as any leading separators and multiple separators in a row
  produce empty substrings. }
function TokenizeString (const Source: String; Separators: CharSet)
```

```
  = Res: PPStrings;

{ Divide a string into substrings, using SpaceCharacters as
  separators. The splitting is done according the usual rules of
  shells, using (and removing) single and double quotes and
  QuotingCharacter. Multiple, leading, and trailing separators are
  ignored. If there is an error, a message is stored in ErrMsg (if
  not Null), and nil is returned. nil is also returned (without an
  error message) if s is empty. }
function ShellTokenizeString (const s: String; var ErrMsg: String) =
  Tokens: PPStrings;

{ String parsing routines }

{ All the following StrReadFoo functions behave similarly. They read
  items from a string s, starting at index i, to a variable Dest.
  They skip any space characters (spaces and tabs) by incrementing i
  first. They return True if successful, False otherwise. i is
  incremented accordingly if successful, otherwise i is left
  unchanged, apart from the skipping of space characters, and Dest
  is undefined. This behaviour makes it easy to use the functions in
  a row like this:

    i := 1;
    if StrReadInt    (s, i, Size)  and StrReadComma (s, i) and
       StrReadQuoted (s, i, Name)  and StrReadComma (s, i) and
       ...
       StrReadReal   (s, i, Angle) and (i > Length (s)) then ...

  (The check `i > Length (s)' is in case you don't want to accept
  trailing "garbage".) }

{ Just skip any space characters as described above. }
procedure StrSkipSpaces (const s: String; var i: Integer);

{ Read a quoted string (as produced by QuoteString) from a string
  and unquote the result using UnQuoteString. It is considered
  failure if the result (unquoted) would be longer than the capacity
  of Dest. }
function  StrReadQuoted (const s: String; var i: Integer; var Dest:
  String): Boolean; attribute (ignorable);

{ Read a string delimited with Delimiter from a string and return
  the result with the delimiters removed. It is considered failure
  if the result (without delimiters) would be longer than the
  capacity of Dest. }
function  StrReadDelimited (const s: String; var i: Integer; var
  Dest: String; Delimiter: Char): Boolean; attribute (ignorable);

{ Read a word (consisting of anything but space characters and
  commas) from a string. It is considered failure if the result
```

```
  would be longer than the capacity of Dest. }
function  StrReadWord (const s: String; var i: Integer; var Dest:
  String): Boolean; attribute (ignorable);

{ Check that a certain string is contained in s (after possible
  space characters). }
function  StrReadConst (const s: String; var i: Integer; const
  Expected: String) = Res: Boolean; attribute (ignorable);

{ A simpler to use version of StrReadConst that expects a ‘,’. }
function  StrReadComma (const s: String; var i: Integer) = Res:
  Boolean; attribute (ignorable);

{ Read an integer number from a string. }
function  StrReadInt (const s: String; var i: Integer; var Dest:
  Integer): Boolean; attribute (ignorable);

{ Read a real number from a string. }
function  StrReadReal (const s: String; var i: Integer; var Dest:
  Real): Boolean; attribute (ignorable);

{ Read a Boolean value, represented by a single character
  from CharactersTrue or CharactersFalse (cf. Char2Boolean), from a
  string. }
function  StrReadBoolean (const s: String; var i: Integer; var Dest:
  Boolean): Boolean; attribute (ignorable);

{ Read an enumerated value, i.e., one of the entries of IDs, from a
  string, and stores the ordinal value, i.e., the index in IDs
  (always zero-based) in Dest. }
function  StrReadEnum (const s: String; var i: Integer; var Dest:
  Integer; const IDs: array of PString): Boolean; attribute
  (ignorable);

{ String hash table }

const
  DefaultHashSize = 1403;

type
  THash = Cardinal;

  PStrHashList = ^TStrHashList;
  TStrHashList = record
    Next: PStrHashList;
    s: PString;
    i: Integer;
    p: Pointer
  end;

  PStrHashTable = ^TStrHashTable;
```

```
  TStrHashTable (Size: Cardinal) = record
    CaseSensitive: Boolean;
    Table: array [0 .. Size - 1] of PStrHashList
  end;

function  HashString          (const s: String): THash;
function  NewStrHashTable     (Size: Cardinal; CaseSensitive:
  Boolean) = HashTable: PStrHashTable;
procedure AddStrHashTable     (HashTable: PStrHashTable; s: String;
  i: Integer; p: Pointer);
procedure DeleteStrHashTable  (HashTable: PStrHashTable; s: String);
function  SearchStrHashTable  (HashTable: PStrHashTable; const s:
  String; var p: Pointer): Integer;  { p may be Null }
procedure StrHashTableUsage   (HashTable: PStrHashTable; var
  Entries, Slots: Integer);
procedure DisposeStrHashTable (HashTable: PStrHashTable);
```

6.15.17 BP compatibility: System

The following listing contains the interface of the System unit.

This unit contains only BP's more exotic routines which are not recommended to be used in new programs. Most of their functionality can be achieved by more standard means already.

Note: 'MemAvail' and 'MaxAvail', provided in this unit, cannot easily be achieved by other means. However, it is not recommended to use them on any multi-tasking system at all, where memory is a shared resource. The notes in the unit give some hints about how to avoid using them.

On special request, i.e., by defining the conditionals '__BP_TYPE_SIZES__', '__BP_RANDOM__' and/or '__BP_PARAMSTR_0__', the unit also provides BP compatible integer type sizes, a 100% BP compatible pseudo random number generator and/or BP compatible 'ParamStr (0)' behaviour (the latter, however, only on some systems).

```
{ BP and partly Delphi compatible System unit for GPC

  This unit is released as part of the GNU Pascal project. It
  implements some rather exotic BP and Delphi compatibility
  features. Even many BP and Delphi programs don't need them, but
  they're here for maximum compatibility. Most of BP's and Delphi's
  System units' features are built into the compiler or the RTS.

  Note: The things in this unit are really exotic. If you haven't
  used BP or Delphi before, you don't want to look at this unit. :-)

  This unit depends on the conditional defines '__BP_TYPE_SIZES__',
  '__BP_RANDOM__', '__BP_PARAMSTR_0__' and '__BP_NO_ALLOCMEM__'.

  If '__BP_TYPE_SIZES__' is defined (with the '-D__BP_TYPE_SIZES__'
  option), the integer data types will be redefined to the sizes
  they have in BP or Delphi. Note that this might cause problems,
  e.g. when passing var parameters of integer types between units
  that do and don't use System. However, of the BP compatibility
  units, only Dos and WinDos use such parameters, and they have been
```

```
taken care of so they work.

If '__BP_RANDOM__' is defined ('-D__BP_RANDOM__'), this unit will
provide an exactly BP compatible pseudo random number generator.
In particular, the range for integer randoms will be truncated to
16 bits like in BP. The RandSeed variable is provided, and if it's
set to the same value as BP's RandSeed, it produces exactly the
same sequence of pseudo random numbers that BP's pseudo random
number generator does (whoever might need this ... ;-). Even the
Randomize function will behave exactly like in BP. However, this
will not be noted unless one explicitly tests for it.

If '__BP_PARAMSTR_0__' is defined ('-D__BP_PARAMSTR_0__'), this
unit will change the value of 'ParamStr (0)' to that of
'ExecutablePath', overwriting the value actually passed by the
caller, to imitate BP's/Dos's behaviour. However *note*: On most
systems, 'ExecutablePath' is *not* guaranteed to return the full
path, so defining this symbol doesn't change anything. In general,
you *cannot* expect to find the full executable path, so better
don't even try it, or your program will (at best) run on some
systems. For most cases where BP programs access their own
executable, there are cleaner alternatives available.

If '__BP_NO_ALLOCMEM__' is defined ('-D__BP_NO_ALLOCMEM__'), the
two Delphi compatible functions 'AllocMemCount' and 'AllocMemSize'
will not be provided. The advantage is that this unit will not
have to 'Mark' the heap which makes memory de-/allocations much
faster if the program doesn't use 'Mark' otherwise.

Copyright (C) 1998-2005 Free Software Foundation, Inc.

Authors: Peter Gerwinski <peter@gerwinski.de>
         Prof. Abimbola A. Olowofoyeku <African_Chief@bigfoot.com>
         Frank Heckenbach <frank@pascal.gnu.de>
         Dominik Freche <dominik.freche@gmx.net>

This file is part of GNU Pascal.

GNU Pascal is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published
by the Free Software Foundation; either version 2, or (at your
option) any later version.

GNU Pascal is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU Pascal; see the file COPYING. If not, write to the
Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
```

```
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

module System;

export System = all (FileMode {$ifdef __BP_TYPE_SIZES__},
  SystemInteger => Integer, SystemWord => Word {$endif});

import GPC (MaxLongInt => GPC_MaxLongInt);

var
  { Chain of procedures to be executed at the end of the program }
  ExitProc: ^procedure = nil;

  { Contains all the command line arguments passed to the program,
    concatenated, with spaces between them }
  CmdLine: CString;

  {$ifdef __BP_RANDOM__}
  { Random seed, initialized by Randomize, but can also be set
    explicitly }
  RandSeed: Integer attribute (Size = 32) = 0;
  {$endif}

type
  OrigInt = Integer;
  OrigWord = Word;

  { Delphi }
  SmallInt = Integer attribute (Size = 16);
  DWord    = Cardinal attribute (Size = 32);

  { Short BP compatible type sizes if wanted }
  {$ifdef __BP_TYPE_SIZES__}
  ByteBool      = Boolean attribute (Size = 8);
  WordBool      = Boolean attribute (Size = 16);
  LongBool      = Boolean attribute (Size = 32);
  ShortInt      = Integer attribute (Size = 8);
  SystemInteger = Integer attribute (Size = 16);
  LongInt       = Integer attribute (Size = 32);
```

```
  Comp         = Integer attribute (Size = 64);
  Byte         = Cardinal attribute (Size = 8);
  SystemWord   = Cardinal attribute (Size = 16);
  LongWord     = Cardinal attribute (Size = 32);  { Delphi }
  {$else}
  SystemInteger = Integer;
  SystemWord   = Word;
  {$endif}

  {$if False}  { @@ doesn't work well (dialec3.pas) -- when GPC gets
  short
                  strings, it will be unnecessary }
  {$ifopt borland-pascal}
  String = String [255];
  {$endif}
  {$endif}

const
  MaxInt     = High (SystemInteger);
  MaxLongInt = High (LongInt);

{ Return the lowest-order byte of x }
function  Lo (x: LongestInt): Byte;

{ Return the second-lowest-order byte of x }
function  Hi (x: LongestInt): Byte;

{ Swap the lowest-order and second-lowest-order bytes, mask out the
  higher-order ones }
function  Swap (x: LongestInt): SystemWord;

{ Store the current directory name (on the given drive number if
  drive <> 0) in s }
procedure GetDir (Drive: Byte; var s: String);

{ Dummy routine for compatibility. @@Use two overloaded versions
  rather than varargs when possible. }
procedure SetTextBuf (var f: Text; var Buf; ...);

{ Mostly useless BP compatible variables }
var
  SelectorInc: SystemWord = $1000;
  Seg0040: SystemWord = $40;
  SegA000: SystemWord = $a000;
  SegB000: SystemWord = $b000;
  SegB800: SystemWord = $b800;
  Test8086: Byte = 2;
  Test8087: Byte = 3;  { floating-point arithmetic is emulated
                         transparently by the OS if not present
                         in hardware }
  OvrCodeList: SystemWord = 0;
```

```
  OvrHeapSize: SystemWord = 0;
  OvrDebugPtr: Pointer = nil;
  OvrHeapOrg: SystemWord = 0;
  OvrHeapPtr: SystemWord = 0;
  OvrHeapEnd: SystemWord = 0;
  OvrLoadList: SystemWord = 0;
  OvrDosHandle: SystemWord = 0;
  OvrEmsHandle: SystemWord = $ffff;
  HeapOrg: Pointer absolute HeapLow;
  HeapPtr: Pointer absolute HeapHigh;
  HeapEnd: Pointer = Pointer (High (PtrCard));
  FreeList: Pointer = nil;
  FreeZero: Pointer = nil;
  StackLimit: SystemWord = 0;
  HeapList: SystemWord = 0;
  HeapLimit: SystemWord = 1024;
  HeapBlock: SystemWord = 8192;
  HeapAllocFlags: SystemWord = 2;
  CmdShow: SystemInteger = 0;
  SaveInt00: Pointer = nil;
  SaveInt02: Pointer = nil;
  SaveInt0C: Pointer = nil;
  SaveInt0D: Pointer = nil;
  SaveInt1B: Pointer = nil;
  SaveInt21: Pointer = nil;
  SaveInt23: Pointer = nil;
  SaveInt24: Pointer = nil;
  SaveInt34: Pointer = nil;
  SaveInt35: Pointer = nil;
  SaveInt36: Pointer = nil;
  SaveInt37: Pointer = nil;
  SaveInt38: Pointer = nil;
  SaveInt39: Pointer = nil;
  SaveInt3A: Pointer = nil;
  SaveInt3B: Pointer = nil;
  SaveInt3C: Pointer = nil;
  SaveInt3D: Pointer = nil;
  SaveInt3E: Pointer = nil;
  SaveInt3F: Pointer = nil;
  SaveInt75: Pointer = nil;
  RealModeRegs: array [0 .. 49] of Byte =
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0);

{ Mostly useless BP compatible pointer functions }
function  Ofs (const x): PtrWord;
function  Seg (const x): PtrWord;
function  Ptr (Seg, Ofs: PtrWord): Pointer;
function  CSeg: PtrWord;
function  DSeg: PtrWord;
```

```
function  SSeg: PtrWord;
function  SPtr: PtrWord;

{ Routines to handle BP's 6 byte `Real' type which is formatted like
  this:

  47                                                   0
  -|------- -------- -------- -------- --------|--------
   |                                           |
   +----------+                   +------------+
  47 Sign Bit |  8..46 Mantissa   | 0..7 Biased Exponent

  This format does not support infinities, NaNs and denormalized
  numbers. The first digit after the binary point is not stored and
  assumed to be 1. (This is called the normalized representation of
  a binary floating point number.)

  In GPC, this type is represented by the type `BPReal' which is
  binary compatible to BP's type, and can therefore be used in
  connection with binary files used by BP programs.

  The functions `RealToBPReal' and `BPRealToReal' convert between
  this type and GPC's `Real' type. Apart from that, `BPReal' should
  be treated as opaque.

  The variables `BPRealIgnoreOverflow' and `BPRealIgnoreUnderflow'
  determine what to do in the case of overflows and underflows. The
  default values are BP compatible. }

var
  { Ignore overflows, and use the highest possible value instead. }
  BPRealIgnoreOverflow: Boolean = False;

  { Ignore underflows, and use 0 instead. This is BP's behaviour,
    but has the disadvantage of diminishing computation precision. }
  BPRealIgnoreUnderflow: Boolean = True;

type
  BPRealInteral = Cardinal attribute (Size = 8);
  BPReal = packed record
    Format: packed array [1 .. 6] of BPRealInteral
  end;

function RealToBPReal (r: Real) = BR: BPReal;
function BPRealToReal (const BR: BPReal) = RealValue: Real;

{ Heap management stuff }

const
  { Possible results for HeapError }
  HeapErrorRunError = 0;
```

```
  HeapErrorNil      = 1;
  HeapErrorRetry    = 2;

var
  { If assigned to a function, it will be called when memory
    allocations do not find enough free memory. Its result
    determines if a run time error should be raised (the default),
    or nil should be returned, or the allocation should be retried
    (causing the routine to be called again if the allocation still
    doesn't succeed).

    Notes:

    - Returning nil can cause some routines of the RTS and units
      (shipped with GPC or third-party) to crash when they don't
      expect nil, so better don't use this mechanism, but rather
      CGetMem where needed.

    - Letting the allocation be retried, of course, only makes sense
      if the routine freed some memory before -- otherwise it will
      cause an infinite loop! So, a meaningful HeapError routine
      should dispose of some temporary objects, if available, and
      return HeapErrorRetry, and return HeapErrorRunError when no
      (more) of them are available. }
  HeapError: ^function (Size: SystemWord): SystemInteger = nil;

{ Just returns HeapErrorNil. When this function is assigned to
  HeapError, GetMem and New will return a nil pointer instead of
  causing a runtime error when the allocation fails. See the comment
  for HeapError above. }
function  HeapErrorNilReturn (Size: SystemWord): SystemInteger;

{ Return the total free memory/biggest free memory block. Except
  under Win32 and DJGPP, these are expensive routines -- try to
  avoid them. Under Win32, MaxAvail returns the same as MemAvail, so
  don't rely on being able to allocate a block of memory as big as
  MaxAvail indicates. Generally it's preferable to not use these
  functions at all in order to do a safe allocation, but just try to
  allocate the memory needed using CGetMem, and check for a nil
  result. What makes these routines unrealiable is, e.g., that on
  multi-tasking systems, another process may allocate memory after
  you've called MemAvail/MaxAvail and before you get to do the next
  allocation. Also, please note that some systems over-commit
  virtual memory which may cause MemAvail to return a value larger
  than the actual (physical plus swap) memory available. Therefore,
  if you want to be "sure" (modulo the above restrictions) that the
  memory is actually available, use MaxAvail. }
function  MemAvail: Cardinal;
function  MaxAvail: Cardinal;

{ Delphi compatibility }
```

```
function  CompToDouble (x: Comp): Double;
function  DoubleToComp (x: Double): Comp;
{$ifndef __BP_NO_ALLOCMEM__}
function  AllocMemCount = Count: SystemInteger;
function  AllocMemSize = Size: SizeType;
{$endif}
procedure Assert (Condition: Boolean);
procedure DefaultAssertErrorProc (const Message, FileName: String;
  LineNumber: SystemInteger; ErrorAddr: Pointer);

var
  AssertErrorProc: ^procedure (const Message, FileName: String;
  LineNumber: SystemInteger; ErrorAddr: Pointer) =
  @DefaultAssertErrorProc;
  NoErrMsg: Boolean = False;
```

6.15.18 Some text file tricks

The following listing contains the interface of the TFDD unit.

This unit provides some tricks with text files, e.g. a "tee" file which causes everything written to it to be written to two other files.

```
{ Some text file tricks.

  Copyright (C) 2002-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
```

```
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

unit TFDD;

interface

uses GPC;

{ Write to multiple files. Everything written to Dest after calling
  this procedure will be written to both File1 and File2. Can be
  chained. }
procedure MultiFileWrite (var Dest, File1, File2: AnyFile);
```

6.15.19 Trap runtime errors

The following listing contains the interface of the Trap unit.

This unit allows you to trap runtime errors, so a runtime error will not abort the program, but pass the control back to a point within the program. Use with care, and read the notes in the interface, please.

```
{ Trapping runtime errors

  The Trap unit allows you to trap runtime errors, so a runtime
  error will not abort the program, but pass the control back to a
  point within the program.

  The usage is simple. The TrapExec procedure can be called with a
  function (p) as an argument. p must take a Boolean argument. p
  will immediately be called with False given as its argument. When
  a runtime error would otherwise be caused while p is active, p
  will instead be called again with True as its argument. After p
  returns, runtime error trapping ends.

  When the program terminates (e.g. by reaching its end or by a Halt
  statement) and a runtime error was trapped during the run, Trap
  will set the ExitCode and ErrorAddr variables to indicate the
  trapped error.

  Notes:

  - After trapping a runtime error, your program might not be in a
    stable state. If the runtime error was a "minor" one (such as a
    range checking or arithmetic error), it should not be a problem.
    But if you, e.g., write a larger application and use Trap to
    prevent a sudden abort caused by an unexpected runtime error,
    you should make the program terminate regularly as soon as
```

```
    possible after a trapped error (perhaps by telling the user to
    save the data, then terminate the program and report the bug to
    you).

  - Since the trapping mechanism *jumps* back, it has all the
    negative effects that a (non-local!) ‘goto’ can have! You should
    be aware of the consequences of all active procedures being
    terminated at an arbitrary point!

  - Nested traps are supported, i.e. you can call TrapExec again
    within a routine called by another TrapExec instance. Runtime
    errors trapped within the inner TrapExec invocation will be
    trapped by the inner TrapExec, while runtime errors trapped
    after its termination will be trapped by the outer TrapExec
    again.

  Copyright (C) 1996-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030303}
{$error This unit requires GPC release 20030303 or newer.}
{$endif}

unit Trap;
```

```
interface

uses GPC;

var
  TrappedExitCode: Integer = 0;
  TrappedErrorAddr: Pointer = nil;
  TrappedErrorMessageString: TString = '';

{ Trap runtime errors. See the comment at the top. }
procedure TrapExec (procedure p (Trapped: Boolean));

{ Forget about saved errors from the innermost TrapExec instance. }
procedure TrapReset;
```

6.15.20 BP compatibility: Turbo3

The following listing contains the interface of the Turbo3 unit.

This is a compatibility unit to BP's 'Turbo3' compatibility unit to TP3. ;−) It is not meant to be used in any newly written code.

```
{ Turbo Pascal 3.0 compatibility unit

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }
```

```
{$gnu-pascal,I-}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

unit Turbo3;

interface

import GPC only (AssignTFDD);
       System (MemAvail => System_MemAvail,
               MaxAvail => System_MaxAvail);
       CRT (LowVideo  => CRT_LowVideo,
            HighVideo => CRT_HighVideo);

var
  Kbd: Text;
  CBreak: Boolean absolute CheckBreak;

procedure AssignKbd (var f: AnyFile);
function  MemAvail: Integer;
function  MaxAvail: Integer;
function  LongFileSize (var f: AnyFile): Real;
function  LongFilePos  (var f: AnyFile): Real;
procedure LongSeek     (var f: AnyFile; aPosition: Real);
procedure LowVideo;
procedure HighVideo;
```

6.15.21 BP compatibility: WinDos

The following listing contains the interface of the WinDos unit.

This is a portable implementation of most routines from BP's 'WinDos' unit. A few routines that are Dos – or even IA32 real mode – specific, are only available if '__BP_UNPORTABLE_ROUTINES__' is defined, Section 7.2 [BP Incompatibilities], page 237.

The same functionality and much more is available in the Run Time System, Section 6.14 [Run Time System], page 105. The RTS routines usually have different names and/or easier and less limiting interfaces (e.g. 'ReadDir' etc. vs. 'FindFirst' etc.), and are often more efficient.

Therefore, using this unit is not recommended in newly written programs.

```
{ Mostly BP compatible portable WinDos unit

  This unit supports most, but not all, of the routines and
  declarations of BP's WinDos unit.

  Notes:

  - The procedures GetIntVec and SetIntVec are not supported since
    they make only sense for Dos real-mode programs (and GPC
    compiled programs do not run in real-mode, even on IA32 under
    Dos). The procedures Intr and MsDos are only supported under
    DJGPP if '__BP_UNPORTABLE_ROUTINES__' is defined (with the
```

```
    '-D__BP_UNPORTABLE_ROUTINES__' option). A few other routines are
    also only supported with this define, but on all platforms (but
    they are crude hacks, that's why they are not supported without
    this define).

  - The internal structure of file variables (TFileRec and TTextRec)
    is different in GPC. However, as far as TFDDs are concerned,
    there are other ways to achieve the same in GPC, see the GPC
    unit.

  Copyright (C) 1998-2005 Free Software Foundation, Inc.

  Author: Frank Heckenbach <frank@pascal.gnu.de>

  This file is part of GNU Pascal.

  GNU Pascal is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published
  by the Free Software Foundation; either version 2, or (at your
  option) any later version.

  GNU Pascal is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with GNU Pascal; see the file COPYING. If not, write to the
  Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA
  02111-1307, USA.

  As a special exception, if you link this file with files compiled
  with a GNU compiler to produce an executable, this does not cause
  the resulting executable to be covered by the GNU General Public
  License. This exception does not however invalidate any other
  reasons why the executable file might be covered by the GNU
  General Public License. }

{$gnu-pascal,I-,maximum-field-alignment 0}
{$if __GPC_RELEASE__ < 20030412}
{$error This unit requires GPC release 20030412 or newer.}
{$endif}

module WinDos;

export WinDos = all (FCarry, FParity, FAuxiliary, FZero, FSign,
  FOverflow,
                     DosError, GetDate, GetTime, GetCBreak,
  SetCBreak,
                     GetVerify, SetVerify, DiskFree, DiskSize,
                     GetFAttr, SetFAttr, GetFTime, SetFTime,
```

```
                    UnpackTime, PackTime,
                    {$ifdef __BP_UNPORTABLE_ROUTINES__}
                    {$ifdef __GO32__}
                    Intr, MsDos,
                    {$endif}
                    DosVersion, SetDate, SetTime,
                    {$endif}
                    CStringGetEnv => GetEnvVar);

import GPC (MaxLongInt => GPC_Orig_MaxLongInt);
       System;
       Dos (FindFirst => Dos_FindFirst,
            FindNext  => Dos_FindNext,
            FindClose => Dos_FindClose);

const
  { File attribute constants }
  faReadOnly  = ReadOnly;
  faHidden    = Hidden;    { set for dot files except '.' and '..' }
  faSysFile   = SysFile;   { not supported }
  faVolumeID  = VolumeID;  { not supported }
  faDirectory = Directory;
  faArchive   = Archive;   { means: not executable }
  faAnyFile   = AnyFile;

  { Maximum file name component string lengths }
  fsPathName  = 79;
  fsDirectory = 67;
  fsFileName  = 8;
  fsExtension = 4;

  { FileSplit return flags }
  fcExtension = 1;
  fcFileName  = 2;
  fcDirectory = 4;
  fcWildcards = 8;

type
  PTextBuf = ^TTextBuf;
  TTextBuf = TextBuf;

  { Search record used by FindFirst and FindNext }
  TSearchRec = record
    Fill: SearchRecFill;
    Attr: Byte8;
    Time, Size: LongInt;
    Name: {$ifdef __BP_TYPE_SIZES__}
          packed array [0 .. 12] of Char
          {$else}
          TStringBuf
          {$endif};
```

```
    Reserved: SearchRec
  end;

  { Date and time record used by PackTime and UnpackTime }
  TDateTime = DateTime;

  { 8086 CPU registers -- only used by the unportable Dos routines }
  TRegisters = Registers;

{ FindFirst and FindNext are quite inefficient since they emulate
  all the brain-dead Dos stuff. If at all possible, the standard
  routines OpenDir, ReadDir and CloseDir (in the GPC unit) should be
  used instead. }
procedure FindFirst (Path: PChar; Attr: Word; var SR: TSearchRec);
procedure FindNext  (var SR: TSearchRec);
procedure FindClose (var SR: TSearchRec);
function  FileSearch (Dest, FileName, List: PChar): PChar;
function  FileExpand (Dest, FileName: PChar): PChar;
function  FileSplit (Path, Dir, BaseName, Ext: PChar) = Res: Word;
function  GetCurDir (Dir: PChar; Drive: Byte): PChar;
procedure SetCurDir (Dir: PChar);
procedure CreateDir (Dir: PChar);
procedure RemoveDir (Dir: PChar);
function  GetArgCount: Integer;
function  GetArgStr (Dest: PChar; ArgIndex: Integer; MaxLen: Word):
  PChar;
```

7 A QuickStart Guide from Borland Pascal to GNU Pascal.

This chapter is intended to be a QuickStart guide for programmers who are familiar with Borland Pascal.

Throughout the manual, we talk of "Borland Pascal" or "BP" for short, to refer to Borland Pascal version 7 for Dos protected mode. Other versions of Borland Pascal and Turbo Pascall don't differ too much, but this one was the very last Dos version Borland has published, so in most if not all cases, you can safely substitute the version you're familiar with.

"Borland Pascal" and "Turbo Pascal" are registered trademarks of Borland Inc.

7.1 BP Compatibility

GNU Pascal (GPC) is compatible to version 7 of Borland Pascal (BP) to a large extent and comes with portable replacements of the BP standard units.

However, BP is a 16-bit compiler while GPC is a 32/64-bit compiler, so the size of the '`Integer`' type, for instance, is 16 bits in BP, but at least 32 bits in GPC. If a BP program has been designed with portability in mind from the ground up, it may work with GPC without any change. Programs which rely on byte order, on the internals or sizes of data types or which use unportable things like interrupts and assembler code, will need to be changed. The following section lists the possible problems with solutions.

The GPC Run Time System (RTS) is fairly complete, and you can use all libraries written for GNU C from GNU Pascal, so there is much less need to use unportable constructs than there was in BP. (For example, BP's Turbo Vision library uses assembler to call a local procedure through a pointer. With GPC you can do this in Pascal just as with global procedures.) Please do not throw away the advantage of full portability by sticking to those workarounds.

We have successfully ported real-world projects (with several 10000s of lines) from BP to GPC, so this is possible for you, too.

7.2 BP Incompatibilities

This sections lists the remaining incompatibilities of GPC to BP, and the problems you might encounter when porting BP programs from 16-bit Dos to other platforms, and gives solutions for them.

By *incompatibilites* we mean problems that can arise when trying to compile a valid BP program with GPC. Of course, there are many features in GPC that BP doesn't know, but we call them extensions unless they can break valid BP programs, so they are not mentioned here. The subsequent sections of the '`Borland Pascal`' chapter mention a number of useful extensions that you might want to know about but which will not break your BP code.

Some of the differences can be "overcome" by command-line switches. As a summary:

```
--borland-pascal -w --uses=System -D__BP_TYPE_SIZES__ --pack-struct
-D__BP_RANDOM__ -D__BP_UNPORTABLE_ROUTINES__ -D__BP_PARAMSTR_0__
```

But please read the following notes, and don't use these switches indiscriminately when not necessary. There are reasons why they are not GPC's defaults.

7.2.1 String type

GPC's internal string format (Extended Pascal string schema) is different from BP's. BP compatible *short strings* will be implemented in GPC soon, but in the meantime, you'll have to live with the difference. In general, GPC's format has many advantages (no length limit of 255 characters, constant and reference parameters always know about their capacity, etc.), but you will see differences if you:

- declare a variable as '`String`' without a capacity. However, GPC will assume 255 then (like BP) and only warn about it (and not even this when using '`--borland-pascal`', see below), so that's not a real problem. The "clean" way, however, is to declare '`String [255]`' when you mean so (but perhaps you'll prefer '`String (2000)`'? :–).
- access "character 0" which happens to hold the length in BP. This does not work with string schemata. Use '`Length`' to get the length, and '`SetLength`' to modify it.
- try to '`FillChar`' a string, e.g. '`FillChar (StringVar, 256, 0);`', which would overwrite the '`Capacity`' field. Using '`FillChar (StringVar[1], ...);`' is alright since it accesses the characters of the string, not the '`Capacity`' and '`Length`' fields. If you want to set the length to zero, use '`SetLength`' (see above) or simply assign an empty string ('`StringVar := ''`'). This is more efficient than clearing all the characters, anyway, and has the same effect for all normal purposes.
- try to read or write strings from/to *binary* files ('`Text`' files are no problem). You will have to rewrite the code. If you also want to get rid of the 255 character limit and handle endianness issues (see below) in one go, you can use the '`ReadStringLittleEndian`' etc. routines (see Section 6.14 [Run Time System], page 105), but if you need BP compatible strings (i.e., with a one-byte length field) in data files, you cannot use them (but you can easily modify them for this purpose).

7.2.2 Qualified identifiers

GPC supports *qualified identifiers*, liek BP does. (In addition, it also supports Extended Pascal style qualified identifiers which behave a little differently for units and modules imported with '`import`' rather than '`uses`'.)

7.2.3 Assembler

GPC's inline assembler is not compatible to BP's. It uses *AT&T syntax*, supports a large variety of processors and works together with GPC's optimizer. So, either convert your inline assembly to AT&T syntax, or (usually better) to Pascal, or put it into an external file which you can assemble with your favourite (32 bit) assembler. A tutorial for using the GPC inline assembler is available at
`http://www.gnu-pascal.de/contrib/misc/gpcasm.zip`

Since many things you usually do with assembler in BP are provided by GPC's Run Time System (RTS), you will not need the inline assembler as often as in BP. (See Section 7.23 [Portability hints], page 255.)

The same applies to BP's '`inline`' directive for hand-written machine code. GPC's '`inline`' directive works for Pascal routines (see Section 7.21 [Miscellaneous], page 253), so you'll have to convert any hand-written machine code to Pascal (and thereby make it more readable, portable and easier to maintain while still getting the performance of inline code).

7.2.4 Move; FillChar

GPC supports '`Move`' and '`FillChar`', and they're fully BP compatible. However, some data structures have different internal formats which may become relevant when using these procedures. E.g., using '`Move`' on file variables does not work in GPC (there are reasons why assigning file variables with '`:=`' is not allowed in Pascal, and circumventing this restriction with '`Move`' is not a good idea). For other examples, see Section 7.2.1 [String type], page 237, Section 7.2.5 [Real type], page 239, and Section 7.2.11 [Endianness assumptions], page 240.

7.2.5 Real type

GPC does not support BP's 6-byte 'Real' type. It supports 'Single', 'Double' and 'Extended' which, at least on the IA32 and some other processors, are compatible to BP.

For BP's 6-byte 'Real' type, GPC's 'System' unit provides an emulation, called 'BPReal', as well as conversion routines to GPC's 'Real' type (which is the same as 'Double'), called 'RealToBPReal' and 'BPRealToReal'. You'll probably only need them when reading or writing binary files containing values of the BP 6-byte real type. There are no operators (e.g., '+') available for 'BPReal', but since GPC supports operator overloading, you could define them yourself (e.g., convert to 'Real', do the operation, and convert back). Needless to say that this is very inefficient and should not be done for any serious computations. Better convert your data after reading them from the file and before writing them back, or simply convert your data files once (the other types are more efficient even with BP on any non-prehistoric processor, anyway).

7.2.6 Graph unit

A mostly BP compatible 'Graph' unit exists as part of the 'GRX' package. It is known to work under DJGPP, Cygwin, mingw, Linux/IA32 with svgalib, and should work under any Unix system with X11 (tested under Linux, Solaris, AIX, etc.).

There is a small difference in the color numbering, but it should be easy to work-around: You can't assume, e.g., that color 1 is always blue, and 2 is green, etc. On a system with 15 or more bits of color depth (i.e., 32768 or more colors, which most PCs today have), they will all be very dark shades of blue. This is not really a bug, but simply a property of modern high-color modes (whereas BP's 'Graph' unit was only designed for 16 and 256 color modes).

However, the names 'Blue', 'Green' etc. stand for the correct colors in the 'Graph' unit of GRX. They are no constants, but functions (because the color depth is in general not known until runtime), so you can't use them in contexts where constants are expected. Also, they might conflict with the identifiers of the 'CRT' unit if you use both units at the same time. If you want to use computed color values in the range 0 . . . 15, you can translate them to the correct colors using the 'EGAColor' function.

7.2.7 OOP units

The OOP stuff (Turbo Vision etc.) is not yet completed, but work on several projects is underway. If you want information about the current status or access to development source, please contact the GPC mailing list.

7.2.8 Keep; GetIntVec; SetIntVec

The routines 'Keep', 'GetIntVec' and 'SetIntVec' in the 'Dos' unit do not even make sense on DJGPP (32 bit Dos extender). If your program uses these, it is either a low-level Dos utility for which porting to a 32 bit environment might cause bigger problems (because the internal issues of DPMI become relevant which are usually hidden by DJGPP), or it installs interrupt handlers which will have to be thought about more carefully because of things like locking memory, knowing about and handling the differences between real and protected mode interrupts, etc. For these kinds of things, we refer you to the DJGPP FAQ (see section "DJGPP FAQ" in *the DJGPP FAQ*).

7.2.9 TFDDs

The internal structure of file variables ('FileRec' and 'TextRec' in BP's 'Dos' unit and 'TFileRec' and 'TTextRec' in BP's 'WinDos' unit) is different in GPC. However, as far as *Text*

File Device Drivers (TFDDs) are concerned, GPC offers a more powerful mechanism. Please see the RTS reference (see Section 6.14 [Run Time System], page 105), under '`AssignTFDD`'.

7.2.10 Mem; Port; Ptr; Seg; Ofs; PrefixSeg; etc.

Those few routines in the '`System`' unit that deal with segmented pointers (e.g., '`Ptr`') are emulated in such a way that such ugly BP constructs like

```
PInteger (Ptr (Seg (a), Ofs (a) + 6 * SizeOf (Integer)))^ = 42
```

work in GPC, but they do not provide access to absolute memory addresses. Neither do '`absolute`' variables (which take a simple address in the program's address space in GPC, rather than a segmented address), and the '`Mem`' and '`Port`' arrays don't exist in GPC.

As a replacement for '`Port`' on IA32 processors, you can use the routines provided in the '`Ports`' unit, Section 6.15.12 [Ports], page 207. If you want to access absolute memory addresses in the first megabyte under DJGPP, you can't do this with normal pointers because DJGPP programs run in a protected memory environment, unless you use a dirty trick called *near pointer hack*. Please see the DJGPP FAQ (see section "DJGPP FAQ" in *the DJGPP FAQ*) for this and for other ways.

For similar reasons, the variable '`PrefixSeg`' in the '`System`' unit is not supported. Apart from *TSR*s, its only meaningful use in BP might be the setting of environment variables. GPC provides the '`SetEnv`' and '`UnSetEnv`' procedures for this purpose which you can use instead of any BP equivalents based on '`PrefixSeg`'. (However note that they will modify the program's own and its childs' environment, not its parent's environment. This is a property – most people call it a feature – of the environments, including DJGPP, that GPC compiled programs run in.)

7.2.11 Endianness assumptions

GPC also runs on big-endian systems (see Section 6.2.12.1 [Endianness], page 80). This is, of course, a feature of GPC, but might affect your programs when running on a big-endian system if they make assumptions about endianness, e.g., by using type casts (or '`absolute`' declarations or variant records misused as type casts) in certain ways. Please see the demo program '`absdemo.pas`' for an example and how to solve it.

Endianness is also relevant (the more common case) when exchanging data between different machines, e.g. via binary files or over a network. Since the latter is not easily possible in BP, and the techniques to solve the problems are mostly the same as for files, we concentrate on files here.

First, you have to choose the endianness to use for the file. Most known data formats have a specified endianness (usually that of the processor on which the format was originally created). If you define your own binary data format, you're free to choose the endianness to use.

Then, when reading or writing values larger than one byte from/to the file, you have to convert them. GPC's Run Time System supports this by some routines. E.g., you can read an array from a little-endian file with the procedure '`BlockReadLittleEndian`', or write one to a big-endian file with '`BlockWriteBigEndian`'. *Note:* The endianness in the procedure names refers to the file, not the system – the routines know about the endianness of the system they run on, but you have to tell them the endianness of the file to use. This means you do not have to (and must not) use an '`ifdef`' to use the version matching the system's endianness.

When reading or writing records or other more complicated structures, either read/write them field by field using '`BlockReadBigEndian`' etc., or read/write them with the regular '`BlockRead`' and '`BlockWrite`' procedures and convert each field after reading or before writing using procedures like '`ConvertFromBigEndian`' or '`ConvertToLittleEndian`' (but remember, when writing, to undo the conversion afterwards, if you want to keep using the data – this is not necessary with '`BlockWriteLittleEndian`' etc.).

Especially for strings, there are ready-made procedures like 'ReadStringBigEndian' or 'WriteStringLittleEndian' which will read/write the length as a 64 bit value (much space for really long strings :−) in the given endianness, followed by the characters (which have no endianness problem).

All these routines are described in detail in the RTS reference (see Section 6.14 [Run Time System], page 105), under 'endianness'. The demo program 'endiandemo.pas' contains an example on how to use these routines.

7.2.12 - -borland-pascal - disable GPC extensions

GPC warns about some BP constructs which are especially "dirty", like misusing typed constants as initialized variables. GPC also supports some features that may conflict with BP code, like macros. The command line option '--borland-pascal' disables both, so you might want to use it for a first attempt to compile your BP code under GPC. However, we suggest you try compiling without this switch and fixing any resulting problems as soon as you've become acquainted with GPC.

7.2.13 -w - disable all warnings

Even in '--borland-pascal' mode, GPC may warn about some dangerous things. To disable **all** warnings, you can use the '-w' option (note: lower-case 'w'!). This is not recommended at all, but you may consider it more BP compatible . . .

7.2.14 - -uses=System - Swap; HeapError; etc.

A few exotic BP routines and declarations (e.g., 'Swap' and 'HeapError') are contained in a 'System' unit, Section 6.15.17 [System], page 222, which GPC (unlike BP) does not automatically use in each program. To use it, you can add a 'uses System;' clause to your program. If you don't want to change your code, the command line option '--uses=System' will do the same.

7.2.15 -D__BP_TYPE_SIZES__ - small integer types etc.

Since GPC runs on 32 and 64 bit platforms, integer types have larger sizes than in BP. However, if you use the 'System' unit (see Section 7.2.14 [- -uses=System - Swap; HeapError; etc.], page 241) and define the symbol '__BP_TYPE_SIZES__' (by giving '-D__BP_TYPE_SIZES__' on the command line), it will redeclare the types to the sizes used by BP. This is less efficient and more limiting, but might be necessary if your program relies on the exact type sizes.

7.2.16 - -pack-struct - disable structure alignment

GPC by default aligns fields of records and arrays suitably for higher performance, while BP doesn't. If you don't want the alignment (e.g., because the program relies on the internal format of your structures), give the '--pack-struct' option.

7.2.17 -D__BP_RANDOM__ - BP compatible pseudo random number generator

GPC uses a more elaborate pseudo random number generator than BP does. Using the 'Random' and 'Randomize' functions works the same way, but there is no 'RandSeed' variable (but a 'SeedRandom' procedure). However, if you use the 'System' unit (see Section 7.2.14 [- -uses=System - Swap; HeapError; etc.], page 241) and define the symbol '__BP_RANDOM__' (by giving '-D__BP_RANDOM__' on the command line), it will provide a 100% BP compatible pseudo random number generator, including the 'RandSeed' variable, which will produce exactly the same sequence of pseudo random numbers that BP's pseudo random number generator does. Even the 'Randomize' function will then behave exactly like in BP.

7.2.18 -D__BP_UNPORTABLE_ROUTINES__ - Intr; DosVersion; etc.

A few more routines in the 'Dos' and 'WinDos' units besides the ones mentioned under Section 7.2.8 [Keep; GetIntVec; SetIntVec], page 239, like 'Intr' or 'DosVersion', are meaningless on non-Dos systems. By default, the 'Dos' unit does not provide these routines (it only provides those that are meaningful on all systems, which are most of its routines, including the most commonly used ones). If you need the unportable ones, you get them by using the 'System' unit (see Section 7.2.14 [- -uses=System - Swap; HeapError; etc.], page 241) and defining the symbol '__BP_UNPORTABLE_ROUTINES__' (by giving '-D__BP_UNPORTABLE_ROUTINES__' on the command line). If you use 'Intr' or 'MsDos', your program will only compile under DJGPP then. Other routines, e.g. 'DosVersion' are emulated quite roughly on other systems. Please see the notes in the 'Dos' unit (see Section 6.15.2 [Dos], page 168) for details.

7.2.19 -D__BP_PARAMSTR_0__ - BP compatible ParamStr (0) behaviour

In BP (or under Dos), 'ParamStr (0)' always contains the full path of the current executable. Under GPC, by default it contains what was passed by the caller as the 0th argument – which is often the name of the executable, but that's merely a convention, and it usually does not include the path.

If you use the 'System' unit (see Section 7.2.14 [- -uses=System - Swap; HeapError; etc.], page 241) and define the symbol '__BP_PARAMSTR_0__' (by giving '-D__BP_PARAMSTR_0__' on the command line), it will change the value of 'ParamStr (0)' to that of 'ExecutablePath', overwriting the value actually passed by the caller, to imitate BP's/Dos's behaviour. **However note:** On most systems, 'ExecutablePath' is **not** guaranteed to return the full path, so defining this symbol doesn't change anything. In general, you **cannot** expect to find the full executable path, so better don't even try it, or your program will (at best) run on some systems. For most cases where BP programs access their own executable, there are cleaner alternatives available.

7.3 IDE versus command line

On the Dos (DJGPP) and Linux platforms, you can use RHIDE for GNU Pascal; check the subdirectories of your DJGPP distribution.

Unfortunately, there is no IDE which would run on all platforms. We are working on it, but this will take some time. Please be patient – or offer your help!

Without an IDE, the GNU Pascal Compiler, GPC, is called about like the command-line version of the Borland Pascal Compiler, BPC. Edit your source file(s) with your favorite ASCII editor, then call GNU Pascal with a command line like

```
C:\GNU-PAS> gpc hello.pas -o hello.exe
```

on your Dos or OS/2 box or

```
myhost:/home/joe/gnu-pascal> gpc hello.pas -o hello
```

on your Unix (or Unix-compatible) system.

Don't omit the '.pas' suffix: GPC is a common interface for a Pascal compiler, a C, ObjC and C++ compiler, an assembler, a linker, and perhaps an Ada and a FORTRAN compiler. From the extension of your source file GPC figures out which compiler to run. GPC recognizes Pascal sources by the extension '.pas', '.p', '.pp' or '.dpr'.

The -o is a command line option which tells GPC how the executable has to be named. If not given, the executable will be called 'a.out' (Unix) or 'a.exe' (Dos). However, you can use the '--executable-file-name' to tell GPC to always call the executable like the source (with the extension removed under Unix and changed to '.exe' under Dos).

Note that GPC is case-sensitive concerning file names and options, so it will *not* work if you type

```
C:\GNU-PAS> GPC HELLO.PAS -O HELLO.EXE
```

GPC is a very quiet compiler and doesn't print anything on the screen unless you request it or there is an error. If you want to see what is going on, invoke GPC with additional options:

```
-Q           "don't be quiet"  (or: Quassel-Modus in German)
```

(with *capital* '`Q`'!) means that GPC prints out the names of procedures and functions it processes, and

```
--verbose
```

or abbreviated

```
-v
```

means that GPC informs you about the stages of compilation, i.e. preprocessing, compiling, assembling, and linking.

One example (this time for OS/2):

```
[C:\GNU-Pascal] gpc --verbose -Q hello.pas
```

Throughout this chapter, we will tell you about a lot of command-line switches. They are all invoked this way.

After compilation, there will be an executable `hello` file in the current directory. (`hello.exe` on Dos or OS/2.) Just run it and enjoy. If you're new to Unix, please note that the current directory is not on the PATH in most installations, so you might have to run your program as '`./hello`'. This also helps to avoid name conflicts with other programs. Such conflicts are especially common with the program name '`test`' which happens to be a standard utility under Unix that does not print any output. If you call your program '`test.pas`', compile it, and then invoke '`test`', you will usually not run your program, but the utility which leads to mysterious problems. So, invoke your program as '`./test`' or, better yet, avoid the name '`test`' for your programs.

If there are compilation errors, GNU Pascal will not stop compilation after the first one – as Borland Pascal does – but try to catch all errors in one compilation. If you get more error messages than your screen can hold, you can catch them in a file (e.g. `gpc.out`) or pipe them to a program like '`more`' in the following way:

```
gpc hello.pas 2> gpc.out
```

This works with OS/2 and any bash-like shell under Unix; for Dos you must get a replacement for `command.com` which supports this kind of redirection, or use the '`redir`' utility (see also the DJGPP FAQ, section "DJGPP FAQ" in *the DJGPP FAQ*.):

```
C:\GNU-PAS> redir -eo gpc hello.pas -o hello.exe | more
```

You can also use Borland's IDE for GNU Pascal on the Dos platform: Install the GNU Pascal Compiler in the Tools menu (via Options/Tools).

```
Name:       GNU Pascal
Path:       gpc
Arguments:  $SAVE ALL --executable-file-name $NAME($EDNAME).pas
HotKey:     Shift+F9
```

Note once more that GPC is case-sensitive, so it is important to specify `.pas` instead of the `.PAS` Borland Pascal would append otherwise!

You can include more command-line arguments to GNU Pascal (e.g. '`--automake`'; see below) as you will learn more about them.

Since Borland Pascal will try to recompile your program if you use its `Run` menu function, you will need another tool to run your program:

```
Name:       Run Program
Path:       command.com
Arguments:  /c $NAME($EDNAME)
HotKey:     Shift+F10
```

7.4 Comments

GPC supports comments surrounded by '`{ }`' and '`(* *)`', just like BP does. According to the ISO 7185 and ISO 10206 standards, Pascal allows comments opened with `(*` and closed with `}`. Borland Pascal does not support such *mixed* comments, so you might have sources where passages containing comments are "commented out" using the other kind of comment delimiters. GPC's default behaviour is (like BP) not to allow mixed comments, so you don't need to worry about this. However, if you happen to like mixed comments, you can turn them on either by a command-line option, or by a compiler directive:

```
--mixed-comments     {$mixed-comments}    (*$mixed-comments*)
```

GPC supports nested comments (e.g., '`{ foo { bar } baz }`'), but they are disabled by default (compatible to BP which doesn't know nested comments at all). You can enable them with the option '`--nested-comments`' (or the equivalent compiler directive)

GPC also supports Delphi style comments starting with '`//`' and extending until the end of the line. This comment style is activated by default unless one of the '`--classic-pascal`', '`--extended-pascal`', '`--object-pascal`' or '`--borland-pascal`' dialect options is given. You can turn them on or off with the '`--[no-]delphi-comments`' option.

7.5 BP Compatible Compiler Directives

All of BP's one-letter compiler directives (except H, P, Q, R, V) are supported by GPC, though some of them are ignored because they are not necessary under GPC. Besides, GPC supports a lot more directives. For an overview, see Section 6.9 [Compiler Directives], page 89.

7.6 Units, GPI files and Automake

You can use units in the same way as in Borland Pascal. However, there are some additional features.

Concerning the syntax of a unit, you can, if you want, use Extended Pascal syntax to specify a unit initializer, i.e., instead of writing

```
begin
  ...
end.
```

at the end of the unit, you can get the same result with

```
to begin do
  begin
    ...
  end;
```

and there also exists

```
to end do
  begin
    ...
  end;
```

which specifies a finalization routine. You can use this instead of Borland Pascal's exit procedures, but for compatibility, the included '`System`' unit also provides the '`ExitProc`' variable. The '`to begin do`' and/or '`to end do`' parts must be followed by the final '`end.`'. See Section 6.1.8.1 [Modules], page 60, for information about Extended Pascal modules, an alternative to units.

When GPC compiles a unit, it produces two files: an `.o` object file (compatible with other GNU compilers such as GNU C) plus a `.gpi` file which describes the interface.

If you are interested in the internal format of GPI file, see Section 12.6 [GPI files], page 485.

If you want to compile a program that uses units, you must "make" the project. (This is the command-line switch '-M' or the IDE keystroke 'F9' in BP.) For this purpose, GPC provides the command-line switch '--automake':

```
gpc --automake hello.pas
```

If you want to force everything to be rebuilt rather than only recompile changed files ('-B' or "build" in BP), use '--autobuild' instead of '--automake':

```
gpc --autobuild hello.pas
```

For more information about the automake mechanism, see Section 12.7 [Automake], page 488.

If you do not want to use the automake mechanism for whatever reason, you can also compile every unit manually and then link everything together.

GPC does not automatically recognize that something is a unit and cannot be linked; you have to tell this by a command line switch:

```
-c            only compile, don't link.
```

(If you omit this switch when compiling a unit, you only get a linker error message 'undefined reference to 'main''. Nothing serious.)

For example, to compile two units, use:

```
gpc -c myunit1.pas myunit2.pas
```

When you have compiled all units, you can compile a program that uses them without using '--automake':

```
gpc hello.pas
```

However, using '--automake' is recommended, since it will recompile units that were modified.

You could also specify the program and the units in one command line:

```
gpc hello.pas myunit1.pas myunit2.pas
```

One of the purposes of writing units is to compile them separately. However, GNU Pascal allows you to have one or more units in the same source file (producing only one .o file but separate .gpi files). You even can have a program and one or more units in one source file; in this case, no .o file is produced at all.

7.7 Optimization

GNU Pascal is a 32/64 bit compiler with excellent optimization algorithms (which are identically the same as those of GNU C). There are six optimization levels, specified by the command line options '-O', '-O2', ..., '-O6'.

One example:

```
program OptimizationDemo;

procedure Foo;
var
  A, B: Integer;
begin
  A := 3;
  B := 4;
  WriteLn (A + B)
end;

begin
```

```
    Foo
  end.
```

When GNU Pascal compiles this program with optimization (`-O3`), it recognizes that the argument to 'WriteLn' is the constant 7 – and optimizes away the variables `A` and `B`. If the variables were global, they would not be optimized away because they might be accessed from other places, but the constant 7 would still be optimized.

For more about optimization, see the GNU C documentation.

7.8 Debugging

The command line option `-g` specifies generation of debugging information for GDB, the GNU debugger. GDB comes with its own documentation. Currently, GDB does not understand Pascal syntax, so you should be familiar with C expressions if you want to use it.

See also "Notes for debugging" in the "Programming" chapter; see Section 6.12 [Notes for Debugging], page 102.

Sometimes it is nice to have a look at the assembler output of the compiler. You can do this in a debugger or disassembler (which is the only way to do it in BP), but you can also tell GPC to produce assembler code directly: When you specify the `-S` command line option, GPC produces an `.s` file instead of an `.o` file. The `.s` file contains assembler source for your program. More about this in the next section.

7.9 Objects

Objects in the Borland Pascal 7.0 notation are implemented into GNU Pascal with the following differences:

- the '`private`', '`protected`', '`public`' and '`published`' directives are recognized but ignored,
- data fields and methods may be mixed:

```
type
  MyObj = object
    x: Integer;
    procedure Foo; virtual;
    y: Real;
    function Bar: Char;
  end;
```

7.10 Strings in BP and GPC

Strings are "Schema types" in GNU Pascal which is something more advanced than Borland-style strings. For variables, you cannot specify just `String` as a type like in Borland Pascal; for parameters and pointer types you can. There is no 255 characters length limit. According to Extended Pascal, the maximum string length must be in (parentheses); GNU Pascal accepts [brackets], too, however, like BP.

For more about strings and schema types see Section 6.2.11.5 [Schema Types], page 72.

GPC supports Borland Pascal's string handling functions and some more (see Section 6.10.2 [String Operations], page 94):

Borland Pascal	**GNU Pascal**
`Length`	`Length`
`Pos`	`Pos, Index` (1)
`Str`	`Str, WriteStr` (1) (2)
`Val`	`Val, ReadStr` (2)

`Copy`	`Copy, SubStr, MyStr[2 .. 7]` (3)
`Insert`	`Insert`
`Delete`	`Delete`
`MyStr[0] := #7`	`SetLength (MyStr, 7)`
`=, <>, <, <=, >, >=`	`=, <>, <, <=, >, >=` (4)
	`EQ, NE, LT, LE, GT, GE`
n/a	`Trim`

Notes:

(1) The order of parameters of the Extended Pascal routines (‘`Index`’, ‘`WriteStr`’) is different from the Borland Pascal routines.

(2) ‘`ReadStr`’ and ‘`WriteStr`’ allow an arbitrary number of arguments, and the arguments are not limited to numbers. ‘`WriteStr`’ also allows comfortable formatting like ‘`WriteLn`’ does, e.g. ‘`WriteStr (Dest, Foo : 20, Bar, 1/3 : 10 : 2)`’.

(3) ‘`SubStr`’ reports a runtime error if the requested substring does not fit in the given string, ‘`Copy`’ does not (like in BP).

(4) By default, the string operators behave like in BP. However, if you use the option ‘`--no-exact-compare-strings`’ or ‘`--extended-pascal`’, they ignore differences of trailing blanks, so, e.g., ‘`'foo'`’ and ‘`'foo '`’ are considered equal. The corresponding functions (‘`EQ`’, . . .) always do exact comparisons.

7.11 Typed Constants

GNU Pascal supports Borland Pascal’s “typed constants” but also Extended Pascal’s initialized variables:

```
var
  x: Integer value 7;
```

or

```
var
  x: Integer = 7;
```

When a typed constant is misused as an initialized variable, a warning is given unless you specify ‘`--borland-pascal`’.

When you want a local variable to preserve its value, define it as ‘`static`’ instead of using a typed constant. Typed constants also become static automatically for Borland Pascal compatibility, but it’s better not to rely on this “feature” in new programs. Initialized variables do not become static automatically.

```
program StaticDemo;

procedure Foo;
{ x keeps its value between two calls to this procedure }
var
  x: Integer = 0; attribute (static);
begin
  WriteLn (x);
  Inc (x)
end;

begin
  Foo;
  Foo;
  Foo;
```

```
end.
```

For records and arrays, GPC supports both BP style and Extended Pascal style initializers. When you initialize a record, you may omit the field names. When you initialize an array, you may provide indices with a :. However, this additional information is ignored completely, so perhaps it's best for the moment to only provide the values . . .

```
program BPInitVarDemo;
const
  A: Integer = 7;
  B: array [1 .. 3] of Char = ('F', 'o', 'o');
  C: array [1 .. 3] of Char = 'Bar';
  Foo: record
    x, y: Integer;
  end = (x: 3; y: 4);
begin
end.
```

7.12 Bit, Byte and Memory Manipulation

The bitwise operators '`shl`', '`shr`', '`and`', '`or`', '`xor`' and '`not`' work in GNU Pascal like in Borland Pascal. As an extension, you can use them as procedures, for example

```
program AndProcedureDemo;
var x: Integer;
begin
  and (x, $0000ffff);
end.
```

as an alternative to

```
program AndOperatorDemo;
var x: Integer;
begin
  x := x and $0000ffff;
end.
```

GPC accepts the BP style notation '`$abcd`' for hexadecimal numbers, but you also can use Extended Pascal notation:

```
program EPBaseDemo;
const
  Binary =  2#11111111;
  Octal  =  8#177;
  Hex    = 16#ff;
begin
end.
```

and so on up to a basis of 36. Of course, you can mix the notations as you like, e.g.:

```
program BPEPBaseDemo;
begin
  WriteLn ($cafe = 2#1100101011111110)
end.
```

'`Inc`' and '`Dec`' are implemented like in Borland Pascal. '`Pred`' and '`Succ`' are generalized according to Extended Pascal and can have a second (optional) parameter:

```
procedure SuccDemo;
var a: Integer = 42;
```

```
begin
  a := Succ (a, 5);
  WriteLn (a) { 47 }
end.
```

BP style '`absolute`' variables work in the context of overloading other variables as well as in the context of specifying an absolute address, but the latter is highly unportable and not very useful even in Dos protected mode.

```
program BPAbsoluteDemo;

type
  TString = String (80);
  TTypeChoice = (t_Integer, t_Char, t_String);

{ @@ WARNING: BAD STYLE! }
procedure ReadVar (var x: Void; TypeChoice: TTypeChoice);
var
  xInt: Integer absolute x;
  xChar: Char absolute x;
  xStr: TString absolute x;
begin
  case TypeChoice of
    t_Integer: ReadLn (xInt);
    t_Char   : ReadLn (xChar);
    t_String : ReadLn (xStr);
  end
end;

var
  i: Integer;
  c: Char;
  s: TString;

begin
  ReadVar (i, t_Integer);
  ReadVar (c, t_Char);
  ReadVar (s, t_String);
  WriteLn (i, ' ', c, ' ', s)
end.
```

GNU Pascal knows Borland Pascal's procedures `FillChar` and `Move`. However, their use can be dangerous because it often makes implicit unportable assumptions about type sizes, endianness, internal structures or similar things. Therefore, avoid them whenever possible. E.g., if you want to clear an array of strings, don't '`FillChar`' the whole array with zeros (this would overwrite the Schema discriminants, see Section 6.15.15 [Strings], page 215), but rather use a '`for`' loop to assign the empty string to each string. In fact, this is also more efficient than '`FillChar`', since it only has to set the length field of each string to zero.

7.13 User-defined Operators in GPC

GNU Pascal allows the user to define operators according to the Pascal-SC syntax:

```
program PXSCOperatorDemo;
```

```
type
  Point = record
    x, y: Real;
  end;

operator + (a, b: Point) c: Point;
begin
  c.x := a.x + b.x;
  c.y := a.y + b.y;
end;

var
  a, b, c: Point = (42, 0.5);

begin
  c := a + b
end.
```

The Pascal-SC operators '+>', '+<', etc. for exact numerical calculations are not implemented, but you can define them.

7.14 Data Types in BP and GPC

- Integer types have different sizes in Borland and GNU Pascal:

Borland Pascal	GNU Pascal	Bits (1)	Signed
ShortInt	ByteInt	8	yes
Integer	ShortInt	16	yes
LongInt	Integer	32	yes
Comp	LongInt, Comp	64	yes
Byte	Byte	8	no
Word	ShortWord	16	no
n/a	Word	32	no
n/a	LongWord	64	no

  (1) The size of the GNU Pascal types may depend on the platform. The sizes above apply to 32 bit platforms, including the IA32.

  If you care for types with exactly the same size as in Borland Pascal, take a look at the 'System' unit and read its comments.

  You can get the size of a type with 'SizeOf' in bytes (like in Borland Pascal) and with 'BitSizeOf' in bits, and you can declare types with a specific size (given in bits), e.g.:

```
program IntegerSizeDemo;
type
  MyInt  = Integer attribute (Size = 42);  { 42 bits, signed }
  MyWord = Word attribute (Size = 2);      { 2 bits, unsigned,
                                              i.e., 0 .. 3 }
  MyCard = Cardinal attribute (Size = 2);  { the same }

  HalfInt = Integer attribute (Size = BitSizeOf (Integer) div 2);
    { A signed integer type which is half as big as the normal
      'Integer' type, regardless of how big 'Integer' is
      on any platform the program is compiled on. }

begin
```

```
end.
```

- Borland's real (floating point) types are supported except for the 6-byte software Real type (but the 'System' unit provides conversion routines for it). GNU Pascals's 'Real' type has 8 bytes on the IA32 and is the same as 'Double'. In addition there are alternative names for real types:

Borland Pascal	**GNU Pascal**
`Single`	`Single, ShortReal`
`Real`	n/a (1)
`Double`	`Double, Real`
`Extended`	`Extended, LongReal`
`Comp`	`LongInt, Comp` (see above)

  (1) But see 'BPReal', 'RealToBPReal' and 'BPRealToReal' in GPC's `System` unit.
- Complex numbers: According to Extended Pascal, GNU Pascal has built-in complex numbers and supports a number of mathematical functions on them, e.g. 'Abs', 'Sqr', 'SqRt', 'Exp', 'Ln', 'Sin', 'Cos', 'ArcTan'.
- Record types: GNU Pascal by default aligns 32-bit fields on 4-byte addresses because this improves performance. So, e.g., the record

```
type
  MyRec = record
    f, o, oo: Boolean;
    Bar: Integer
  end;
```

  has 8 bytes, not 7. Use the `--pack-struct` option or declare the record as 'packed' to force GPC to pack it to 7 bytes. However, note that this produces somewhat less efficient code on the IA32 and far less efficient code on certain other processors. Packing records and arrays is mostly useful only when using large structures where memory usage is a real concern, or when reading or writing them from/to binary files where the exact layout matters.

7.15 BP Procedural Types

In addition to BP's procedural types, GNU Pascal has pointers to procedures:

```
type
  FuncPtr = ^function (Real): Real;
```

The differences between procedure pointers and procedural types are only syntactical:

- In the first case, one can pass/assign a procedure/function with '@myproc', in the latter case just with 'myproc' (which can lead to confusion in the case of functions – though GPC should always recognize the situation and not try to call the function).
- In the first case, one can call the routine via 'myprocptr^', in the latter case just with 'myprocvar'.
- To retrieve the address of a procedure stored in a variable, one can use 'myprocptr' in the first case and '@myprocvar' in the latter.
- If, for some reason, one needs the address of the variable itself, in the first case, that's obtained with '@myprocptr', in the second case with '@@myprocvar'!
- Bottom line: BP style procedural types are simpler to use in normal cases, but somewhat strange in the last example.

One can use both kinds in the same program, of course, though it is recommended to stick to one kind throughout to avoid maximum confusion.

GNU Pascal also supports Standard Pascal's procedural parameters (see Section 7.20 [Special Parameters], page 253).

Furthermore, GNU Pascal allows you to call even local procedures through procedural pointers, variables or parameters without reverting to any dirty tricks such as assembler (which is necessary in BP).

The differences between the various kinds of procedural types, pointers and parameters are demonstrated in the demo program '`procvardemo.pas`'. An example for calling local routines through procedural parameters can be found in the demo program '`iteratordemo.pas`'.

7.16 Files

- GPC supports files like in Borland Pascal, including untyped files, '`BlockRead`', '`BlockWrite`' and '`Assign`'. Instead of '`Assign`', you can also use the '`Bind`' mechanism of Extended Pascal.

  Besides the routines supproted by BP, there are many more routines available that deal with files, file names and similar things in a portable way. In contrast to Borland Pascal, you don't have to use any platform-specific units to do these kinds of things, though portable emulations of those units (e.g., of the '`Dos`' and '`WinDos`' units) are also available for compatibility.

7.17 Built-in Constants

- The '`MaxInt`', '`MaxLongInt`', '`Pi`' constants are supported like in BP.
- Other built-in constants: GNU Pascal has '`MaxChar`', '`MaxReal`', '`MinReal`', '`EpsReal`' and a number of other useful constants.

7.18 Built-in Operators in BP and GPC

Besides the operators found in Borland Pascal, GNU Pascal supports the following operators:

- Exponentiation: According to Extended Pascal, GNU Pascal supports the exponentiation operators `pow` and `**` which do not exist in Borland Pascal. You can use `x pow y` for integer and `x ** y` for real or complex exponents. The basis may be integer, real or complex in both cases.
- GNU Pascal has a symmetric set difference operator `set1 >< set2`. For more about this, see Section 6.10.7 [Set Operations], page 98.

7.19 Built-in Procedures and Functions

- '`GetMem`' and '`FreeMem`' are supported like in BP.

  The second parameter to '`FreeMem`' is ignored by GNU Pascal and may be omitted. Memory blocks are always freed with the same size they were allocated with.

  Remark: Extended Pascal Schema types provide a cleaner approach to most of the applications of '`GetMem`' and '`FreeMem`'.
- '`Min`' and '`Max`': GNU Pascal has built-in '`Min`' and '`Max`' functions (two arguments) which work for all ordinal types ('`Integer`', '`Char`', ...) plus '`Real`'.
- '`UpCase`', '`High`', '`Low`' and similar functions are built-in. In contrast to Borland Pascal, GNU Pascal's '`UpCase`' function is aware of non-ASCII characters of certain languages (e.g., accented letters and "umlauts"), but for compatibility this feature is disables in '`--borland-pascal`' mode. There is also a '`LoCase`' function.
- '`Lo`', '`Hi`', '`Swap`' functions: not built-in, but available in the '`System`' unit.

7.20 Special Parameters

- Untyped reference parameters can be denoted by
  ```
  procedure Foo (var x);
  ```
  like in Borland Pascal. In GNU Pascal, you can also use
  ```
  procedure Foo (var x: Void);
  ```
- GNU Pascal defines *ellipsis* parameters for variable argument lists:
  ```
  procedure Foo (a: Integer; ...);
  ```
  However, GPC does not (yet) provide a portable mechanism to access the additional arguments.
- Structured function result types: According to Extended Pascal, GNU Pascal allows functions to return records and arrays.
- BP style *open array parameters*
  ```
  procedure Foo (a: array of Integer);
  ```
  are implemented. However, Standard Pascal '`conformant array parameters`' are usually a cleaner mechanism to pass arrays of variable size.
- Besides BP compatible procedural types and procedure pointers (see Section 7.15 [BP Procedural Types], page 251), GNU Pascal supports Standard Pascal's procedural parameters:
  ```
  procedure DrawGraph (function f (x: Real): Real);
  ```

7.21 Miscellaneous

- Headlines: According to Extended Pascal, a program headline must contain the program's parameters:
  ```
  program Foo (Input, Output);
  begin
  end.
  ```
  In GNU Pascal, headline parameters are optional. If the headline is omitted entirely, a warning is given unless you have specified '`--borland-pascal`' in the command line.
- '`case`' statements: In a '`case`' statement, GNU Pascal allows `otherwise` (according to Extended Pascal) as an alternative to `else`:
  ```
  program CaseOtherwiseDemo;
  var x: Integer;
  begin
    ReadLn (x);
    case x of
      1: WriteLn ('one');
      2: WriteLn ('two');
      otherwise
         WriteLn ('many')
    end
  end.
  ```
  Note: In the absence of a '`case`' or '`otherwise`' branch, missing cases labels cause an error in Extended Pascal (which goes unnoticed in Borland Pascal). GPC does not give this error, but a warning if the '`-Wswitch`' option is given, however only for enumeration types.
- Character constants: BP compatible character constants like '`^M`' as well as '`#13`' are implemented into GNU Pascal.
- Sets: GNU Pascal has a `Card` function for sets which counts their elements. Unlike Borland Pascal, GNU Pascal does not limit sets to the range 0 .. 255.

- Inline: GNU Pascal allows "inline" Pascal procedures and functions, while Borland Pascal only allows machine code to be inlined:

  Borland Pascal:

  ```
  function Max (x, y: Integer): Integer;
    inline ($58 / $59 / $3b / $c1 / $7f / $01 / $91);
  ```

  GNU Pascal:

  ```
  program InlineDemo;

  function Max (x, y: Integer): Integer; attribute (inline);
  begin
    if x > y then
      Max := x
    else
      Max := y
  end;

  begin
    WriteLn (Max (42, 17), ' ', Max (-4, -2))
  end.
  ```

  (Actually, a more general 'Max' function is already built in.)

  This feature is not so important as it might seem because in optimization level 3 or higher (see Section 5.2 [GPC Options], page 44), GNU Pascal automatically inlines short procedures and functions.

7.22 BP and Extended Pascal

Pascal is a well-known programming language and hardly needs to be described here. Note, however, that there is a large difference between the language used by the BP compiler and the Pascal Standards.

Extended Pascal is a standardized language based on the original Standard Pascal, but with significant extensions. Unfortunately, Borland Pascal does not conform to any of the Pascal standards. Writing a program that both complies to Extended Pascal (or even Standard Pascal) and compiles with BP is almost impossible for any non-trivial task.

On the other hand, BP has some nice features that make it very powerful in the environments in which it runs. However, some of those features are of little use on non-Dos systems and would not be good candidates for standardization.

There are also several BP features which are semantically similar to features in Standard Pascal or Extended Pascal, but syntactically different.

Therefore, in order to be useful to users coming from either side, GPC supports both the standards and the BP dialect as good as possible. By default, GPC allows features from any dialect it knows. By giving a dialect option such as '--borland-pascal' or '--extended-pascal', you can tell GPC to disable the features not found in that dialect, and to adjust its warning behaviour to the dialect.

The different sets of reserved words are a little problem, but GPC solves it by making the words in question only "conditionally reserved" which works transparently without problems in most cases. Still, giving a dialect option will disable all keywords not part of this dialect.

Apart from this, there are surprisingly few real conflicts between the dialects. Therefore, you can usually compile your BP code without the '--borland-pascal' option and make use of all of GPC's features. You might be surprised, though, when GPC accepts things you didn't know were allowed. :–)

Finally, if you want to make use of some of GPC's extensions (compared to BP) and still keep the code compileable with BP without using '`ifdef`'s all over the place, we suggest you look at the unit '`gpc-bp.pas`', shipped with GPC, which contains BP versions of some of GPC's features. Please read the comments at the beginning of the unit to find out more about it.

7.23 Portability hints

GPC offers you the possibility to make your code fully portable to each of the many platforms supported by GPC. It would be a pity not to make use of this.

This section lists some known pitfalls that often hinder otherwise well-written programs to take full advantage of GPC. If you have never used any compiler but Borland Pascal and similar compilers, some of the advices might look strange to you. But this is just the same level of strangeness that your old programs will have for you once you have understood the principles of cross-platform portability. Remember that many tricks you have always been applying almost automatically in Borland Pascal were necessary to overcome certain limitations of the Dos platform and to compensate for the compiler's missing optimization. Programming with an optimizing compiler like GPC for platforms without a 64 kB limit is a completely new experience – and perhaps it is among the reasons why you are now working with GPC in the first place?

Portability – why?

Okay – but why should I bother and make my program portable? I know that all who want to use my program are running WXYZ-OS anyway.

Yes, but that's the result of a self-fulfilling prophecy. It depends on **you** whether it will always remain like this or not. Consider a program ABC written for a single platform, WXYZ-OS. Naturally, only WXYZ-OS-users get interested in ABC. The author gets feedback only from WXYZ-OS users and does not see any reason to make the program cross-platform. Then people realize that if they want to run ABC they must move to WXYZ-OS. The author concludes that people only want WXYZ-OS programs, and so on.

To break out, just create a portable version of your program **now**. Then all OSes have equal chances to show their abilities when running your program, and your customers can choose their OS. Then, maybe, they decide to use your program just for the reason that they can be sure that it will run on all present and future platforms and not only on a specific one – who knows?

My program is a tool specifically designed to make the best of the STUV feature of WXYZ-OS. There is no point in making it portable.

How much do you know about non-WXYZ-OSes? Just ask an expert how the STUV feature is named elsewhere. Be sure, if it is of value, it exists almost everywhere.

Low-level features

I am using a lot of low-level stuff in my programs, so they cannot be portable.

You do not use those low-level routines directly in your high-level routines, do you? There should always be a layer "in-between" that encapsulates the low-level routines and present an API to your program that exactly reflects the needs of your application. This "API in between" is the point where you can exchange the low-level routines by portable calls to GPC's Run Time System.

If you do not have such a layer in-between, then the API of the low-level routines you call are your first approximation for such a layer. If you have ever thought "it would be great if that API function had that additional parameter", then your own extended version of that API function that **has** that parameter can become part of your "API in between". But then don't stop here: Certainly the API of the OS is **not** ideal for your program's needs. Just create more routines that encapsulate all OS-specific stuff . . .

When the low-level stuff in question consists of interrupts, assembler and similar things, then the first thing you need is a portable replacement of the functionality. Fortunately, GPC covers many things already in Pascal that require assembler in Borland Pascal:

- GPC's units and libraries come with source. You do not need to learn assembler and to write a complete replacement for the CRT unit if you only want to adapt some tiny detail in the behavior of CRT to your personal needs.
- GPC's Run Time System is fairly complete. For example, to extract the assigned name of a '`File`' variable, you do not need to mess around with the internal representation of those variables, but you can type '`uses GPC`' and then use the '`FileName`' function. In the same unit, you will find a '`FileExists`' function and much more.
- Manually "constructing" an object is covered by the '`SetType`' procedure in GPC. This is where Turbo Vision uses assembler to load an object from a stream.
- Calling local procedures and functions via pointers simply works in GPC. This is another place where, for instance, Turbo Vision's '`ForEach`' method uses assembler, while GPC lets you do the same thing in Pascal.
- Interfacing with the OS can be done through library calls. GPC's built-in functions and the GPC unit offer a rather complete set of routines. And again: You have the source of all this.
- Using '`FillChar`' and '`Move`' does not necessarily speed up your programs. Using them to circumvent restrictions of the language (e.g. for direct assignments between variables of object or file type) is asking for trouble. '`FillChar`' was created in UCSD Pascal to set consecutive chars in a string to the same value, and '`Move`' was created to move the chars within the same string. Better do not use them for other purposes.

8 The Alphabetical GPC Language Reference

This chapter is still under development. All keywords and built-in identifiers are listed, but not all with explanations.

This chapter contains an alphabetical list of all keywords (reserved words) and built-in identifiers of the GNU Pascal compiler. For detailed and comprehensive description of syntax and reserved words, see Chapter 6 [Programming], page 47. This chapter explains only built-in procedures and functions in detail.

It does not cover extensions provided by external units and libraries which are supposed to come with their own documentation. For the interfaces of the units that come with GPC, see Section 6.15 [GPC Units], page 151.

Abs

Synopsis

```
function Abs (i: integer_type): integer_type;
```

or

```
function Abs (x: real_type): real_type;
```

or

```
function Abs (z: complex_type): real_type;
```

Description

Returns the absolute value of the argument. For integer or real values of 'x', the definition is

```
function Abs (x: integer_or_real_type): integer_or_real_type;
begin
  if x < 0 then
    Abs := -x
  else
    Abs := x
end;
```

whereas for complex values it is

```
function Abs (x: Complex): Real;
begin
  Abs := SqRt (x * Conjugate (x))
end;
```

Conforming to

The function 'Abs' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program AbsDemo;
var
  i1: Complex;
begin
```

```
  WriteLn (Abs (42));              { 42 }
  WriteLn (Abs (-42));             { 42 }
  WriteLn (Abs (-12.1) : 0 : 1);   { 12.1 }
  i1 := Cmplx (1, 1);              { 1 + i }
  WriteLn (Abs (i1) : 0 : 3)       { 1.414, i.e. SqRt (2) }
end.
```

See also

[Sqr], page 423.

absolute

Synopsis

```
var
  variable_name: data_type absolute variable_reference;
```

or

```
var
  variable_name: data_type absolute integer_expression;
```

Description

The first meaning of the '`absolute`' directive allows to put a variable to the address of another one and thus provides a type-casting mechanism.

In most cases, *variable_reference* will be just a variable name, but GPC also allows arbitrary pointer expressions here. If *variable_reference* has neither a constant address nor is a variable parameter, GPC prints a warning. This warning is suppressed in "extended syntax" mode which is switched on by the '`--extended-syntax`' option or the '`{$X+}`' compiler directive.

GPC also allows explicit type casts. Variant records (as defined in ISO 7185 Pascal), however, have no *guaranteed* overlaying and are therefore *not* suitable for type casts.

The second meaning of '`absolute`' places a variable at a specified address. This is useful on machines without virtual memory addressing for doing certain low-level operations, but should be avoided on systems with memory protection such as Unix-like systems. GPC does not check whether the specified virtual address makes any sense and does not provide a built-in mechanism to map it to a real address.

GPC warns about this second use of '`absolute`' unless "extended syntax" has been requested.

Conforming to

'`absolute`' is a Borland Pascal extension.

Borland Pascal has a slightly different syntax for the second meaning related to the addressing scheme of IA32 processors working in real mode.

Allowing arbitrary memory references instead of just variable names in the first meaning of '`absolute`' is a GNU Pascal extension.

Example

```
program AbsoluteDemo;

{$X+}

const
  IOMem = $f0000000;
  MaxVarSize = MaxInt div 8;

var
  Mem: array [0 .. MaxVarSize - 1] of Byte absolute 0;

  { This address has no actual meaning }
  MyPort: Byte absolute IOMem + $c030;

{ Beware: Using any of the variables above will crash
  your program unless you know exactly what you do!
  That's why GPC warns about it without the $X+ directive. }

var
  x: Real;
  a: array [1 .. SizeOf (Real)] of Byte absolute x;
  i: Integer;
  b: Byte absolute a[i];  { GNU Pascal extension:
                             non-constant memory reference. }

begin
  x := 3.14;

  { Look at the internal representation of a real variable. }
  for i := 1 to SizeOf (Real) do
    Write (a[i] : 4);
  WriteLn;

  { The same again, more ugly ... }
  for i := 1 to SizeOf (Real) do
    Write (b : 4);
  WriteLn;

  { And yes, there's an even more ugly way to do it ... }
  for i := 1 to SizeOf (Real) do
    Write (Mem[PtrCard (@x) + i - 1] : 4);
  WriteLn
end.
```

See also

Chapter 9 [Keywords], page 453, [record], page 399, Section 6.7 [Type Casts], page 85.

abstract

Not yet implemented.

Synopsis

Description

Abstract object type or method declaration.

Conforming to

'abstract' is an Object Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Addr

Synopsis

```
function Addr (const Foo): Pointer;
```

Description

'Addr' returns the address of its argument. It is equivalent to the address operator.

Note: In BP, 'Addr' returns an untyped pointer. GPC does this only with '--borland-pascal'. Otherwise it returns a typed pointer. 'Addr' never depends on the '--[no]-typed-address' option/compiler directive, unlike the address operator. (It is recommended you never rely on untyped pointer results, but use a type-cast if really necessary.)

Conforming to

'Addr' is a Borland Pascal extension.

Example

```
program AddrDemo;
var
  Foo: ^Integer;
  Bar: Integer;
begin
  Foo := Addr (Bar);  { Let 'Foo' point to 'Bar'. }
  Bar := 17;
  Foo^ := 42;  { Change the value of 'Bar' to 42 }
  WriteLn (Bar)
end.
```

See also

Section 6.3 [Operators], page 82.

AlignOf

Synopsis

```
function AlignOf (var x): Integer;
```

Description

Returns the alignment of a type or variable in bytes.

Conforming to

'AlignOf' is a GNU Pascal extension.

Example

```
program AlignOfDemo;
var
  a: Integer;
  b: array [1 .. 8] of Char;
begin
  WriteLn (AlignOf (a));  { Alignment of 'Integer', e.g. 4 bytes. }
  WriteLn (AlignOf (Integer));  { The same. }
  WriteLn (AlignOf (b));  { Alignment of 'Char'; usually 1 byte. }
end.
```

Although the array is bigger than a single char, it is accessed char by char, so there usually is no need to align it on a 4 byte boundary or such. (This may be false on some platforms.)

See also

[SizeOf], page 421, [BitSizeOf], page 278, [TypeOf], page 438.

all

(Under construction.)

Synopsis

Description

'export' extension ('export foo = all').

Conforming to

'all' is a GNU Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

and

Synopsis

```
operator and (operand1, operand2: Boolean) = Result: Boolean;
```

or

```
operator and (operand1, operand2: integer_type) = Result: integer_type;
```

or

```
procedure and (var operand1: integer_type; operand2: integer_type);
```

Description

In GNU Pascal, '`and`' has three built-in meanings:

1. Logical "and" between two '`Boolean`'-type expressions. The result of the operation is of '`Boolean`' type.

   By default, '`and`' acts as a short-circuit operator in GPC: If the first operand is '`False`', the second operand is not evaluated because the result is already known to be '`False`'. You can change this to complete evaluation using the '`--no-short-circuit`' command-line option or the '`{$B+}`' compiler directive.
2. Bitwise "and" between two integer-type expressions. The result is of the common integer type of both expressions.
3. Use as a "procedure": '`operand1`' is "and"ed bitwise with '`operand2`'; the result is stored in '`operand1`'.

Conforming to

The logical '`and`' operator is defined in ISO 7185 Pascal.

According to ISO, you cannot rely on '`and`' being a short-circuit operator. On the other hand, GPC's default behaviour does *not* contradict the ISO standard. (See [and_then], page 264.) However, since it seems to be a de-facto standard among ISO Pascal compilers to evaluate both operands of '`and`', GPC switches to '`--no-short-circuit`' mode if one of the language dialect options selecting ISO Pascal, for instance '`--extended-pascal`', is given. Use '`--short-circuit`' to override.

Use of '`and`' as a bitwise operator for integers is a Borland Pascal extension.

Use of '`and`' as a "procedure" is a GNU Pascal extension.

Example

```
program AndDemo;
var
  a, b, c: Integer;
begin
  if (a = 0) and (b = 0) then  { logical 'and' }
```

```
      c := 1
    else if (a and b) = 0 then  { bitwise 'and' }
      c := 2
    else
      and (c, a)  { same as 'c := c and a' }
  end.
```

Note the difference between the logical '`and`' and the bitwise '`and`': When '`a`' is 2 and '`b`' is 4, then '`a and b`' is 0. **Beware:** '`a and b = 0`' has nothing to do with '`(a = 0) and (b = 0)`'!

Since bitwise '`and`' has a higher priority than the '`=`' operator, parentheses are needed in '`if (a = 0) and (b = 0)`' because otherwise '`0 and b`' would be calculated first, and the remainder would cause a parse error.

See also

Chapter 9 [Keywords], page 453, [and_then], page 264, [and then], page 263, [or], page 375, [xor], page 451, Section 6.3 [Operators], page 82.

and then

Synopsis

```
{ 'and then' is built in. A user-defined operator cannot consist of
   two words. }
operator and then (operand1, operand2: Boolean) = Result: Boolean;
```

Description

'`and then`' is an alias for the short-circuit logical operator '`and_then`'.

Conforming to

While '`and_then`' is defined in ISO 10206 Extended Pascal, '`and then`' is a GNU Pascal extension.

Example

```
program AndThenDemo;
var
  p: ^Integer;
begin
  New (p);
  ReadLn (p^);
  if (p <> nil) and then (p^ < 42) then  { This is safe. }
    WriteLn (p^, ' is less than 42')
end.
```

See also

Chapter 9 [Keywords], page 453, [and_then], page 264, [and], page 262, [or else], page 377.

and_then

Synopsis

```
operator and_then (operand1, operand2: Boolean) = Result: Boolean;
```

Description

The 'and_then' short-circuit logical operator performs the same operation as the logical operator 'and'. But while the ISO standard does not specify anything about the evaluation of the operands of 'and' – they may be evaluated in any order, or not at all – 'and_then' has a well-defined behaviour: It evaluates the first operand. If the result is 'False', 'and_then' returns 'False' without evaluating the second operand. If it is 'True', the second operand is evaluated and returned.

Since the behaviour described above is the most efficient way to implement 'and', GPC by default treats 'and' and 'and_then' exactly the same. If you want, for some reason, to have both operands of 'and' evaluated completely, you must assign both to temporary variables and then use 'and' – or 'and_then', it does not matter.

Conforming to

'and_then' is an ISO 10206 Extended Pascal extension.

Some people think that the ISO standard requires both operands of 'and' to be evaluated. This is false. What the ISO standard *does* say is that you cannot rely on a certain order of evaluation of the operands of 'and'; in particular things like the following program can crash according to ISO Pascal, although they cannot crash when compiled with GNU Pascal running in default mode.

```
program AndBug;
var
  p: ^Integer;
begin
  New (p);
  ReadLn (p^);
  if (p <> nil) and (p^ < 42) then  { This is NOT safe! }
    WriteLn ('You''re lucky. But the test could have crashed ...')
end.
```

Example

```
program And_ThenDemo;
var
  p: ^Integer;
begin
  New (p);
  ReadLn (p^);
  if (p <> nil) and_then (p^ < 42) then  { This is safe. }
    WriteLn (p^, ' is less than 42')
end.
```

See also

Chapter 9 [Keywords], page 453, [and then], page 263, [and], page 262, [or_else], page 378.

AnsiChar

Synopsis

```
type
  AnsiChar = Char;
```

Description

'AnsiChar' is an 8 bit char type. Currently, it is the same as 'Char', but this might change in the future, once 'wide chars' (16 bit chars) will be introduced into GPC. Depending on the platform, 'Char' might be either 'AnsiChar' or 'WideChar' then.

Conforming to

'AnsiChar' is a Borland Delphi extension.

Example

```
program AnsiCharDemo;
var
  A: AnsiChar;  { There is nothing special with ‘AnsiChar’. }
  B: Char;
begin
  A := 'A';
  A := B
end.
```

See also

[PAnsiChar], page 382, [Char], page 288.

AnyFile

Synopsis

```
type
  AnyFile  { built-in type }
```

Description

'AnyFile' is a built-in type that can only be used for parameters and pointer targets. Any kind of file variable ('Text', untyped and typed 'file') can be passed to such a parameter and their address assigned to such a pointer. On the other side, only generic file operations are possible with 'AnyFile' parameters/pointer targets.

This type is useful for implementing generic file handling routines. Also some built-in file routines use this type for their parameters, e.g. 'IOSelectRead' (see Section 6.14 [Run Time System], page 105).

'BlockRead' (see [BlockRead], page 279) and 'BlockWrite' (see [BlockWrite], page 280) treat 'AnyFile' specially, in that they accept all 'AnyFile's as arguments (even if the actual file is a typed or 'Text' file) and always use a block size of 1 (even if the actual file is an untyped file with different block size or a typed file of a type with size not equal to one). This is the only way to reliably read/write a certain amount of data from/to an 'AnyFile'.

'AnyFile' pointers cannot be allocated with 'New' (because it would be unspecified which kind of file to create).

Conforming to

'AnyFile' is a GNU Pascal extension.

Example

```
program AnyFileDemo;

procedure Test (var f: AnyFile);
var v: ^AnyFile;
begin
  { Generic file operations are allowed for 'AnyFile' }
  Rewrite (f);

  { 'AnyFile' can also be accessed via pointers }
  v := @f;
  Close (v^)
end;

var
  t: Text;
  f: file;
  g: file of Integer;

begin
  { Any kind of file variable can be passed as 'AnyFile' }
  Test (t);
  Test (f);
  Test (g)
end.
```

See also

[Text], page 429, [file], page 322.

Append

Synopsis

```
procedure Append (var F: any_file; [FileName: String;]
                                    [BlockSize: Cardinal]);
```

Description

'Append' opens a file for writing. If the file does not exist, it is created. If it does exist, the file pointer is positioned after the last element.

Like '`Rewrite`', '`Reset`' and '`Extend`' do, '`Reset`' accepts an optional second parameter for the name of the file in the filesystem and a third parameter for the block size of the file. The third parameter is allowed only (and by default also required) for untyped files. For details, see [Rewrite], page 405.

Conforming to

'`Append`', including the '`BlockSize`' parameter, is a Borland Pascal extension. ISO 10206 Extended Pascal has [Extend], page 318 instead. The '`FileName`' parameter is a GNU Pascal extension.

Example

```
program AppendDemo;
var
  Sample: Text;
begin
  Assign (Sample, 'sample.txt');
  Rewrite (Sample);
  WriteLn (Sample, 'Hello, World!');  { 'sample.txt' now has one line }
  Close (Sample);

  { ... }

  Append (Sample);
  WriteLn (Sample, 'Hello again!');  { 'sample.txt' now has two lines }
  Close (Sample)
end.
```

See also

[Assign], page 272, [Reset], page 402, [Rewrite], page 405, [Update], page 441, [Extend], page 318.

ArcCos

Synopsis

```
function ArcCos (x: Real): Real;
```

or

```
function ArcCos (z: Complex): Complex;
```

Description

'`ArcCos`' returns the (principal value of the) arcus cosine of the argument. The result is in the range '`0 < ArcCos (x) < Pi`' for real arguments.

Conforming to

'`ArcCos`' is a GNU Pascal extension.

Example

```
program ArcCosDemo;
begin
  { yields 3.14159 as ArcCos (0.5) = Pi / 3 }
  WriteLn (3 * ArcCos (0.5) : 0 : 5)
end.
```

See also

[ArcSin], page 268, [ArcTan], page 268, [Sin], page 420, [Cos], page 299, [Ln], page 349, [Arg], page 269.

ArcSin

Synopsis

```
function ArcSin (x: Real): Real;
```

or

```
function ArcSin (z: Complex): Complex;
```

Description

'ArcSin' returns the (principal value of the) arcus sine of the argument. The result is in the range '-Pi / 2 < ArcSin (x) < Pi / 2' for real arguments.

Conforming to

'ArcSin' is a GNU Pascal extension.

Example

```
program ArcSinDemo;
begin
  { yields 3.14159 as ArcSin (0.5) = Pi / 6 }
  WriteLn (6 * ArcSin (0.5) : 0 : 5)
end.
```

See also

[ArcCos], page 267, [ArcTan], page 268, [Sin], page 420, [Cos], page 299, [Ln], page 349, [Arg], page 269.

ArcTan

Synopsis

```
function ArcTan (x: Real): Real;
```

or

```
function ArcTan (z: Complex): Complex;
```

Description

'ArcTan' returns the (principal value of the) arcus tangent of the argument. The result is in the range '-Pi / 2 < ArcTan (x) < Pi / 2' for real arguments.

Conforming to

'ArcTan' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program ArcTanDemo;
begin
  { yields 3.14159 as ArcTan (1) = Pi / 4 }
  WriteLn (4 * ArcTan (1) : 0 : 5)
end.
```

See also

[ArcSin], page 268, [ArcCos], page 267, [Sin], page 420, [Cos], page 299, [Ln], page 349, [Arg], page 269.

Arg

Synopsis

```
function Arg (z: Complex): Real;
```

Description

'Arg' returns the complex "argument", i.e. the angle (in radian) in the complex plane with respect to the real axis, of its parameter 'z'. The result is in the range of '-Pi < Arg (z) <= Pi'.

Conforming to

'Arg' is an ISO 10206 Extended Pascal extension.

Example

```
program ArgDemo;
var
  z: Complex;
begin
  z := Cmplx (1, 1);  { 1 + i }
  WriteLn (Arg (z) : 0 : 5)  { yields 0.78540, i.e. Pi / 4 }
end.
```

See also

[ArcTan], page 268, [Ln], page 349, [Polar], page 387.

array

Synopsis

In type definitions:

`array [`*index_type*`] of` *element_type*

or

`array [`*index_type*`, ...,` *index_type*`] of` *element_type*

In parameter list declarations:

`array of` *element_type*

Description

The reserved word '`array`' is used to define an array type.
`@@`conformant/open arrays

Conforming to

Array types are defined in ISO 7185 Pascal.

Example

```
program ArrayDemo;
type
  IntArray = array [1 .. 20] of Integer;
  WeekDayChars = array [(Mon, Tue, Wed, Thu, Fri, Sat, Sun)] of Char;
  Foo = array [0 .. 9, 'a' .. 'z', (Baz, Glork1, Fred)] of Real;
  TwoDimIntArray = array [1 .. 10] of IntArray;
  { is equivalent to: }
  TwoDimIntArray2 = array [1 .. 10, 1 .. 20] of Integer;

procedure PrintChars (F: array of Char);
var
  i: Integer;
begin
  for i := Low (F) to High (F) do
    WriteLn (F[i])
end;

var
  Waldo: WeekDayChars;

begin
  Waldo := 'HiWorld';
  PrintChars (Waldo)
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.2.11.2 [Array Types], page 70, [High], page 333, [Low], page 356

as

(Under construction.)

Synopsis

Description

Object type membership test and conversion.

Conforming to

'`as`' is an Object Pascal and a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [is], page 345, [TypeOf], page 438, Section 6.8 [OOP], page 86.

asm

(Under construction.)

Synopsis

Description

See '`http://www.gnu-pascal.de/contrib/misc/gpcasm.zip`'.

Conforming to

'`asm`', as implemented in GPC, is a GNU Pascal extension. It is mostly compatible to GCC's '`asm`', but not compatible to that of Borland Pascal.

Example

See also

Chapter 9 [Keywords], page 453.

asmname

Synopsis

Description

Deprecated! Use '`external name`' now.

Conforming to

Example

See also

Chapter 9 [Keywords], page 453, [external], page 320, [name], page 366, Section 6.11.1 [Importing Libraries from Other Languages], page 100.

Assert

Synopsis

```
procedure Assert (Condition: Boolean);
```

or

```
procedure Assert (Condition: Boolean; const Message: String);
```

Description

'`Assert`' checks the given '`Condition`' at run-time. If it is true, it does nothing. If it is false, it raises a runtime error, using the second argument for the message if given.

However, if the switch '`--no-assertions`' is given (see Section 5.1 [GPC Command Line Options], page 35), '`Assert`' is deactivated. It still evaluates the condition if it has side effects, but never raises a runtime error.

Conforming to

'`Assert`' is a Borland Delphi extension.

Example

See also

[CompilerAssert], page 293.

Assign

(Under contruction.)

Synopsis

```
procedure Assign (var F: any_file; FileName: String);
```

Description

Conforming to

'`Assign`' is a Borland Pascal extension.

Example

See also

[Reset], page 402, [Rewrite], page 405, [Update], page 441, [Extend], page 318, [Append], page 266.

Assigned

(Under construction.)

Synopsis

```
function Assigned (p: Pointer): Boolean;
```

or

```
function Assigned (p: procedural_type): Boolean;
```

Description

The '`Assigned`' function returns '`True`' if the pointer parameter or the address of the procedural parameter is not '`nil`'; it returns '`False`' if it is '`nil`'.

Conforming to

'`Assigned`' is a Borland Pascal extension.

Example

```
program AssignedDemo;
type
  PInt = ^Integer;

procedure TellIfOdd (p: PInt);
begin
  if Assigned (p) and then Odd (p^) then
    WriteLn ('The pointer p points to an odd value.')
end;

var
  foo: Integer;
begin
  TellIfOdd (nil);
  foo := 1;
  TellIfOdd (@foo);
  foo := 2;
  TellIfOdd (@foo)
end.
```

See also

[Null], page 372, [nil], page 370, [Pointer], page 386.

attribute

(Under construction.)

Synopsis

declaration `attribute (`*name*`);`

or

declaration `attribute (`*name* `=` *parameter*`);`

or

declaration `attribute (`*name* `(`*parameter*`,` *parameter* `...));`

Description

Several attributes can be given in one '`attribute`' directive, separated with '`,`', or in several '`attribute`' directives.

Besides the attributes that GCC supports (see section "Attribute Syntax" in *the GCC manual*), GPC allows the following attributes for variables:

- static
- register
- volatile
- const
- external
- name (with a string constant parameter)

For routines it allows the following additional attributes:

- ignorable
- inline
- iocritical
- name (with a string constant parameter)

For types it allows the following additional attributes:

- iocritical (for procedural [pointer] types)
- size (with an integer constant parameter)

'`Size`' can be applied to integer and Boolean types to produce types with a specified size in bits; for example

```
type
  Card16 = Cardinal attribute (Size = 16);
```

defines an unsigned integer type with 16 bits.

Variable and routine attributes are preceded by a '`;`', type attributes are not. So, e.g., in the following example, the '`Size`' attribute applies to the type, and the '`static`' attribute to the variable.

```
var a: Integer attribute (Size = 64); attribute (static);
```

Conforming to

'`attribute`' is a GNU Pascal extension.

Example

```
program AttributeDemo;

{ Demo for 'iocritical' attribute. }

{ Program will abort with a runtime error! }

{$I-}
procedure p; attribute (iocritical);
var t: Text;
begin
  Reset (t)  { Will not cause a runtime error here because I/O
               checking is off, but leave InOutRes set. }
end;
{$I+}

begin

  p;
  { Since 'p' was declared 'iocritical', and I/O checking is now on,
    InOutRes is checked immediately after the call to p, and a
    runtime error raised. }

  { So this statement is never reached. }
  InOutRes := 0;

  { Neither this one, which would be reached without the
    'iocritical' attribute. }
  WriteLn ('never gets here')

end.
```

See also

Chapter 9 [Keywords], page 453, [external], page 320.

begin

Synopsis

```
begin
  statement;
  statement;
  ...
  statement
end;
```

Description

The reserved word '`begin`' opens a '`begin ... end`' statement which joins several *statements* to one compound statement.

Conforming to

'begin' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program BeginDemo;
begin
  if True then
    WriteLn ('single statement');
  if True then
    begin                     { clamp statement1 ... }
      WriteLn ('statement1');
      WriteLn ('statement2')
    end                       { ... to statement2 }
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.2 [begin end Compound Statement], page 56, [end], page 311

Bind

(Under construction.)

Synopsis

```
procedure Bind (var F: any_file; B: BindingType);
```

Description

Conforming to

'Bind' is an ISO 10206 Extended Pascal extension.

Example

See also

[Unbind], page 439, [Binding], page 277, [BindingType], page 277, [bindable], page 276.

bindable

(Under construction.)

Synopsis

Description

External bindability of files.

Conforming to

'bindable' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453, [Bind], page 276, [Unbind], page 439, [Binding], page 277, [BindingType], page 277.

Binding

(Under construction.)

Synopsis

```
function Binding (F: any_file): BindingType;
```

Description

Conforming to

'Binding' is an ISO 10206 Extended Pascal extension.

Example

See also

[Bind], page 276, [Unbind], page 439, [BindingType], page 277, [bindable], page 276.

BindingType

(Under construction.)

Synopsis

```
type
  UnixTimeType = LongInt;
  BindingType = packed record
    Bound,
    Force,                     { Can be set to allow binding to
                                 directories or inaccessible files }
    Extensions_Valid,
    Readable,
    Writable,
    Executable,
    Existing,                  { Binding points to an existing file }
    Directory,                 { Binding points to an existing
                                 directory; 'Existing' is False
```

```
                                  in this case }
    Special,                    { Binding points to an existing
                                  special file (device, pipe, socket,
                                  etc.); ‘Existing’ is False in
                                  this case }
    SymLink: Boolean;           { Binding points to a symbolic link }
    Size: FileSizeType;         { Size of file, or -1 if unknown }
    AccessTime,                 { Time of last access }
    ModificationTime,           { Time of last modification }
    ChangeTime: UnixTimeType;   { Time of last change }
    User,                       { User ID of owner }
    Group,                      { Group ID of owner }
    Mode,                       { Access permissions, cf. ChMod }
    Device,                     { Device the file is on }
    INode,                      { Unix INode number }
    Links: Integer;             { Number of hard links }
    TextBinary: Boolean;        { Open a Text file in binary mode }
    Handle: Integer;            { Can be set to bind a Pascal file to
                                  a given file handle }
    CloseFlag: Boolean;         { If Handle is used, tell whether to
                                  close it when file is closed }
    Name: String (BindingNameLength)
  end;
```

`BindingNameLength` is an implementation-defined constant.

Description

Conforming to

‘`BindingType`’ is an ISO 10206 Extended Pascal extension. The fields ‘`Bound`’ and ‘`Name`’ are required by Extended Pascal, the other ones are GNU Pascal extensions.

Example

See also

[Bind], page 276, [Unbind], page 439, [Binding], page 277, [bindable], page 276.

BitSizeOf

Synopsis

```
function BitSizeOf (var x): SizeType;
```

Description

Returns the size of a type or variable in bits.

Conforming to

‘`BitSizeOf`’ is a GNU Pascal extension.

Example

```
program BitSizeOfDemo;
type
  Int12 = Integer attribute (Size = 12);
var
  a: Integer;
  b: array [1 .. 8] of Char;
  c: Int12;
  d: packed record
       x: Int12;
       y: 0 .. 3
     end;
begin
  WriteLn (BitSizeOf (a));    { Size of an ‘Integer’; usually 32 bits. }
  WriteLn (BitSizeOf (Integer));  { The same. }
  WriteLn (BitSizeOf (b));    { Size of eight ‘Char’s; usually 64 bits. }
  WriteLn (BitSizeOf (c));    { e.g. 16 bits (smallest addressable
                                 space holding a 12 bit integer). }
  WriteLn (BitSizeOf (d));    { e.g. 16 }
  WriteLn (BitSizeOf (d.x));  { 12 }
  WriteLn (BitSizeOf (d.y))   { 2 }
end.
```

See also

[SizeOf], page 421, [AlignOf], page 261, [TypeOf], page 438.

BlockRead

(Under construction.)

Synopsis

```
procedure BlockRead (var F: File; var Buffer; Blocks: Integer);
```

or

```
procedure BlockRead (var F: File; var Buffer; Blocks: Integer;
                     var BlocksRead: Integer);
```

Description

Conforming to

‘BlockRead’ is a UCSD Pascal extension.

Example

See also

BlockWrite

(Under construction.)

Synopsis

```
procedure BlockWrite (var F: File; const Buffer; Blocks: Integer);
```

or

```
procedure BlockWrite (var F: File; const Buffer; Blocks: Integer;
                      var BlocksWritten: Integer);
```

Description

Conforming to

'BlockWrite' is a UCSD Pascal extension.

Example

See also

Boolean

Synopsis

```
type
  Boolean = (False, True); { built-in type }
```

Description

The intrinsic 'Boolean' represents boolean values, i.e. it can only assume the two values 'True' and 'False' (which are predefined constants).

Conforming to

'Boolean' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program BooleanDemo;
var
  a: Boolean;
begin
  a := True;
  WriteLn (a)
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [ShortBool], page 416, [MedBool], page 360, [WordBool], page 449, [LongBool], page 350, [LongestBool], page 351.

Break

Synopsis

```
Break  { simple statement }
```

Description

With 'Break' you can exit the body of the current loop instantly. It can only be used within a 'while', 'repeat' or a 'for' loop.

Conforming to

'Break' is a Borland Pascal extension. Mac Pascal has 'Leave' instead.

Example

```
program BreakDemo;
var
  Foo: Integer;
begin
  while True do
    begin
      repeat
        WriteLn ('Enter a number less than 100:');
        ReadLn (Foo);
        if Foo < 100 then
          Break;  { Exits 'repeat' loop }
        WriteLn (Foo, ' is not exactly less than 100! Try again ...')
      until False;
      if Foo > 50 then
        Break;  { Exits 'while' loop }
      WriteLn ('The number entered was not greater than 50.')
    end
end.
```

See also

Section 6.1.7.13 [Loop Control Statements], page 60, [Continue], page 298, [Cycle], page 303, [Exit], page 315, [Halt], page 333, [Leave], page 347, [Return], page 404, [goto], page 331.

Byte

Synopsis

```
type
  Byte  { built-in type }
```

Description

'Byte' is an unsigned integer type which is one "unit" wide. On most platforms one unit has 8 bits, therefore the type is named "byte" and usually has a range of '0 .. 255'. (It is the same as [ByteCard], page 283.)

'Byte' in GNU Pascal is compatible to 'unsigned char' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'Byte' is a Borland Pascal extension. (For something equivalent in ISO Pascal, see Section 6.2.11.1 [Subrange Types], page 70.)

Example

```
program ByteDemo;
var
  a: Byte;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

ByteBool

Synopsis

```
type
  ByteBool = Boolean attribute (Size = BitSizeOf (Byte));
```

Description

The intrinsic 'ByteBool' represents boolean values, but occupies the same memory space as a 'Byte'. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

'ByteBool' is a Borland Pascal extension.

Example

```
program ByteBoolDemo;
var
  a: ByteBool;
begin
  Byte (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ShortBool], page 416, [MedBool], page 360, [WordBool], page 449, [LongBool], page 350, [LongestBool], page 351.

ByteCard

Synopsis

```
type
  ByteCard = Cardinal attribute (Size = BitSizeOf (Byte));
```

Description

'ByteCard' is an unsigned integer type which is one "unit" wide. On most platforms one unit has 8 bits, therefore the type is prefixed "byte-" and usually has a range of '0 .. 255'.

'ByteCard' in GNU Pascal is compatible to 'unsigned char' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'ByteCard' is a GNU Pascal extension.

Example

```
program ByteCardDemo;
var
  a: ByteCard;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

ByteInt

Synopsis

```
type
  ByteInt = Integer attribute (Size = BitSizeOf (Byte));
```

Description

'ByteInt' is a signed integer type which is one "unit" wide. On most platforms one unit has 8 bits, therefore the type is prefixed "byte-" and usually has a range of '-128 .. 127'.

'ByteInt' in GNU Pascal is compatible to 'signed char' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'`ByteInt`' is a GNU Pascal extension.

'`ByteInt`' in GNU Pascal corresponds to [ShortInt], page 417 in Borland Pascal.

Example

```
program ByteIntDemo;
var
  a: ByteInt;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

c

Synopsis

Description

Deprecated! Use '`external`' now.

Conforming to

Example

See also

Chapter 9 [Keywords], page 453, Section 6.11.1 [Importing Libraries from Other Languages], page 100, [external], page 320.

Card

Synopsis

```
function Card (S: any_set): Integer;
```

Description

The function '`Card (S)`' returns the number of elements in the set '`S`'.

Conforming to

'`Card`' is an ISO 10206 Extended Pascal extension.

Example

```
program CardDemo;
var
  Foo: set of 1 .. 100;
begin
  Foo := [1, 2, 3, 5, 1, 1, 1, 2, 2, 2, 3, 3, 5, 5];  { four elements }
  WriteLn ('foo consists of ', Card (Foo), ' elements')
end.
```

See also

[set], page 412

Cardinal

Synopsis

```
type
  Cardinal  { built-in type }
```

Description

'`Cardinal`' is the "natural" unsigned integer type in GNU Pascal. On some platforms it is 32 bits wide and thus has a range of '`0 .. 4294967295`'. Use it whenever you need a general-purpose unsigned integer type and don't need to care about compatibility to other Pascal dialects.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'`Cardinal`' is not defined in ISO Pascal, but several Pascal compilers support it as an extension. In Borland Delphi, for instance, it is an unsigned 16-bit in version 1.0, an unsigned 31-bit integer from version 2.0 on, and an unsigned 32-bit integer from version 4.0 on.

Example

```
program CardinalDemo;
var
  a: Cardinal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

case

Synopsis

```
case expression of
  selector: statement;
  ...
  selector: statement;
end;
```

or, with alternative statement sequence:

```
case expression of
  selector: statement;
  ...
  selector: statement;
otherwise  { ``else'' instead of ``otherwise'' is allowed }
  statement;
  ...
  statement;
end;
```

or, as part of the invariant `record` type definition:

```
foo = record
  field_declarations
case bar: variant_type of
  selector: (field_declarations);
  selector: (field_declarations);
  ...
end;
```

or, without a variant selector field,

```
foo = record
  field_declarations
case variant_type of
  selector: (field_declarations);
  selector: (field_declarations);
  ...
end;
```

Description

'`case`' opens a case statement. For further description see Section 6.1.7.4 [case Statement], page 56.

For '`case`' in a variant record type definition, see Section 6.2.11.3 [Record Types], page 71.

Conforming to

The '`case`' statement is defined in ISO 7185 Pascal and supported by all known Pascal variants.

According to ISO 7185 Pascal, the selector type must be a named type. UCSD Pascal and Borland Pascal allow any ordinal type here.

The alternative statement execution with '`otherwise`' it is an Extended Pascal extension; with '`else`' it is a Borland Pascal extension. In GNU Pascal, both are allowed.

Example

```
program CaseDemo;
var
  Foo: String (10);
  Bar: Integer;
begin
  WriteLn ('Enter up to ten arbitrary characters:');
  ReadLn (Foo);
  for Bar := 1 to Length (Foo) do
    begin
      Write (Foo[Bar], ' is ');
      case Foo[Bar] of
        'A' .. 'Z', 'a' .. 'z':
          WriteLn ('an English letter');
        '0' .. '9':
          WriteLn ('a number');
      otherwise
        WriteLn ('an unrecognized character')
      end
    end
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.3 [if Statement], page 56, Section 6.2.11.3 [Record Types], page 71, [else], page 310, [otherwise], page 379

CBoolean

(Under construction.)

Synopsis

```
type
  CBoolean  { built-in type }
```

Description

'`CBoolean`' is a Boolean type. In GNU Pascal it is compatible to '`_Bool`' in GNU C (which is defined as '`bool`' in '`stdbool.h`'). This compatibility is the reason why '`CBoolean`' exists.

Conforming to

'`CBoolean`' is a GNU Pascal extension.

Example

```
program CBooleanDemo;
var
  a: CBoolean;
begin
```

```
  a := True;
  if Ord (a) = 1 then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, Section 6.2.3 [Integer Types], page 64, [Boolean], page 280, [True], page 434, [False], page 320, [ByteBool], page 282, [ShortBool], page 416, [MedBool], page 360, [WordBool], page 449, [LongBool], page 350, [LongestBool], page 351.

CCardinal

Synopsis

```
type
  CCardinal  { built-in type }
```

Description

'CCardinal' is an unsigned integer type. On some platforms it is 32 bits wide and thus has a range of '0 .. 4294967295'.

'CCardinal' in GNU Pascal is compatible to 'unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'CCardinal' is a GNU Pascal extension.

Example

```
program CCardinalDemo;
var
  a: CCardinal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

Char

Synopsis

```
type
  Char  { built-in type }
```

Description

The built-in type 'Char' holds elements of the operating system's character set (usually ASCII). The 'Char' type is a special case of ordinal type. Conversion between character types and ordinal types is possible with the built-in functions 'Ord' and 'Chr'.

Conforming to

'Char' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program CharDemo;
var
  a: Char;
begin
  a := 'x';
  WriteLn (a)
end.
```

See also

Section 6.2.6 [Character Types], page 68, Section 6.2.2 [Ordinal Types], page 64, Section 6.7 [Type Casts], page 85, [Ord], page 377, [Chr], page 290.

ChDir

Synopsis

```
procedure ChDir (Directory: String);
```

Description

'ChDir' changes the current directory to *Directory*, if its argument is a valid parameter to the related operating system's function. Otherwise, a runtime error is caused.

Conforming to

'ChDir' is a Borland Pascal extension.

Example

```
program ChDirDemo;
var
  Foo: String (127);
begin
  WriteLn ('Enter directory name to change to:');
  ReadLn (Foo);
  {$I-}  { Don't abort the program on error }
  ChDir (Foo);
  if IOResult <> 0 then
    WriteLn ('Cannot change to directory `', Foo, '''.')
```

```
  else
    WriteLn ('Okay.')
end.
```

See also

[MkDir], page 364, [RmDir], page 406

Chr

Synopsis

```
function Chr (AsciiCode: Integer): Char;
```

Description

'Chr' returns a character whose ASCII code corresponds to the value given by 'AsciiCode'.

Conforming to

'Chr' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program ChrDemo;
var
  x: Integer;
begin
  for x := 32 to 122 do
    Write (Chr (x))
end.
```

See also

Section 6.2.6 [Character Types], page 68, [Ord], page 377, [Char], page 288

CInteger

Synopsis

```
type
  CInteger  { built-in type }
```

Description

'CInteger' is a signed integer type. On some platforms it is 32 bits wide and thus has a range of '-2147483648 .. 2147483647'.

'CInteger' in GNU Pascal is compatible to 'int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'CInteger' is a GNU Pascal extension.

Example

```
program CIntegerDemo;
var
  a: CInteger;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

c_language

Synopsis

Description

Deprecated! Use 'external' now.

Conforming to

Example

See also

Chapter 9 [Keywords], page 453, Section 6.11.1 [Importing Libraries from Other Languages], page 100, [external], page 320.

class

Not yet implemented.

Synopsis

Description

OOE/Delphi style object class.

Conforming to

'class' is an Object Pascal and a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453.

Close

(Under construction.)

Synopsis

```
procedure Close (var F: any_file);
```

Description

Conforming to

'Close' is a UCSD Pascal extension.

Example

See also

Cmplx

Synopsis

```
function Cmplx (RealPart, ImaginaryPart: Real): Complex;
```

Description

'Cmplx' makes a complex number from 'RealPart' and 'ImaginaryPart'.

Conforming to

'Cmplx' is an ISO 10206 Extended Pascal extension.

Example

```
program CmplxDemo;
var
  z: Complex;
  x, y: Real;
begin
  z := Cmplx (x, y)  { z := x + iy }
end.
```

See also

[Re], page 397, [Im], page 335, [Polar], page 387, [Arg], page 269

Comp

Synopsis

```
type
  Comp = LongInt;
```

Description

'Comp' is a signed integer type which is longer than 'Integer'. On some platforms it is 64 bits wide and thus has a range of '-9223372036854775808 .. 9223372036854775807'.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'Comp' is a Borland Pascal extension.

In some contexts, Borland Pascal treats 'Comp' as a "real" type – this behaviour is not supported by GPC.

Example

```
program CompDemo;
var
  a: Comp;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

CompilerAssert

Synopsis

```
procedure CompilerAssert (Condition: Boolean);
```

or

```
function CompilerAssert (Condition: Boolean): Boolean;
```

or

```
function CompilerAssert (Condition: Boolean;
           ResultValue: Any_Type): type of ResultValue;
```

Description

'CompilerAssert' checks the given 'Condition' at compile-time. If it is a compile-time constant of Boolean type with the value 'True', it returns 'ResultValue', or if called with only one argument, it returns 'True' or nothing if used as a procedure.

If 'Condition' cannot be evaluated at compile-time or does not have the value 'True', it causes a compile-time error.

So it can be used to make sure that certain assumptions hold before relying on them.

'CompilerAssert' does not depend on the '--[no-]assertions' options. It does not generate any run-time code.

Conforming to

'CompilerAssert' is a GNU Pascal extension.

Example

```
program CompilerAssertDemo;

var
  a: LongInt;

const
  { Make sure that the highest value a can hold is larger than
    MaxInt, and set b to that value. }
  b = CompilerAssert (High (a) > MaxInt, High (a));

  { Do a similar check for the minimum value, setting c to True
    (which can be ignored). }
  c = CompilerAssert (Low (a) < Low (Integer));

begin
  { Procedure-like use of CompilerAssert in the statement part. }
  CompilerAssert (MaxInt >= 100000);

  WriteLn (b, ' ', c)
end.
```

See also

[Assert], page 272.

Complex

(Under construction.)

Synopsis

```
type
  Internal_Complex = record  { not visible }
    RealPart, ImaginaryPart: Real
  end;
  Complex = restricted Internal_Complex;
```

Description

Conforming to

'Complex' is an ISO 10206 Extended Pascal extension.

Example

```
program ComplexDemo;
var
  a: Complex;
begin
  a := Cmplx (42, 3);
  WriteLn (Re (a), ' + ', Im (a), ' i')
end.
```

See also

Concat

(Under construction.)

Synopsis

```
function Concat (S1, S2: String): String;
```

or

```
function Concat (S1, S2, S3: String): String;
```

or

```
...
```

Description

Conforming to

'Concat' is a UCSD Pascal extension.

Example

See also

Conjugate

Synopsis

```
function Conjugate (z: Complex): Complex;
```

Description

'Conjugate' computes the complex conjugate of the complex number 'z'

Conforming to

'Conjugate' is a GNU Pascal extension.

Example

```
program ConjugateDemo;
var
  z: Complex;
begin
  z := Cmplx (2, 3);  { z is 2 + i * 3 }
  WriteLn ('z = ', Re (z) : 0 : 5, ' + i * ', Im (z) : 0 : 5);
  z := Conjugate (z);  { z conjugate is 2 - i * 3 }
  WriteLn ('z conjugate = ', Re (z) : 0 : 5,' + i * ', Im (z) : 0 : 5)
end.
```

See also

[Cmplx], page 292, [Re], page 397, [Im], page 335, [Abs], page 257

const

(Under construction.)

Synopsis

Description

Constant declaration or constant parameter declaration.

Conforming to

'const' is defined in ISO 7185 Pascal and supported by all known Pascal variants. 'const' parameters are a Borland Pascal extension. Pointers to 'const' are a GNU Pascal extension.

Constant declarations allow you to define names for constant (unchanging) values, such as using 'SecondsPerHour' instead of 3600. This can make your program much more readable and maintainable.

GNU Pascal allows you to define constant strings, records and arrays as well as simple numeric constants.

GNU Pascal also implements the const parameter extension which allows the compiler to pass parameters by reference while still allowing you to pass constant values as inputs. See Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53 for more information.

@@ Pointers to 'const' @@

Example

```
program ConstDemo;

type
  Rec = record
    x: Integer;
    y: Integer;
  end;

const
  a = 5;
  constr: Rec = (10, 12);

procedure doit (const r: Rec; const s: String);
begin
  WriteLn (r.x);
  WriteLn (r.y);
  WriteLn (s);
end;

var
  variabler: Rec;

begin
  variabler.x := 16;
  variabler.y := 7;
  doit (variabler, 'Should be 16 and 7');
  doit (constr, 'Should be 10 and 12');
end.
```

See also

Chapter 9 [Keywords], page 453, [var], page 444, [protected], page 391, Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53.

constructor

(Under construction.) ;−)

Synopsis

Description

Object constructor.

Conforming to

'`constructor`' is an Object Pascal and a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Continue

Synopsis

```
Continue  { simple statement }
```

Description

'Continue' goes on with loop iteration by jumping to the end of the current loop body. Note: 'Continue' can only stand within a 'while', 'repeat' or a 'for' loop.

Conforming to

'Continue' is a Borland Pascal extension, Mac Pascal has 'Cycle' instead.

Example

```
program ContinueDemo;
var
  Foo, Bar: Integer;
begin
  WriteLn ('Enter three numbers:');
  for Bar := 1 to 3 do
    begin
      ReadLn (Foo);
      if Foo < 5 then
        Continue;
      WriteLn ('Your number was greater than 5.')
    end
end.
```

See also

Section 6.1.7.13 [Loop Control Statements], page 60, [Break], page 281, [Cycle], page 303, [Exit], page 315, [Halt], page 333, [Leave], page 347, [Return], page 404, [goto], page 331.

Copy

Synopsis

```
function Copy (S: String; FirstChar, Count: Integer): String;
```

or

```
function Copy (S: String; FirstChar: Integer): String;
```

Description

'`Copy`' returns a sub-string of '`S`' starting with the character at position *FirstChar*. If *Count* is given, such many characters will be copied into the sub-string. If *Count* is omitted, the sub-string will range to the end of *S*.

If '`Count`' is too large for the sub-string to fit in *S*, the result will be truncated at the end of *S*. If '`FirstChar`' exceeds the length of *S*, the empty string will be returned. (For a function which does not truncate but triggers a runtime error instead, see [SubStr], page 427.)

Please note that GPC's strings may be longer than 255 characters. If you want to isolate the second half of a string *S* starting with the third character, use '`Copy (S, 3)`' instead of '`Copy (S, 3, 255)`'.

Conforming to

'`Copy`' is a UCSD Pascal extension. The possibility to omit the third parameter is a GNU Pascal extension.

Example

```
program CopyDemo;
var
  S: String (42);
begin
  S := 'Hello';
  WriteLn (Copy (S, 2, 3));  { yields 'ell' }
  WriteLn (Copy (S, 3));     { yields 'llo' }
  WriteLn (Copy (S, 4, 7));  { yields 'lo' }
  WriteLn (Copy (S, 42))     { yields the empty string }
end.
```

See also

[SubStr], page 427, Section 6.5 [String Slice Access], page 83.

Cos

Synopsis

```
function Cos (x: Real): Real;
```

or

```
function Cos (z: Complex): Complex;
```

Description

'`Cos`' returns the cosine of the argument. The result is in the range '`-1 < Cos (x) < 1`' for real arguments.

Conforming to

The function '`Cos`' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program CosDemo;
begin
  { yields 0.5 since Cos (Pi / 3) = 0.5 }
  WriteLn (Cos (Pi / 3) : 0 : 5)
end.
```

See also

[ArcTan], page 268, [Sin], page 420, [Ln], page 349, [Arg], page 269.

CString

(Under construction.)

Synopsis

```
type
  CString = ^Char;
```

Description

Conforming to

'CString' is a GNU Pascal extension.

Example

```
program CStringDemo;
var
  s: CString;
begin
  s := 'Hello, world!';
  {$X+}
  WriteLn (s)
end.
```

See also

CString2String

(Under construction.)

Synopsis

```
function CString2String (S: CString): String;
```

Description

Conforming to

'CString2String' is a GNU Pascal extension.

Example

See also

CStringCopyString

(Under construction.)

Synopsis

```
function CStringCopyString (Dest: CString; const Source: String): CString;
```

Description

Conforming to

'CStringCopyString' is a GNU Pascal extension.

Example

See also

CurrentRoutineName

Synopsis

```
function CurrentRoutineName: String;
```

Description

'CurrentRoutineName' returns the name of the current routine from where it's called.

Conforming to

'CurrentRoutineName' is a GNU Pascal extension.

Example

```
program CurrentRoutineNameDemo;

procedure FooBar;
begin
  WriteLn (CurrentRoutineName)  { 'FooBar' }
end;

begin
  WriteLn (CurrentRoutineName);  { 'main program' }
  FooBar
end.
```

See also

CWord

Synopsis

```
type
  CWord = CCardinal;
```

Description

'CCardinal' is an unsigned integer type. On some platforms it is 32 bits wide and thus has a range of '0 .. 4294967295'. It is the same as [CCardinal], page 288.

'CWord' in GNU Pascal is compatible to 'unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'CWord' is a GNU Pascal extension.

Example

```
program CWordDemo;
var
  a: CWord;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[CCardinal], page 288, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

Cycle

Synopsis

```
Cycle  { simple statement }
```

Description

'Cycle' goes on with loop iteration by jumping to the end of the current loop body. Note: 'Cycle' can only stand within a 'while', 'repeat' or a 'for' loop.

Conforming to

'Cycle' is a Mac Pascal extension. Borland Pascal has 'Continue' instead.

Example

```
program CycleDemo;
var
  Foo, Bar: Integer;
begin
  WriteLn ('Enter three numbers:');
  for Bar := 1 to 3 do
    begin
      ReadLn (Foo);
      if Foo < 5 then
        Cycle;
      WriteLn ('Your number was greater than 5.')
    end
end.
```

See also

Section 6.1.7.13 [Loop Control Statements], page 60, [Break], page 281, [Continue], page 298, [Exit], page 315, [Halt], page 333, [Leave], page 347, [Return], page 404, [goto], page 331.

Date

Synopsis

```
function Date (T: TimeStamp): packed array [1 .. Date_length] of Char;
```

Description

Date takes a TimeStamp parameter and returns the date as a string (in the form of a packed array of Char). *Date_length* is an implementation defined invisible constant.

Conforming to

'Date' is an ISO 10206 Extended Pascal extension.

Example

Set [TimeStamp], page 431.

See also

[TimeStamp], page 431, [GetTimeStamp], page 331, [Time], page 430, Section 6.10.8 [Date And Time Routines], page 99.

Dec

Synopsis

For ordinal types:

`procedure Dec (var x:` *ordinal_type*`);`

or

`procedure Dec (var x:` *ordinal_type*`; Amount:` *and_integer_type*`);`

For pointer types:

`procedure Dec (var p:` *any_pointer_type*`);`

or

`procedure Dec (var p:` *any_pointer_type*`; Amount:` *and_integer_type*`);`

Description

For ordinal types, '`Dec`' decreases the value of '`x`' by one or by '`amount`' if specified.

If the argument '`p`' is pointing to a specified type (typed pointer), '`Dec`' decreases the address of '`p`' by the size of the type '`p`' is pointing to or by '`amount`' times that size respectively. If '`p`' is an untyped pointer (i.e. '`p`' is of type [Pointer], page 386), '`p`' is decreased by one, otherwise by '`amount`' if specified.

Conforming to

'`Dec`' is a Borland Pascal extension. The combination of the second argument with application to pointers is a GNU Pascal extension.

Example

```
program DecDemo;
var
  x: Integer;
  y: array [1 .. 5] of Integer;
  p: ^Integer;
begin
  x := 9;
  Dec (x, 10);  { yields -1 }
  {$X+}         { Turn on extended systax }
  p := @y[5];   { p points to y[5] }
  Dec (p, 3)    { p points to y[2] }
end.
```

See also

[Inc], page 338, [Pred], page 388, [Succ], page 428, Section 6.6 [Pointer Arithmetics], page 84.

DefineSize

(Under construction.)

Synopsis

```
procedure DefineSize (var F: any_file; NewSize: Integer);
```

Description

Conforming to

'DefineSize' is a GNU Pascal extension.

Example

See also

Delete

(Under construction.)

Synopsis

```
procedure Delete (var S: String; FirstChar, Count: Integer);
```

or

```
procedure Delete (var S: String; FirstChar: Integer);
```

Description

Conforming to

'Delete' is a UCSD Pascal extension. The possibility to omit the third parameter is a GNU Pascal extension.

Example

See also

destructor

(Under construction.)

Synopsis

Description

Object destructor.

Conforming to

'`destructor`' is an Object Pascal and a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Discard

Synopsis

```
Discard (Value);
```

Description

'`Discard`' does nothing, but tells the compiler that the value given as an argument is not further used. It can be applied, e.g., to routine parameters which are to be ignored, so no warning about them will be given with '`-Wunused`', or when calling a function and ignore its result.

An alternative for the latter case is to give the function the '`ignorable`' attribute. This is useful is the function's result is expected to be ignored regularly. If, however, a result is generally meaningful and only to be ignored in a particular case, using '`Discard`' is preferable.

Conforming to

'`Discard`' is a GNU Pascal extension.

Example

program DiscardDemo;

function Foo (a: Integer): Integer; begin WriteLn (a); Foo := a + 1 end;

{ Parameter 'a' is there only to make the parameter list compatible to that of function 'Foo'. } function Bar (a: Integer): Integer; begin Discard (a); { Tell the compiler that we intentionally do not use 'a' in this function. } Bar := a + 1 end;

var c: Char; f: function (a: Integer): Integer;

begin Write ('With output? '); ReadLn (c); if LoCase (c) = 'y' then f := Foo else f := Bar; Discard (f (42)) { Call the function, but ignore its result } end.

See also

Dispose

(Under construction.)

Synopsis

```
Dispose (PointerVar: Pointer);
```

or

```
Dispose (PointerVar: Pointer; tag_field_values);
```

or

```
Dispose (ObjectPointerVar: Pointer; destructor_call);
```

Description

Conforming to

'Dispose' is defined in ISO 7185 Pascal and supported by most known Pascal variants, but not by UCSD Pascal. Its use for objects is a Borland Pascal extension.

Example

See also

div

Synopsis

```
operator div (p, q: Integer) = r: Integer;
```

Description

Integer division operator.

Conforming to

'div' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program DivDemo;

var
  a, b: Integer;

begin
  a := 16;
  b := 7;
  WriteLn (a div b);  { '2' }
end.
```

See also

Chapter 9 [Keywords], page 453.

do

Synopsis

```
for ... do
  statement
```
or
```
while ... do
  statement
```
or
```
with ... do
  statement
```
or
```
to begin do
  statement
```
or
```
to end do
  statement
```

Description

The '`do`' reserved word is used in combination with other Pascal keywords in many ways. For description and examples see the relevant reference sections: '`for`', '`while`', '`with`', '`to begin`', '`to end`'.

Conforming to

'`do`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See references.

See also

Chapter 9 [Keywords], page 453, [for], page 325, [while], page 447, [with], page 447, [to begin do], page 433, [to end do], page 433.

Double

(Under construction.)

Synopsis

```
type
  Double = Real;
```

Description

'Double' is a synonym for the 'Real' data type and supported for compatibility with other compilers.

Conforming to

'Double' is a Borland Pascal extension.

Example

```
program DoubleDemo;
var
  A: Double;  { There is nothing special with ‘Double’. }
  B: Real;
begin
  A := Pi;
  A := B
end.
```

See also

downto

Synopsis

```
for variable := value1 downto value2 do
  statement
```

Description

The 'downto' reserved word is used in combination with 'for' to build a 'for' loop.

Conforming to

'downto' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

program DowntoDemo; var i: Integer; begin for i := 10 downto 1 do WriteLn (i) end.

See also

Chapter 9 [Keywords], page 453, [for], page 325.

else

Synopsis

As part of the `if ... then ... else` statement:

```
if Boolean_expression then
  statement1
else
  statement2
```

or, as part of the `case ... else` statement:

```
case expression of
  selector: statement;
  ...
  selector: statement
else  { ``otherwise'' instead of ``else'' is allowed }
  statement;
  ...
  statement
end
```

Description

'else' is part of the 'if ... then ... else' statement which provides a possibility to execute statements alternatively. In the `case` statement, 'else' starts a series of statements which is executed if no selector fit in *expression*. In this situation, 'else' is a synonym for `otherwise`.

Conforming to

'else' in 'if' statements is defined in ISO 7185 Pascal and supported by all known Pascal variants. 'else' in 'case' statements is a Borland Pascal extension; ISO 10206 Extended Pascal has 'otherwise' instead.

Example

```
program ElseDemo;
var
  i: Integer;
begin
  Write ('Enter a number: ');
  ReadLn (i);
  if i > 42 then
    WriteLn ('The number is greater than 42')
  else
    WriteLn ('The number is not greater than 42')
end.
```

See also

Chapter 9 [Keywords], page 453, [if], page 334, [case], page 286, [otherwise], page 379.

Empty

(Under construction.)

Synopsis

```
function Empty (var F: any_file): Boolean;
```

Description

Conforming to

'`Empty`' is an ISO 10206 Extended Pascal extension.

Example

See also

end

Synopsis

```
begin
  statement;
  statement;
  ...
  statement
end
```

Description

The reserved word '`end`' closes a '`begin`' ... '`end`'; statement which joins several *statements* together into one compound statement.

`@@` end of a '`case`' statement `@@` end of a record or object declaration `@@` part of a '`to end do`' module destructor

Conforming to

'`end`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program EndDemo;
begin
  if True then
    WriteLn ('single statement');
  if True then
    begin  { clamp statement1 ... }
      WriteLn ('statement1');
      WriteLn ('statement2')
    end    { ... to statement2 }
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.2 [begin end Compound Statement], page 56, [begin], page 275

EOF

(Under construction.)

Synopsis

```
function EOF ([var F: any_file]): Boolean;
```

or

```
function EOF: Boolean;
```

Description

Conforming to

'EOF' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

EOLn

(Under construction.)

Synopsis

```
function EOLn ([var F: Text]): Boolean;
```

or

```
function EOLn: Boolean;
```

Description

Conforming to

'EOLn' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

EpsReal

(Under construction.)

Synopsis

Description

Conforming to

'EpsReal' is an ISO 10206 Extended Pascal extension.

Example

See also

EQ

(Under construction.)

Synopsis

```
function EQ (S1, S2: String): Boolean;
```

Description

Conforming to

'EQ' is an ISO 10206 Extended Pascal extension.

Example

See also

EQPad

(Under construction.)

Synopsis

```
function EQPad (S1, S2: String): Boolean;
```

Description

Conforming to

'EQPad' is a GNU Pascal extension.

Example

See also

Erase

(Under construction.)

Synopsis

```
procedure Erase (var F: any_file);
```

Description

Conforming to

'Erase' is a Borland Pascal extension.

Example

See also

Exclude

Synopsis

```
Exclude (set_variable, ordinal_value);
```

Description

Remove (subtract) a single element from a set. *ordinal_value* must be compatible with the base type of *set_variable*. Exclude is equivalent to:

```
set_variable := set_variable - [ordinal_value];
```

If *set_variable* does not contain *ordinal_value*, nothing happens.

Conforming to

'Exclude' is a Borland Pascal extension.

Example

```
program ExcludeDemo;

var
  Ch: Char;
  MyCharSet: set of Char;
```

```
begin
  MyCharSet := ['P', 'N', 'L'];
  Exclude (MyCharSet , 'N')  { L, P }
end.
```

See other examples in [set], page 412 and Section 6.10.7 [Set Operations], page 98.

See also

Chapter 9 [Keywords], page 453, Section 6.10.7 [Set Operations], page 98, [set], page 412, [in], page 337, [Include], page 339.

Exit

Synopsis

```
procedure Exit;
```

or

```
procedure Exit (program);
```

or

```
procedure Exit (Identifier);
```

Description

'Exit' without an argument leaves the currently executed procedure or function. Note: If 'Exit' is called within the main program, it will be terminated instantly.

'Exit' with an argument that is 'program' or the name of the current program, terminates the program, and is equivalent to 'Halt'.

'Exit' with an argument that is the name of the current or an encompassing routine leaves that routine.

Conforming to

'Exit' is a UCSD Pascal extension. Borland Pascal only allows it without an argument.

Example

```
program ExitDemo;

procedure Foo (Bar: Integer);
var
  Baz, Fac: Integer;
begin
  if Bar < 1 then
    Exit;  { Exit `Foo' }
  Fac := 1;
  for Baz := 1 to Bar do
    begin
      Fac := Fac * Baz;
      if Fac >= Bar then
        Exit;  { Exit `Foo' }
```

```
      WriteLn (Bar,' is greater than ', Baz, '!, which is equal to ', Fac)
  end
end;

begin
  Foo (-1);
  Foo (789);
  Exit;            { Terminates program }
  Foo (987654321)  { This is not executed anymore }
end.
```

See also

[Break], page 281, [Continue], page 298, [Halt], page 333.

Exp

Synopsis

```
function Exp (x: Real): Real;
```

or

```
function Exp (z: Complex): Complex;
```

Description

The exponential function 'Exp' computes the value of e to the power of x, where the Euler number e = Exp (1) is the base of the natural logarithm.

Conforming to

The function 'Exp' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program ExpDemo;
var
  z: Complex;
begin
  z := Cmplx (1, - 2 * Pi);  { z = 1 - 2 pi i }
  z := Exp (z);  { yields e = Exp (1), since Exp ix = Cos x + i Sin x }
  WriteLn (Ln (Re (z)) : 0 : 5)  { prints 1 = Ln (Exp (1)) }
end.
```

See also

[Ln], page 349

export

(Under construction.)

Synopsis

```
export ‘interface_name’ = (identifier, identifier, ...);
```

or

```
export ‘interface_name’ = all;
```

Description

Interface export for Extended Pascal modules.

‘all’ means to automatically export all identifiers declared in the interface module.

Conforming to

‘export’ is an ISO 10206 Extended Pascal extension. It also exists in Borland Pascal, but with a different meaning, not (yet) supported by GPC.

‘export all’ is a GNU Pascal extension.

Example

```
program ExportDemo;

import AllInterface in 'somemodule.pas';

begin
  Bar (a);
  WriteLn (b)
end.
```

```
module SomeModule interface;

export
  SomeInterface = (a);
  AllInterface = all;  { Same as ‘AllInterface = (a, b, Bar);’ }

var
  a, b: Integer;

procedure Bar (i: Integer);

end.

module SomeModule implementation;

procedure Bar (i: Integer);
begin
  b := a
end;
```

```
to begin do
  a := 42;

end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.8.1 [Modules], page 60.

exports

Not yet implemented.

Synopsis

Description

Library export.

Conforming to

'`exports`' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Extend

(Under construction.)

Synopsis

```
procedure Extend (var F: any_file; [FileName: String;]
                                    [BlockSize: Cardinal]);
```

Description

'`Extend`' opens a file for writing. If the file does not exist, it is created. If it does exist, the file pointer is positioned after the last element.

Like '`Rewrite`', '`Reset`' and '`Append`' do, '`Reset`' accepts an optional second parameter for the name of the file in the filesystem and a third parameter for the block size of the file. The third parameter is allowed only (and by default also required) for untyped files. For details, see [Rewrite], page 405.

Conforming to

'`Extend`' is an ISO 10206 Extended extension. Borland Pascal has [Append], page 266 instead. The '`BlockSize`' parameter is a Borland Pascal extension. The '`FileName`' parameter is a GNU Pascal extension.

Example

```
program ExtendDemo;
var
  Sample: Text;
begin
  Assign (Sample, 'sample.txt');
  Rewrite (Sample);
  WriteLn (Sample, 'Hello, World!');  { `sample.txt' now has one line }
  Close (Sample);

  { ... }

  Extend (Sample);
  WriteLn (Sample, 'Hello again!');  { `sample.txt' now has two lines }
  Close (Sample)
end.
```

See also

[Assign], page 272, [Reset], page 402, [Rewrite], page 405, [Update], page 441, [Append], page 266.

Extended

(Under construction.)

Synopsis

```
type
  Extended = LongReal;
```

Description

Conforming to

'Extended' is a Borland Pascal extension.

Example

```
program ExtendedDemo;
var
  a: Extended;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

external

(Under construction.)

Synopsis

declaration `external;`

or

declaration `external name` *linker_name*`;`

Description

Declaration of external object.

Conforming to

'`external`' is a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Fail

(Under construction.)

Synopsis

Description

Conforming to

'`Fail`' is a Borland Pascal extension.

Example

See also

False

Synopsis

```
type
  Boolean = (False, True);  { built-in type }
```

Description

'False' is one of the two Boolean values and is used to represent a condition which is never fullfilled. For example, the expression, '1 = 2' always yields 'False'. It is the opposite of 'True'. 'False' has the ordinal value 0.

Conforming to

'False' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program FalseDemo;

var
  a: Boolean;

begin
  a := 1 = 2;  { yields False }
  WriteLn (Ord (False));  { 0 }
  WriteLn (a);  { False }
  if False then WriteLn ('This is not executed.')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [True], page 434, [Boolean], page 280.

far

Synopsis

Description

The 'far' directive can be appended to a procedure or function heading but is ignored by GPC. It is there for Borland compatibility, only. (Since the GNU compilers provide a flat memory model, the distinction between 'near' and 'far' pointers is void.)

Conforming to

'far' is a Borland Pascal extension.

Example

```
program FarDemo;

var
  p: procedure;

{$W no-near-far}  { Don't warn about the uselessness of 'far' }
```

```
procedure Foo; far;  { ‘far’ has no effect in GPC }
begin
  WriteLn ('Foo')
end;

begin
  p := Foo;  { Would also work without ‘far’ in GPC. }
  p
end.
```

See also

Chapter 9 [Keywords], page 453, [near], page 368.

file

(Under construction.)

Synopsis

In type definitions:

`file of` *Type*

or

`file`

Description

Non-text file type declaration.

Conforming to

Typed files (‘`file of` *Type*’) are defined in ISO 7185 Pascal and supported by all known Pascal variants. Untyped files (‘`file`’) are a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453, [Text], page 429, [AnyFile], page 265.

FilePos

(Under construction.)

Synopsis

`function FilePos (var F:` *any_file*`): Integer;`

Description

Conforming to

'`FilePos`' is a Borland Pascal extension.

Example

See also

FileSize

(Under construction.)

Synopsis

```
function FileSize (var F: any_file): Integer;
```

Description

Conforming to

'`FileSize`' is a Borland Pascal extension.

Example

See also

FillChar

(Under construction.)

Synopsis

```
procedure FillChar (var Dest; Count: SizeType; Val: Char);
```

or

```
procedure FillChar (var Dest; Count: SizeType; Val: Byte);
```

Description

Conforming to

'`FillChar`' is a UCSD Pascal extension.

Example

See also

finalization

(Under construction.)

Synopsis

Description

Unit finalization.

It is equivalent to Extended Pascal's '`to end do`'.

Conforming to

'`finalization`' is a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [initialization], page 340, [to end do], page 433.

Finalize

(Under construction.)

Synopsis

```
procedure Finalize (var Aynthing);
```

Description

'`Finalize`' does all necessary clean-ups for the parameter. This is normally done automatically when a variable goes out of scope, so you need to call '`Finalize`' only in special situations, e.g. when you deallocate a dynamic variable with '`FreeMem`' rather than '`Dispose`'.

Using a variable after applying '`Finalize`' to it (and without applying '`Initialize`' to it again) is invalid and produces undefined results.

Conforming to

'`Finalize`' is a Borland Delphi extension.

Example

See also

[Initialize], page 341, [Dispose], page 307, [FreeMem], page 328.

Flush

(Under construction.)

Synopsis

```
procedure Flush (var F: any_file);
```

Description

Conforming to

'`Flush`' is a Borland Pascal extension.

Example

See also

for

Synopsis

For ordinal index variables:

`for` *ordinal_variable* `:=` *initial_value* `to` *final_value* `do`
  statement

or

`for` *ordinal_variable* `:=` *initial_value* `downto` *final_value* `do`
  statement

For sets:

`for` *set_element_type_variable* `in` *some_set* `do`
  statement

For pointer index variables:

`for` *pointer_variable* `:=` *initial_address* `to` *final_address* `do`
  statement

or

`for` *pointer_variable* `:=` *initial_address* `downto` *final_address* `do`
  statement

`@@` Set member iteration

Description

The '`for`' statement is a count loop. For further information see Section 6.1.7.5 [for Statement], page 57.

Conforming to

'`for`' is defined in ISO 7185 Pascal and supported by all known Pascal variants. Iteration of Pointers is a Borland Pascal extension. Set member iteration is an ISO 10206 Extended Pascal extension.

Example

```
program ForDemo;
var
  CharSet: set of Char;
  c: Char;
  n: Integer;
  Fac: array [0 .. 10] of Integer;
  PInt: ^Integer;
begin
  CharSet := ['g', 'p', 'c'];
  for c in CharSet do
    WriteLn (c);  { prints 'c', 'g', 'p' in three lines }
  Fac[0] := 1;
  for n := 1 to 10 do  { computes the factorial of n for n = 0 .. 10 }
    Fac[n] := Fac[n - 1] * n;
  {$X+}
  { prints n! for n = 0 .. 10 }
  for PInt := @Fac[0] to @Fac[10] do
    WriteLn (PInt - @Fac[0], '! = ', PInt^)
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.2.11.6 [Set Types], page 76, Section 6.6 [Pointer Arithmetics], page 84

FormatString

(Under construction.)

Synopsis

Description

Conforming to

'`FormatString`' is a GNU Pascal extension.

Example

See also

forward

(Under construction.)

Synopsis

Description

Declaration of a routine whose definition follows below.

Conforming to

'`forward`' is a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Frac

Synopsis

```
function Frac (x: Real): Real;
```

Description

'`Frac`' returns the fractional part of a floating point number.

Conforming to

'`Frac`' is a Borland Pascal extension.

Example

```
program FracDemo;

begin
  WriteLn (Frac (12.345) : 1 : 5);  { 0.34500 }
  WriteLn (Int (12.345) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.345) : 1);  { 12 }
  WriteLn (Trunc (12.345) : 1);  { 12 }

  WriteLn (Frac (-12.345) : 1 : 5);  { -0.34500 }
  WriteLn (Int (-12.345) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.345) : 1);  { -12 }
  WriteLn (Trunc (-12.345) : 1);  { -12 }

  WriteLn (Frac (12.543) : 1 : 5);  { 0.54300 }
  WriteLn (Int (12.543) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.543) : 1);  { 13 }
  WriteLn (Trunc (12.543) : 1);  { 12 }

  WriteLn (Frac (-12.543) : 1 : 5);  { -0.54300 }
  WriteLn (Int (-12.543) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.543) : 1);  { -13 }
  WriteLn (Trunc (-12.543) : 1);  { -12 }
end.
```

See also

Section 6.2.4 [Real Types], page 68, [Real], page 398, [Int], page 342, [Round], page 407, [Trunc], page 435.

FrameAddress

(Under construction.)

Synopsis

Description

Conforming to

'`FrameAddress`' is a GNU Pascal extension.

Example

See also

FreeMem

Synopsis

```
procedure FreeMem (var p: Pointer; Size: Cardinal);
```

or

```
procedure FreeMem (var p: Pointer);
```

Description

Releases a chunk of memory previously allocated using '`GetMem`'. The parameter *Size* is optional. Its value is currently ignored.

Since Extended Pascal's schemata provide a cleaner way to implement dynamical arrays and such, we recommend using '`GetMem`' and '`FreeMem`' only for low-level applications or for interfacing with other languages.

Conforming to

'`FreeMem`' is a Borland Pascal extension. '`FreeMem`' with only one parameter is a GNU Pascal extension.

Example

See [GetMem], page 330.

See also

[GetMem], page 330, Section 6.2.11.5 [Schema Types], page 72, [Dispose], page 307, [Mark], page 358, [Release], page 401.

function

(Under construction.)

Synopsis

Description

Function declaration.

Conforming to

'`function`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

GE

(Under construction.)

Synopsis

```
function GE (S1, S2: String): Boolean;
```

Description

Conforming to

'`GE`' is an ISO 10206 Extended Pascal extension.

Example

See also

GEPad

(Under construction.)

Synopsis

```
function GEPad (S1, S2: String): Boolean;
```

Description

Conforming to

'GEPad' is a GNU Pascal extension.

Example

See also

Get

(Under construction.)

Synopsis

```
procedure Get (var F: typed_file);
```

Description

Conforming to

'Get' is defined in ISO 7185 Pascal and supported by all known Pascal variants except UCSD/Borland Pascal and its variants.

Example

See also

GetMem

Synopsis

```
procedure GetMem (var p: Pointeger; Size: Cardinal);
```

Description

Allocates dynamical storage on the heap and returns a pointer to it in 'p'.

Since Extended Pascal's schemata provide a cleaner way to implement dynamical arrays and such, we recommend using 'GetMem' and 'FreeMem' only for low-level applications.

Conforming to

'GetMem' is a Borland Pascal extension.

Example

The Borland-comatibility unit 'Graph' from the 'BPcompat' package supports a 'GetImage' and a 'PutImage' procedure which need a variable of size 'ImageSize' as a buffer. Since these are "black box" routines, the buffer can't reasonably be a schema providing a dynamical array. Instead, we have to use 'GetMem' and 'FreeMem' for dynamical memory allocation.

```
program GetMemDemo;
var
  Buffer: Pointer;
  Size: Cardinal;
begin
  Size := Random (10000);  { the size can be determined at run time }
  GetMem (Buffer, Size);
  { Do something with Buffer }
  FreeMem (Buffer)  { or: FreeMem (Buffer, Size) }
end.
```

See also

[FreeMem], page 328, [New], page 369, Section 6.2.11.5 [Schema Types], page 72.

GetTimeStamp

Synopsis

```
procedure GetTimeStamp (var T: TimeStamp);
```

Description

GetTimeStamp gets the current local date and time as a `TimeStamp` record containing the Year, Month, Day, Hour, Minute, Second, and so on.

Conforming to

'GetTimeStamp' is an ISO 10206 Extended Pascal extension.

Example

Set [TimeStamp], page 431.

See also

[TimeStamp], page 431, [Date], page 303, [Time], page 430, Section 6.10.8 [Date And Time Routines], page 99.

goto

(Under construction.)

Synopsis

`goto` *label*

Description

The 'goto' statement transfers control to statement with the label 'label'.

Conforming to

'goto' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

GT

(Under construction.)

Synopsis

```
function GT (S1, S2: String): Boolean;
```

Description

Conforming to

'GT' is an ISO 10206 Extended Pascal extension.

Example

See also

GTPad

(Under construction.)

Synopsis

```
function GTPad (S1, S2: String): Boolean;
```

Description

Conforming to

'GTPad' is a GNU Pascal extension.

Example

See also

Halt

Synopsis

```
Halt;
```

or

```
Halt (ExitCode: Integer);
```

Description

'`Halt`' terminates the program with exitcode 0. If '`ExitCode`', is specified, it is returned by '`Halt`' on exit.

Conforming to

'`Halt`' is an Extended Pascal and a UCSD Pascal extension.

Example

```
program HaltDemo;
begin
  WriteLn ('This string will always be this program''s output.');
  Halt;  { Terminate right here and right now. }
  WriteLn ('And this string won''t ever!')
end.
```

See also

[Break], page 281, [Continue], page 298, [Exit], page 315, [Return], page 404, [goto], page 331.

High

Synopsis

`function High (`*ordinal_type_or_variable*`):` *ordinal_type*`;`

or

`function High (`*array_type_or_variable*`):` *array_index_type*`;`

or

`function High (`*string_variable*`): Integer;`

Description

For ordinal types or variables of that type, '`High`' returns the highest value a variable of that type can assume.

For array types or variables of that type, '`High`' returns the highest index a variable of that type can assume. Note: the result is of the same type as the array index is. If the array has more than one dimension, '`High`' returns the highest index in the first dimension.

If the argument is a string variable, '`High`' returns the discriminant of the string type (i.e. its capacity).

Conforming to

'High' is a Borland Pascal extension.

Example

```
program HighDemo;
type
  Colors = (Red, Green, Blue);
var
  Col: array [Colors] of (Love, Hope, Faithfulness);
  Foo: Colors;
  Bar: Integer;
  Baz: String (123);
begin
  Foo := High (Col);              { yields Blue }
  Bar := Ord (High (Col[Foo]));  { yields Ord (Faithfulness), i.e., 2 }
  Bar := High (Integer);          { highest possible value of 'Integer' }
  Bar := High (Baz)               { returns 123 }
end.
```

See also

[Low], page 356

if

Synopsis

if *Boolean_expression* then
  statement

or with an alternative statement:

if *Boolean_expression* then
  statement1
else
  statement2

Description

The 'if ... then' statement executes *statement1* depending on 'Boolean expression' being true. If 'else' is specified, it continues executing *statement2* instead.

Conforming to

'if' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program IfDemo;
var
  Foo, Bar: Boolean;
```

```
begin
  Foo := True;
  Bar := False;
  if ((1 = 1) or (2 = 3)) and (Foo = not Bar) then
    begin
      { This is executed if either Foo is true but not Bar or vice versa }
      WriteLn ('Either Foo or Bar is true.');
      if Bar then
        WriteLn ('You will see this text if Bar is true.')
    end
  else  { This whole ‘else’ branch is not executed }
    if 1 = 1 then
      if True = False then
        WriteLn ('This text is never written on screen.')
      else  { Note: This ‘‘else’’ belongs to ‘‘if True = False’’ }
        WriteLn ('This text is never written on screen as well.')
    else  { Note: This ‘‘else’’ belongs to ‘‘if 1 = 1’’ }
      WriteLn ('Nor is this.')
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.3 [if Statement], page 56, [else], page 310, [then], page 430

Im

Synopsis

```
function Im (z: Complex): Real;
```

Description

‘Im’ extracts the imaginary part of the complex number ‘z’. The result is a real value.

Conforming to

‘Im’ is an ISO 10206 Extended Pascal extension.

Example

```
program ImDemo;
var
  z: Complex;
begin
  z := Cmplx (1, 2);  { 1 + i * 2 }
  WriteLn (Im (z) : 0 : 5)  { 2.00000 }
end.
```

See also

[Cmplx], page 292, [Re], page 397, [Arg], page 269.

implementation

(Under construction.)

Synopsis

Description

Module or unit implementation part.

Conforming to

'`implementation`' is an Extended Pascal and a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

import

Synopsis

```
program @@fragment foo;

import
  bar1;
  bar3 (baz1 => glork1) in 'baz.pas';
  bar2 only (baz2, baz3 => glork2);
  bar4 qualified;

[...]
```

In module:

```
module @@fragment Bar;

import
  bar1;
  bar3 (baz1 => glork1) in 'baz.pas';
  bar2 only (baz2, baz3 => glork2);
  bar4 qualified;

[...]
```

Description

The reserved word '`import`' in the *import part* of a program makes the program import an interface.

The '`in`' above tells GPC to look for the module in the specified file; otherwise the file name is derived from the name of the interface by adding first '`.p`', then '`.pas`' – which only works if the name of the exported interface coincides with the file name.

The symbol '`=>`' denotes import renaming: The entity which is exported under the name '`baz1`' by the interface '`bar3`' will be known under the new name '`glork1`' in the program.

The '`only`' qualifier means that only the listed identifiers will be imported from the interface. Renaming works together with '`only`', too.

The '`qualified`' qualifier means "qualified import": The imported identifiers will be accessible only trough a "qualified name". The entity which is exported under the name '`baz2`' by the interface '`bar4`' will be accessible as '`bar4.baz2`'.

There must be at most one import part at the begining of a block. Each module or function/procedure (and also main program) can have its own import part. A single import part can import arbitrarly many interfaces.

The interfaces needn't be exported by Extended Pascal modules but may be UCSD/Borland Pascal units as well. GPC also allows '`import`' parts in units.

Conforming to

'`import`' and modules in general are an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453, [module], page 365, [unit], page 439, [uses], page 441.

in

Synopsis

As part of the `set` membership test, as a boolean expression:

ordinal_value `in` *set_expression*

or, as part of a '`for`' loop iterating through a set:

`for` *ordinal_variable* `in` *set_expression* `do ...`

Description

When '`in`' is used as a membership test, it acts as a binary operator taking *ordinal_value* as its left parameter and *set_expression* as its right parameter and returning a boolean result which is true if *set_expression* contains the element *ordinal_value*.

When '`in`' is used as part of a '`for`' loop, it iterates *ordinal_variable* over the elements contained in *set_expression*, that is every *ordinal_value* that would return true if tested as *ordinal_value* `in` *set_expression*.

Conforming to

'`in`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

Set [set], page 412

See also

Chapter 9 [Keywords], page 453, Section 6.10.7 [Set Operations], page 98, [set], page 412, [Exclude], page 314, [Include], page 339, [for], page 325.

Inc

Synopsis

For ordinal types:

`procedure Inc (var x:` *ordinal_type*`);`

or

`procedure Inc (var x:` *ordinal_type*`; Amount:` *and_integer_type*`);`

For pointer types:

`procedure Inc (var p:` *any_pointer_type*`);`

or

`procedure Inc (var p:` *any_pointer_type*`; Amount:` *and_integer_type*`);`

Description

For ordinal types, '`inc`' increases the value of '`x`' by one or by '`amount`' if it is given.

If the argument '`p`' is pointing to a specified type (typed pointer), '`inc`' increases the address of '`p`' by the size of the type '`p`' is pointing to or by '`amount`' times that size respectively. If '`p`' is an untyped pointer (i.e. '`p`' is of type [Pointer], page 386), '`p`' is increased by one.

Conforming to

'`Inc`' is a Borland Pascal extension. Yet application of '`Inc`' to pointers is defined in Borland Pascal. The combination of the second argument with application to pointers is a GNU Pascal extension.

Example

```
program IncDemo;
var
  Foo: Integer;
  Bar: array [1 .. 5] of Integer;
  Baz: ^Integer;
begin
  Foo := 4;
  Inc (Foo, 5);      { yields 9 }
  {$X+}             { Turn on extended systax }
  Baz := @Bar[1];   { Baz points to y[1] }
  Inc (Baz, 2);      { Baz points to y[3] }
end.
```

See also

[Dec], page 304, [Pred], page 388, [Succ], page 428, Section 6.6 [Pointer Arithmetics], page 84.

Include

Synopsis

`Include (`*set_variable*`, `*ordinal_value*`);`

Description

Add (join) a single element to a set. *ordinal_value* must be compatible with the base type of *set_variable*. Include is equivalent to:

set_variable `:=` *set_variable* `+ [`*ordinal_value*`];`

If *set_variable* already contains *ordinal_value*, nothing happens.

Conforming to

'`Include`' is a Borland Pascal extension.

Example

```
program IncludeDemo;

var
  Ch: Char;
  MyCharSet: set of Char;

begin
  MyCharSet := ['P', 'N', 'L'];
  Include (MyCharSet , 'A')  { A, L, N, P }
end.
```

See other examples in [set], page 412 and Section 6.10.7 [Set Operations], page 98.

See also

Chapter 9 [Keywords], page 453, Section 6.10.7 [Set Operations], page 98, [set], page 412, [in], page 337, [Exclude], page 314.

Index

(Under construction.)

Synopsis

Description

Conforming to

'Index' is an ISO 10206 Extended Pascal extension.

Example

See also

inherited

(Under construction.)

Synopsis

Description

Reference to methods of ancestor object types.

Conforming to

'inherited' is an Object Pascal, Borland Pascal and traditional Macintosh Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

initialization

(Under construction.)

Synopsis

Description

Unit initialization.
It is equivalent to Extended Pascal's 'to begin do'.

Conforming to

'initialization' is a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [finalization], page 324, [to begin do], page 433.

Initialize

(Under construction.)

Synopsis

```
procedure Initialize (var Aynthing);
```

Description

'`Initialize`' does all necessary initializations for the parameter (e.g., setting of string and schema discriminants, and object VMT pointers, initialization of file variables). This is normally done automatically at the start of the lifetime of a variable, so you need to call '`Initialize`' only in special situations, e.g. when you allocate a dynamic variable with '`GetMem`' rather than '`New`'.

Conforming to

'`Initialize`' is a Borland Delphi extension.

Example

See also

[Finalize], page 324, [New], page 369, [GetMem], page 330.

InOutRes

(Under construction.)

Synopsis

```
var
  InOutRes: Integer;
```

Description

Conforming to

'`InOutRes`' is a UCSD Pascal extension.

Example

See also

Input

(Under construction.)

Synopsis

```
var
  Input: Text;
```

Description

Conforming to

'Input' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Insert

(Under construction.)

Synopsis

```
procedure Insert (Source: String; var Dest: String; Position: Integer);
```

Description

Conforming to

'Insert' is a UCSD Pascal extension.

Example

See also

Int

Synopsis

```
function Int (x: Real): Real;
```

Description

'Int' returns the integer part of a floating point number as a floating point number. Use 'Trunc' to get the integer part as an integer.

Conforming to

'Int' is a UCSD Pascal extension.

Example

```
program IntDemo;

begin
  WriteLn (Frac (12.345) : 1 : 5);  { 0.34500 }
  WriteLn (Int (12.345) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.345) : 1);  { 12 }
  WriteLn (Trunc (12.345) : 1);  { 12 }

  WriteLn (Frac (-12.345) : 1 : 5);  { -0.34500 }
  WriteLn (Int (-12.345) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.345) : 1);  { -12 }
  WriteLn (Trunc (-12.345) : 1);  { -12 }

  WriteLn (Frac (12.543) : 1 : 5);  { 0.54300 }
  WriteLn (Int (12.543) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.543) : 1);  { 13 }
  WriteLn (Trunc (12.543) : 1);  { 12 }

  WriteLn (Frac (-12.543) : 1 : 5);  { -0.54300 }
  WriteLn (Int (-12.543) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.543) : 1);  { -13 }
  WriteLn (Trunc (-12.543) : 1);  { -12 }
end.
```

See also

Section 6.2.4 [Real Types], page 68, [Real], page 398, [Frac], page 327, [Round], page 407, [Trunc], page 435.

Integer

Synopsis

```
type
  Integer  { built-in type }
```

Description

'Integer' is the "natural" signed integer type in GNU Pascal. On some platforms it is 32 bits wide and thus has a range of '-2147483648 .. 2147483647'. Use it whenever you need a general-purpose signed integer type.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

In ISO Pascal, 'Integer' is the only built-in integer type. (However see Section 6.2.11.1 [Subrange Types], page 70.)

Example

```
program IntegerDemo;
var
  a: Integer;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

interface

(Under construction.)

Synopsis

Description

Module or unit interface part.

Conforming to

'interface' is an Extended Pascal and a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

interrupt

Not yet implemented.

Synopsis

Description

Interrupt handler declaration (not yet implemented).

Conforming to

'interrupt' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

IOResult

(Under construction.)

Synopsis

```
function IOResult: Integer;
```

Description

Conforming to

'`IOResult`' is a UCSD Pascal extension.

Example

See also

is

Synopsis

Description

Object type membership test.

Conforming to

'`is`' is an Object Pascal and a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [as], page 271, [TypeOf], page 438, Section 6.8 [OOP], page 86.

label

(Under construction.)

Synopsis

Description

Label declaration for a '`goto`' statement.

Conforming to

'`label`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

LastPosition

(Under construction.)

Synopsis

```
function LastPosition (var F: typed_file): index_type;
```

Description

Conforming to

'`LastPosition`' is an ISO 10206 Extended Pascal extension.

Example

See also

LE

(Under construction.)

Synopsis

```
function LE (S1, S2: String): Boolean;
```

Description

Conforming to

'`LE`' is an ISO 10206 Extended Pascal extension.

Example

See also

Leave

Synopsis

```
Leave  { simple statement }
```

Description

With 'Leave' you can exit the body of the current loop instantly. It can only be used within a 'while', 'repeat' or a 'for' loop.

Conforming to

'Leave' is a Mac Pascal extension. Borland Pascal has 'Break' instead.

Example

```
program LeaveDemo;
var
  Foo: Integer;
begin
  while True do
    begin
      repeat
        WriteLn ('Enter a number less than 100:');
        ReadLn (Foo);
        if Foo < 100 then
          Leave;  { Exits `repeat' loop }
        WriteLn (Foo, ' is not exactly less than 100! Try again ...')
      until False;
      if Foo > 50 then
        Leave;  { Exits `while' loop }
      WriteLn ('The number entered was not greater than 50.')
    end
end.
```

See also

Section 6.1.7.13 [Loop Control Statements], page 60, [Break], page 281, [Continue], page 298, [Cycle], page 303, [Exit], page 315, [Halt], page 333, [Return], page 404, [goto], page 331.

Length

(Under construction.)

Synopsis

```
function Length (S: String): Integer;
```

Description

Conforming to

'`Length`' is an Extended Pascal and a UCSD Pascal extension.

Example

See also

LEPad

(Under construction.)

Synopsis

```
function LEPad (S1, S2: String): Boolean;
```

Description

Conforming to

'`LEPad`' is a GNU Pascal extension.

Example

See also

library

Not yet implemented.

Synopsis

Description

Library declaration.

Conforming to

'`library`' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Ln

Synopsis

```
function Ln (x: Real): Real;
```

or

```
function Ln (z: Complex): Complex;
```

Description

The natural logarith 'Ln' is the logarithm with base e, where e is the Euler number e = Exp (1) = 2.718281828...

Conforming to

The function 'Ln' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program LnDemo;
var
  z: Complex;
begin
  z := Cmplx (1, 1);
  z := Ln (z)  { yields Ln (SqRt (2)) + i * Pi / 4 }
               { since Ln (i * x) = Ln Abs (x) + i * Arg (x) }
end.
```

See also

LoCase

(Under construction.)

Synopsis

```
function LoCase (Ch: Char): Char;
```

Description

Conforming to

'LoCase' is a GNU Pascal extension.

Example

See also

LongBool

Synopsis

```
type
  LongBool = Boolean attribute (Size = BitSizeOf (LongInt));
```

Description

The intrinsic 'LongBool' represents boolean values, but occupies the same memory space as a 'LongInt'. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

'LongBool' is a Borland Pascal extension.

Example

```
program LongBoolDemo;
var
  a: LongBool;
begin
  LongInt (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [ShortBool], page 416, [MedBool], page 360, [WordBool], page 449, [LongestBool], page 351.

LongCard

Synopsis

```
type
  LongCard = Cardinal attribute (Size = BitSizeOf (LongInt));
```

Description

'LongCard' is an unsigned integer type which is longer than 'Cardinal'. On some platforms it is 64 bits wide and thus has a range of '0 .. 18446744073709551615'.

'LongCard' in GNU Pascal is compatible to 'long long unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongCard' is a GNU Pascal extension.

Example

```
program LongCardDemo;
var
  a: LongCard;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

LongestBool

Synopsis

```
type
  LongestBool = Boolean attribute (Size = BitSizeOf (LongestInt));
```

Description

The intrinsic '`LongestBool`' represents boolean values, but occupies the same memory space as a '`LongestInt`'. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

'`LongestBool`' is a GNU Pascal extension.

Example

```
program LongestBoolDemo;
var
  a: LongestBool;
begin
  LongestInt (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [ShortBool], page 416, [MedBool], page 360, [WordBool], page 449, [LongBool], page 350.

LongestCard

Synopsis

```
type
  LongestCard = Cardinal attribute (Size = BitSizeOf (LongestInt));
```

Description

'LongestCard' is GPC's longest-possible unsigned integer type. Currently, this is the same as [LongCard], page 350. On some platforms it is 64 bits wide and thus has a range of '0 .. 18446744073709551615'.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongestCard' is a GNU Pascal extension.

Example

```
program LongestCardDemo;
var
  a: LongestCard;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

LongestInt

Synopsis

```
type
  LongestInt = LongInt;  { might get bigger than LongInt someday }
```

Description

'LongestInt' is GPC's longest-possible signed integer type. Currently, this is the same as [LongInt], page 354. On some platforms it is 64 bits wide and thus has a range of '-9223372036854775808 .. 9223372036854775807'.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongestInt' is a GNU Pascal extension.

Example

```
program LongestIntDemo;
var
  a: LongestInt;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

LongestReal

(Under construction.)

Synopsis

```
type
  LongestReal = LongReal;  { might get bigger than LongReal someday }
```

Description

Conforming to

'LongestReal' is a GNU Pascal extension.

Example

```
program LongestRealDemo;
var
  a: LongestReal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

LongestWord

Synopsis

```
type
  LongestWord = LongestCard;
```

Description

'LongestWord' is GPC's longest-possible unsigned integer type. Currently, this is the same as [LongWord], page 355. On some platforms it is 64 bits wide and thus has a range of '0 .. 18446744073709551615'. (It is the same as [LongestCard], page 352.)

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongestWord' is a GNU Pascal extension.

Example

```
program LongestWordDemo;
var
  a: LongestWord;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[LongestCard], page 352, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

LongInt

Synopsis

```
type
  LongInt  { built-in type }
```

Description

'LongInt' is a signed integer type which is longer than 'Integer'. On some platforms it is 64 bits wide and thus has a range of '-9223372036854775808 .. 9223372036854775807'.

'LongInt' in GNU Pascal is compatible to 'long long int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongInt' is a Borland Pascal extension. Borland Pascal defines 'LongInt' as a 32-bit signed integer type ([Integer], page 343 in GNU Pascal).

Example

```
program LongIntDemo;
var
  a: LongInt;
begin
```

```
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

LongReal

(Under construction.)

Synopsis

```
type
  LongReal  { built-in type }
```

Description

Conforming to

'LongReal' is a GNU Pascal extension.

Example

```
program LongRealDemo;
var
  a: LongReal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

LongWord

Synopsis

```
type
  LongWord = LongCard;
```

Description

'LongWord' is an unsigned integer type which is larger than 'Word'. On some platforms it is 64 bits wide and thus has a range of '0 .. 18446744073709551615'. It is the same as [LongCard], page 350.

'LongWord' in GNU Pascal is compatible to 'long long unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'LongWord' is a GNU Pascal extension.

Example

```
program LongWordDemo;
var
  a: LongWord;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[LongCard], page 350, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

Low

Synopsis

`function Low (`*ordinal_type_or_variable*`):` *ordinal_type*`;`

or

`function Low (`*array_type_or_variable*`):` *array_element_type*`;`

or

`function Low (`*string_variable*`): Integer;`

Description

For ordinal types or variables of that type, 'Low' returns the lowest value a variable of that type can assume.

For array types or variables of that type, 'Low' returns the lowest index a variable of that type can assume. Note: the result is of the same type as the array index is. If the array has more than one dimension, 'Low' returns the lowest index in the first dimension.

If the argument is a string variable, 'Low' returns one.

Conforming to

'Low' is a Borland Pascal extension.

Example

```
program LowDemo;
type
  Colors = (Red, Green, Blue);
var
  Col: array [12 .. 20] of Colors;
  Foo: 12 .. 20;
```

```
  Bar: Integer;
begin
  Foo := Low (Col);            { returns 12 }
  Col[Foo] := Low (Col[Foo]);  { returns Red }
  Bar := Low (Integer)         { lowest possible value of ‘Integer’ }
end.
```

See also

[High], page 333

LT

(Under construction.)

Synopsis

```
function LT (S1, S2: String): Boolean;
```

Description

Conforming to

‘LT’ is an ISO 10206 Extended Pascal extension.

Example

See also

LTPad

(Under construction.)

Synopsis

```
function LTPad (S1, S2: String): Boolean;
```

Description

Conforming to

‘LTPad’ is a GNU Pascal extension.

Example

See also

Mark

(Under construction.)

Synopsis

```
procedure Mark (var P: Pointer);
```

Description

Conforming to

'Mark' is a UCSD Pascal extension.

Example

See also

Max

(Under construction.)

Synopsis

function Max (x1, x2: *ordinal_or_real_type*): *same_type*;

Description

Conforming to

'Max' is a GNU Pascal extension.

Example

See also

MaxChar

(Under construction.)

Synopsis

Description

The value of `MaxChar` is the largest value of `Char`.

Conforming to

'`MaxChar`' is an ISO 10206 Extended Pascal extension.

Example

See also

MaxInt

(Under construction.)

Synopsis

Description

The `MaxInt` constant defines the maximum value of `Integer`. This constant is a built-in compiler value.

Conforming to

'`MaxInt`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

MaxReal

(Under construction.)

Synopsis

Description

Conforming to

'`MaxReal`' is an ISO 10206 Extended Pascal extension.

Example

See also

[MinReal], page 363.

MedBool

Synopsis

```
type
  MedBool = Boolean attribute (Size = BitSizeOf (MedInt));
```

Description

The intrinsic 'MedBool' represents boolean values, but occupies the same memory space as a 'MedInt'. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

'MedBool' is a GNU Pascal extension.

Example

```
program MedBoolDemo;
var
  a: MedBool;
begin
  MedInt (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [ShortBool], page 416, [WordBool], page 449, [LongBool], page 350, [LongestBool], page 351.

MedCard

Synopsis

```
type
  MedCard = Cardinal attribute (Size = BitSizeOf (MedInt));
```

Description

'MedCard' is an unsigned integer type which is not smaller than 'Cardinal'. On some platforms it actually is the same as 'Cardinal' and 32 bits wide and thus has a range of '0 .. 4294967295'.

'MedCard' in GNU Pascal is compatible to 'long unsigned int' in GNU C. This compatibility is the reason why 'MedCard' exists.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'MedCard' is a GNU Pascal extension.

Example

```
program MedCardDemo;
var
  a: MedCard;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

MedInt

Synopsis

```
type
  MedInt  { built-in type }
```

Description

'MedInt' is a signed integer type which is not smaller than 'Integer'. On some platforms it actually is the same as 'Integer' and 32 bits wide and thus has a range of '-2147483648 .. 2147483647'.

'MedInt' in GNU Pascal is compatible to 'long int' in GNU C. This compatibility is the reason why 'MedInt' exists.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'MedInt' is a GNU Pascal extension.

Example

```
program MedIntDemo;
var
  a: MedInt;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

MedReal

(Under construction.)

Synopsis

```
type
  MedReal = Real;
```

Description

Conforming to

'MedReal' is a GNU Pascal extension.

Example

```
program MedRealDemo;
var
  a: MedReal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

MedWord

Synopsis

```
type
  MedWord = MedCard;
```

Description

'MedWord' is an unsigned integer type which is not smaller than 'Word'. On some platforms it actually is the same as 'Word' and 32 bits wide and thus has a range of '0 .. 4294967295'. It is the same as [MedCard], page 360.

'MedWord' in GNU Pascal is compatible to 'long unsigned int' in GNU C. This compatibility is the reason why 'MedWord' exists.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'MedWord' is a GNU Pascal extension.

Example

```
program MedWordDemo;
var
  a: MedWord;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[MedCard], page 360, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

Min

(Under construction.)

Synopsis

`function Min (x1, x2:` *ordinal_or_real_type*`):` *same_type*`;`

Description

Conforming to

'`Min`' is a GNU Pascal extension.

Example

See also

MinReal

(Under construction.)

Synopsis

Description

Conforming to

'`MinReal`' is an ISO 10206 Extended Pascal extension.

Example

See also

[MaxReal], page 359.

MkDir

Synopsis

```
procedure MkDir (Directory: String);
```

Description

'MkDir' creates the given *Directory*, if its argument is a valid parameter to the related operating system's function. Otherwise a runtime error is caused.

Conforming to

'MkDir' is a Borland Pascal extension.

Example

```
program MkDirDemo;
var
  Foo: String (127);
begin
  WriteLn ('Enter directory name to create:');
  ReadLn (Foo);
  {$I-}  { Don't abort program on error }
  MkDir (Foo);
  if IOResult <> 0 then
    WriteLn ('Directory `', Foo, ''' could not be created')
  else
    WriteLn ('Okay')
end.
```

See also

[ChDir], page 289, [RmDir], page 406

mod

(Under construction.)

Synopsis

```
operator mod (p, q: Integer) = r: Integer;
```

Description

Integer remainder operator.

Conforming to

'mod' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

module

(Under construction.)

Synopsis

Description

EP style or PXSC style module.

Conforming to

'`module`' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Move

(Under construction.)

Synopsis

```
procedure Move (const Source; var Dest; Count: Integer);
```

Description

Conforming to

'`Move`' is a Borland Pascal extension.

Example

See also

MoveLeft

(Under construction.)

Synopsis

```
procedure MoveLeft (const Source; var Dest; Count: Integer);
```

Description

Conforming to

'MoveLeft' is a UCSD Pascal extension.

Example

See also

MoveRight

(Under construction.)

Synopsis

```
procedure MoveRight (const Source; var Dest; count: Integer);
```

Description

Conforming to

'MoveRight' is a UCSD Pascal extension.

Example

See also

name

(Under construction.)

Synopsis

procedure/function_header; external name *name*;
procedure/function_header; attribute (name = *name*);
or
variable_declaration; external name *name*;
variable_declaration; attribute (name = *name*);
or
unit *Name*; attribute (name = *name*);

Description

The 'name' directive declares the external name of a procedure, function or variable. It can be used after 'external' or within 'attribute'.

This directive declares the external name of a procedure, function or variable. The external name of the routine is given explicitly as a case-sensitive constant string expression. This is useful when interfacing with libraries written in other languages.

With this extension it is possible to access all external functions, for example the X11 interface functions, and not only those written in lowercase.

'name' can also be applied to units and module interfaces. In this case it denotes the *prefix* prepended to the external name of the initializer of the unit: While it is normally called 'init_Modulename', it is called 'init_*name*_Modulename' when 'name' is given.

This is not of interest under normal circumstances since the initializers are called automatically. It can help avoiding conflicts when there are several units of the same name within one program. Again, this does not happen normally, but e.g., when a program uses a unit/module that has the same name as one of the units the RTS consists of: The RTS uses 'GPC' as the name for its units to avoid conflicts.

In the future, a 'name' directive applied to units, modules and programs (the latter is recognized syntactically already, but has no effect yet) will also affect the default external name of routines and variables which have no 'name' directive themselves. Again, this is mostly useful for libraries etc., and will not be necessary for normal units, modules and programs.

Conforming to

'name' is a Borland Pascal extension. 'attribute' and the application of 'name' to units, modules and programs are GNU Pascal extensions.

Example

```
program NameDemo;

{ Make two variables aliases of each other by using ‘name’.
  This is not good style. If you must have aliases for any reason,
  ‘absolute’ declarations may be the lesser evil ... }
var
  Foo: Integer; attribute (name = 'Foo_Bar');
  Bar: Integer; external name 'Foo_Bar';

{ A function from the C library }
function PutS (Str: CString): Integer; external name 'puts';

var
  Result: Integer;
begin
  Result := PutS ('Hello World!');
  WriteLn ('puts wrote ', Result, ' characters (including a newline).');
  Foo := 42;
  WriteLn ('Foo = ', Foo);
  Bar := 17;
  WriteLn ('Setting Bar to 17.');
  WriteLn ('Now, Foo = ', Foo, '!!!')
end.
```

See also

Chapter 9 [Keywords], page 453, [attribute], page 274, [external], page 320, Section 6.11.1 [Importing Libraries from Other Languages], page 100.

NE

(Under construction.)

Synopsis

```
function NE (S1, S2: String): Boolean;
```

Description

Conforming to

'NE' is an ISO 10206 Extended Pascal extension.

Example

See also

near

Synopsis

Description

The 'near' directive can be appended to a procedure or function heading but is ignored by GPC. It is there for Borland compatibility, only. (Since the GNU compilers provide a flat memory model, the distinction between 'near' and 'far' pointers is void.)

Conforming to

'near' is a Borland Pascal extension.

Example

```
program NearDemo;

var
  p: procedure;

{$W no-near-far}  { Don't warn about the uselessness of 'near' }

procedure Foo; near;  { 'near' has no effect in GPC }
begin
```

```
  WriteLn ('Foo')
end;

begin
  p := Foo;  { Works, despite the 'near'. }
  p
end.
```

See also

Chapter 9 [Keywords], page 453, [far], page 321.

NEPad

(Under construction.)

Synopsis

```
function NEPad (S1, S2: String): Boolean;
```

Description

Conforming to

'`NEPad`' is a GNU Pascal extension.

Example

See also

New

(Under construction.)

Synopsis

`procedure New (var P:` *any_Pointer*`);`

or

`procedure New (var P:` *Pointer_to_a_variant_record*`;` *tag_fields*`);`

or

`procedure New (var P:` *Pointer_to_a_schema*`;` *discriminants*`);`

or

`procedure New (var P:` *Pointer_to_an_object*`;` *constructor_call*`);`

or

`function New (`*any_Pointer_type*`):` *same_type*`;`

or

```
function New (variant_record_Pointer_type;
              tag_fields): same_type;
```

or

```
function New (schema_Pointer_type;
              discriminants): same_type;
```

or

```
function New (object_Pointer_type;
              constructor_call): same_type;
```

Description

Conforming to

'New' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

NewCString

(Under construction.)

Synopsis

```
function NewCString (const S: String): CString;
```

Description

Conforming to

'NewCString' is a GNU Pascal extension.

Example

See also

nil

Synopsis

'nil' is a predefined constant

Description

'nil' is a predefined pointer constant that indicates an unassigned pointer. "nil" stands for "not in list". *Every* pointer type can be associated with this constant.

Conforming to

'nil' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program NilDemo;
const
  NodeNum = 10;
type
  PNode = ^TNode;
  TNode = record
    Key: Integer;
    Next: PNode
  end;
var
  Root, Node: PNode;
  Foo: Integer;
begin
  New (Root);
  Root^.Key := 1;             { Set root key }
  Node := Root;
  for Foo := 2 to NodeNum do  { Create linked list with NODE_NUM nodes }
    begin
      New (Node^.Next);
      Node := Node^.Next;
      Node^.Key := Foo        { Set key }
    end;
  Node^.Next := nil;          { Mark end of linked list }
  { Shorten list by removing its first element until list is empty }
  while Root <> nil do
    begin
      Node := Root;
      WriteLn ('Current key:', Root^.Key);
      Root := Root^.Next;
      Dispose (Node);
      Node := nil             { Indicate old node is not assigned }
    end
end.
```

See also

Chapter 9 [Keywords], page 453, [Assigned], page 273, [Pointer], page 386

not

(Under construction.)

Synopsis

```
operator not (b1, b2: Boolean) = Result: Boolean;
```

or

```
operator not (i1, i2: integer_type) = Result: integer_type;
```

Description

Boolean or bitwise negation operator.

Conforming to

'not' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

Null

Synopsis

```
var
  Null: Void absolute 0;
```

Description

'Null' is a predefined variable at address 'nil'. 'Null' can be passed as a "void" argument to a procedure, function or operator expecting a "var" parameter. *Note*: Make sure they can handle this case, otherwise this is likely to cause an exception and the program will be terminated. Since 'Null' is an L-value, it can be taken as "nil-reference".

Conforming to

'Null' is a Borland Delphi extension.

Example

```
program NullDemo;
type
  PString = ^String;
var
  Com1: String (25) = 'This is an amazing number';
  Com2: String (25) = 'This is a boring number';

procedure FooBar (Foo: Integer; var Comment: PString);
begin
  if Odd (Foo) then
    WriteLn ('FooBar:', Foo, ' is odd')
   else
    WriteLn ('FooBar:', Foo, ' is even');
  if @Comment <> nil then
    if not Odd (Foo) then
      Comment := @Com1
    else
```

```
      Comment := @Com2
  end;

  var
    S: String (25);
    P: PString value @S;

  begin
    { FooBar allows you to leave out variables
      for any information you might not need }
    FooBar (1, Null);
    { But FooBar is flexible, after all }
    FooBar (6, P);
    WriteLn ('FooBar said about 6: `', P^, '''')
  end.
```

See also

[nil], page 370

object

Synopsis

Description

The keyword ‘object’ is used to declare a new object type:

```
type
  foo = object
    a: Integer;
    constructor Init;
    procedure Bar (x: Integer); virtual;
  end;
```

(For a longer example, see Section 6.8 [OOP], page 86.)

Conforming to

GNU Pascal follows the Borland Pascal 7.0 object model.

ISO Pascal does not support Object-oriented programming. There is an ANSI draft for an “Object Pascal” language which is not yet supported by GPC, but planned. The Delphi language, also called “Object Pascal” by Borland, is currently not supported by GPC either.

Example

See also

Chapter 9 [Keywords], page 453, Section 6.8 [OOP], page 86, [record], page 399.

Odd

Synopsis

```
function Odd (i: Integer): Boolean;
```

Description

'Odd' checks the parity of its argument 'i'. It returns 'True' if the argument is odd, 'False' if it is even.

Conforming to

'Odd' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program OddDemo;
var
  Foo: Integer;
begin
  Write ('Please enter an odd number: ');
  ReadLn (Foo);
  if not Odd (Foo) then
    WriteLn ('Odd''s not even! Something''s odd out there ...')
  else
    WriteLn (Foo, ' is pretty odd.')
end.
```

See also

of

(Under construction.)

Synopsis

Description

Part of an 'array', 'set' or typed 'file' type declaration, a 'case' statement, a variant 'record' type or a 'type of' type inquiry.

Conforming to

'of' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

only

(Under construction.)

Synopsis

Description

Import specification.

Conforming to

'only' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

operator

(Under construction.)

Synopsis

Description

Operator declaration.

Conforming to

'operator' is PASCAL_SC extension.

Example

See also

Chapter 9 [Keywords], page 453.

or

Synopsis

```
operator or (operand1, operand2: Boolean) = Result: Boolean;
```

or

```
operator or (operand1, operand2: integer_type) = Result: integer_type;
```

or

```
procedure or (var operand1: integer_type; operand2: integer_type);
```

Description

In GNU Pascal, 'or' has three built-in meanings:

1. Logical "or" between two 'Boolean'-type expressions. The result of the operation is of 'Boolean' type.

   By default, 'or' acts as a short-circuit operator in GPC: If the first operand is 'True', the second operand is not evaluated because the result is already known to be 'True'. You can change this to complete evaluation using the '--no-short-circuit' command-line option or the '{$B+}' compiler directive.
2. Bitwise "or" between two integer-type expressions. The result is of the common integer type of both expressions.
3. Use as a "procedure": 'operand1' is "or"ed bitwise with 'operand2'; the result is stored in 'operand1'.

Conforming to

The logical 'or' operator is defined in ISO 7185 Pascal.

According to ISO, you cannot rely on 'or' being a short-circuit operator. On the other hand, GPC's default behaviour does *not* contradict the ISO standard. (See [or_else], page 378.) However, since it seems to be a de-facto standard among ISO Pascal compilers to evaluate both operands of 'or', GPC switches to '--no-short-circuit' mode if one of the language dialect options selecting ISO Pascal, for instance '--extended-pascal', is given. Use '--short-circuit' to override.

Use of 'or' as a bitwise operator for integers is a Borland Pascal extension.

Use of 'or' as a "procedure" is a GNU Pascal extension.

Example

```
program OrDemo;
var
  a, b, c: Integer;
begin
  if (a = 0) or (b = 0) then  { logical 'or' }
    c := 1
  else if (a or b) = 0 then  { bitwise 'or' }
    c := 2
  else
    or (c, a)  { same as 'c := c or a' }
end.
```

Note the difference between the logical 'or' and the bitwise 'or': When 'a' is 2 and 'b' is 4, then 'a or b' is 6. **Beware:** 'a or b = 0' happens to mean the same as '(a = 0) and (b = 0)'. (Note the 'and'!)

Since bitwise 'or' has a higher priority than the '=' operator, parentheses are needed in 'if (a = 0) or (b = 0)' because otherwise '0 or b' would be calculated first, and the remainder would cause a parse error.

See also

Chapter 9 [Keywords], page 453, [and], page 262, [xor], page 451, Section 6.3 [Operators], page 82.

Ord

Synopsis

```
function Ord (ordinal_value): Integer;
```

Description

'Ord' returns the ordinal value of any ordinal variable or constant. For characters, this would be the ASCII code corresponding to the character. For enumerated types, this would be the ordinal value of the constant or variable (remember that ordinal value of enumerated constants start from zero).

Conforming to

'Ord' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program OrdDemo;
var
  Ch: Char;
  Day: (Monday, Tuesday, Wednesday, Thursday, Friday);
begin
  Ch := 'A';
  WriteLn (Ord (Ch));  { 65 }
  Day := Thursday;
  WriteLn (Ord (Day));  { 3 }
end.
```

See also

Section 6.2.6 [Character Types], page 68, Section 6.2.2 [Ordinal Types], page 64, [Chr], page 290, [Char], page 288

or else

Synopsis

```
{ 'or else' is built in. A user-defined operator cannot consist of
   two words. }
operator or else (operand1, operand2: Boolean) = Result: Boolean;
```

Description

'or else' is an alias for the short-circuit logical operator 'or_else'.

Conforming to

While 'or_else' is defined in ISO 10206 Extended Pascal, 'or else' is a GNU Pascal extension.

Example

```
program OrElseDemo;
var
  a: Integer;
begin
  ReadLn (a);
  if (a = 0) or else (100 div a > 42) then  { This is safe. }
    WriteLn ('100 div a > 42')
end.
```

See also

Chapter 9 [Keywords], page 453, [or_else], page 378, [or], page 375, [and then], page 263.

or_else

Synopsis

```
operator or_else (operand1, operand2: Boolean) = Result: Boolean;
```

Description

The 'or_else' short-circuit logical operator performs the same operation as the logical operator 'or'. But while the ISO standard does not specify anything about the evaluation of the operands of 'or' – they may be evaluated in any order, or not at all – 'or_else' has a well-defined behaviour: It evaluates the first operand. If the result is 'True', 'or_else' returns 'True' without evaluating the second operand. If it is 'False', the second operand is evaluated and returned.

GPC by default treats 'or' and 'or_else' exactly the same. If you want, for some reason, to have both operands of 'or' evaluated completely, you must assign both to temporary variables and then use 'or' – or 'or_else', it does not matter.

Conforming to

'or_else' is an ISO 10206 Extended Pascal extension.

Some people think that the ISO standard requires both operands of 'or' to be evaluated. This is false. What the ISO standard *does* say is that you cannot rely on a certain order of evaluation of the operands of 'or'; in particular things like the following program can crash according to ISO Pascal, although they cannot crash when compiled with GNU Pascal running in default mode.

```
program OrBug;
var
  a: Integer;
begin
  ReadLn (a);
  if (a = 0) or (100 div a > 42) then  { This is *not* safe! }
    WriteLn ('You''re lucky. But the test could have crashed ...')
end.
```

Example

```
program Or_ElseDemo;
var
  a: Integer;
begin
  ReadLn (a);
  if (a = 0) or_else (100 div a > 42) then  { This is safe. }
    WriteLn ('100 div a > 42')
end.
```

See also

Chapter 9 [Keywords], page 453, [or else], page 377, [or], page 375, [and_then], page 264.

otherwise

Synopsis

Default '`case`' branch as part of the `case` ... `otherwise` statement:

```
case expression of
  selector: statement;
  ...
  selector: statement
otherwise  { ``else'' instead of ``otherwise'' is allowed }
  statement;
  ...
  statement
end
```

Use in a structured value of '`array`' type:

```
[index1: value1; index2: value2
 otherwise value_otherwise]
```

Description

'`otherwise`' starts a series of statements which is executed if no selector matches *expression*. In this situation, '`else`' is a synonym for `otherwise`.

'`otherwise`' also defines the default value in an Extended Pascal structured values of array type.

Conforming to

'`otherwise`' is an ISO 10206 Extended Pascal extension.

Example

```
program OtherwiseDemo;

var
  i: Integer;
```

```
  a: array [1 .. 10] of Integer value [1: 2; 4: 5 otherwise 3];

begin
  for i := 1 to 10 do
    case a[i] of
      2:        WriteLn ('a[', i, '] has value two.');
      3:        WriteLn ('a[', i, '] has value three.');
      otherwise WriteLn ('a[', i, '] has neither value two nor three.')
    end
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.4 [case Statement], page 56, [case], page 286, [else], page 310.

Output

(Under construction.)

Synopsis

```
var
  Output: Text;
```

Description

Conforming to

'`Output`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Pack

(Under construction.)

Synopsis

Description

```
procedure Pack (Source: unpacked_array;
                FirstElement: index_type;
                var Dest: packed_array);
```

Conforming to

'`Pack`' is defined in ISO 7185 Pascal and supported by all known Pascal variants except UCSD/Borland Pascal and its variants.

Example

See also

packed

Synopsis

Description

'`packed`' is a reserved word. According to ISO 7185 Pascal it can precede '`array`' and '`record`' type definitions to indicate that memory usage should be minimized for variables of this type, possibly at the expense of loss of speed.

As a GNU Pascal extension, '`packed`' can also be applied to Section 6.2.11.1 [Subrange Types], page 70.

Conforming to

The reserved word '`packed`' is defined in ISO 7185 Pascal.

According to ISO standard, only *packed* arrays of char with lower bound 1 qualify as strings of fixed length. GNU Pascal neither requires '`packed`' nor the lower bound of 1 here.

Example

```
program PackedDemo;

type
  MonthInt = packed 1 .. 12;  { needs one byte }
  FastMonthInt = 1 .. 12;     { needs e.g. four bytes }

  FixString10 = packed array [1 .. 10] of Char;
  FoxyString10 = array [0 .. 9] of Char;

  Flags = packed array [1 .. 32] of Boolean;  { needs four bytes }

  Int15 = Integer attribute (Size = 15);
  DateRec = packed record
    Day: 1 .. 31;       { five bits }
    Month: MonthInt;    { four bits }
    Year: Int15         { 15 bits = -16384 .. 16383 }
  end;

  Dates = array [1 .. 1000] of DateRec;

var
  S: FixString10;
  T: FoxyString10;

begin
```

```
    S := 'Hello!';  { blank padded }
    WriteLn (S);

    T := 'GNU Pascal';  { GPC extension: this also works. }
    WriteLn (T)
  end.
```

'DateRec' has 24 bits = 3 bytes in total; 'Dates' has 3000 bytes.

See also

Chapter 9 [Keywords], page 453, [Pack], page 380, [Unpack], page 440, [SizeOf], page 421, [AlignOf], page 261, [BitSizeOf], page 278.

Page

(Under construction.)

Synopsis

```
procedure Page ([var F: Text]);
```

or

```
procedure Page;
```

Description

Conforming to

'Page' is defined in ISO 7185 Pascal, but missing in Borland Pascal.

Example

See also

PAnsiChar

(Under construction.)

Synopsis

```
type
  PAnsiChar = ^AnsiChar;
```

Description

Conforming to

'PAnsiChar' is a Borland Delphi extension.

Example

```
program PAnsiCharDemo;
var
  s: PAnsiChar;
begin
  s := 'Hello, world!';
  {$X+}
  WriteLn (s)
end.
```

See also

ParamCount

Synopsis

```
function ParamCount: Integer;
```

Description

'ParamCount' returns the number of command-line arguments given to the program. 'ParamCount' returns 0 if no arguments have been given to the program; the name of the program as an implicit argument is not counted.

Conforming to

'ParamCount' is a Borland Pascal extension.

Example

```
program ParamCountDemo;

var
  i: Integer;

begin
  WriteLn ('You have invoked this program with ',
           ParamCount, ' arguments.');
  WriteLn ('These are:');
  for i := 1 to ParamCount do
    WriteLn (ParamStr (i))
end.
```

See also

[ParamStr], page 384.

ParamStr

(Under construction.)

Synopsis

```
function ParamStr (ParmNumber: Integer): String;
```

Description

Please note: If you are using the Dos (DJGPP) or MS-Windows (mingw32) version of GPC and are getting unexpected results from '`ParamStr`', please see the section "Command-line Arguments Handling in DJGPP" of the DJGPP FAQ list.

Conforming to

'`ParamStr`' is a Borland Pascal extension.

Example

```
program ParamStrDemo;

var
  i: Integer;

begin
  WriteLn ('You have invoked this program with ',
           ParamCount, ' arguments.');
  WriteLn ('These are:');
  for i := 1 to ParamCount do
    WriteLn (ParamStr (i))
end.
```

See also

PChar

(Under construction.)

Synopsis

```
type
  PChar = ^Char;
```

or

```
type
  PChar = CString;
```

Description

Conforming to

'`PChar`' is a Borland Pascal extension.

Example

```
program PCharDemo;
var
  s: PChar;
begin
  s := 'Hello, world!';
  {$X+}
  WriteLn (s)
end.
```

See also

Pi

(Under construction.)

Synopsis

Description

Conforming to

'Pi' is a Borland Pascal extension.

Example

See also

PObjectType

Synopsis

```
type
  InternalSignedSizeType =
    Integer attribute (Size = BitSizeOf (SizeType));
  PObjectType = ^const record
    Size:        SizeType;
    NegatedSize: InternalSignedSizeType;
    Parent:      PObjectType;
    Name:        ^const String
  end;
```

(Note: '^record' is not valid syntax. It is just used here in the explanation because the record type has no name by itself. Because of the added method pointers (see below), there is no useful usage of the record type.)

Description

'PObjectType' is the type returned by 'TypeOf' and required by 'SetType'. In fact, the record pointed to (the *VMT*, "virtual method table") also contains pointers to the virtual methods. However, these are not declared in 'PObjectType' because they vary from object type to object type. The fields declared here are those that are shared by every object type and can be accessed via 'TypeOf'.

'Size' contains the size of the object type, 'NegatedSize' contains the size negated (for runtime checks). 'Parent' contains a pointer to the parent type's VMT (or nil if the type has no parent). 'Name' points to a string containing the type's name.

Conforming to

'PObjectType' is a GNU Pascal extension.

Example

See also

[TypeOf], page 438, [SetType], page 414, Section 6.8 [OOP], page 86.

Pointer

(Under construction.)

Synopsis

```
type
  Pointer  { built-in type }
```

Description

Conforming to

'Pointer' is a Borland Pascal extension.

Example

```
program PointerDemo;
var
  a: Integer;
  b: Boolean;
  p: Pointer;
begin
  p := nil;
  p := @a;
  p := @b
end.
```

See also

Polar

(Under construction.)

Synopsis

```
function Polar (rho, phi: Real): Complex;
```

Description

Conforming to

'`Polar`' is an ISO 10206 Extended Pascal extension.

Example

See also

Pos

(Under construction.)

Synopsis

```
function Pos (SearchPattern, Source: String): Integer;
```

Description

Conforming to

'`Pos`' is a UCSD Pascal extension.

Example

See also

Position

(Under construction.)

Synopsis

```
function Position (var F: typed_file): index_type;
```

Description

Conforming to

'Position' is an ISO 10206 Extended Pascal extension.

Example

See also

pow

(Under construction.)

Synopsis

```
operator pow (base: Real; exponent: Integer) = power: Real;
```

or

```
operator pow (base: Complex; exponent: Integer) = power: Complex;
```

Description

Exponentiation operator with integer exponent.

Conforming to

'pow' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Pred

Synopsis

```
function Pred (i: ordinal_type): ordinal_type;
```

or

```
function Pred (i: ordinal_type; j: and_integer_type): ordinal_type;
```

or, with extended syntax ('--extended-syntax' or '{$X+}'),

```
function Pred (p: Pointer_type): Pointer_type;
```

or

```
function Pred (p: Pointer_type; j: and_integer_type): Pointer_type;
```

Description

Returns the predecessor of the *ordinal_type* value '`i`', or, if the second argument '`j`' is given, its '`j`'th predecessor. For integer values '`i`', this is '`i - 1`' (or '`i - j`'). (No, '`Pred`' does *not* work faster than plain subtraction. Both are optimized the same way, often to a single machine instruction.)

If extended syntax is on, the argument may also be a pointer value. In this case, the address is decremented by the size of the variable pointed to, or, if '`j`' is given, by '`j`' times the size of the variable pointed to. If '`p`' points to an element of an array, the returned pointer will point to the ('`j`'th) previous element of the array.

Conforming to

The '`Pred`' function is defined in ISO 7185 Pascal. The optional second parameter is defined in ISO 10206 Extended Pascal. Application of '`Pred`' to pointers is defined in Borland Pascal. The combination of the second argument with application to pointers is a GNU Pascal extension.

Example

```
program PredDemo;

type
  Metasyntactical = (foo, bar, baz);

var
  m: Metasyntactical;
  c: Char;
  a: array [1 .. 7] of Integer;
  p: ^Integer;

begin
  m := Pred (bar);     { foo }
  c := Pred ('Z', 2);  { 'X' }
  a[1] := 42;
  a[4] := Pred (a[1]);     { 41 }
  a[5] := Pred (a[4], 3);  { 38 }
  {$X+}
  p := @a[5];
  p := Pred (p);     { now p points to a[4] }
  p := Pred (p, 3);  { now p points to a[1] }
end.
```

See also

[Succ], page 428, [Dec], page 304, Section 6.6 [Pointer Arithmetics], page 84.

private

(Under construction.)

Synopsis

Description

GPC currently accepts but ignores the '`private`' directive in object type declarations.

Conforming to

'`private`' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453, [protected], page 391, [public], page 394, [published], page 395.

procedure

(Under construction.)

Synopsis

Description

Procedure declaration.

Conforming to

'`procedure`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

program

(Under construction.)

Synopsis

Description

Start of a Pascal program.

Conforming to

'`program`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

property

Not yet implemented.

Synopsis

Description

Object properties.

Conforming to

'`property`' is an Object Pascal and a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453.

protected

(Under construction.)

Synopsis

Description

The Extended Pascal meaning of '`protected`' is supported by GPC.
GPC currently accepts but ignores the '`protected`' directive in object type declarations.

Conforming to

Extended Pascal and Borland Pascal, but with different meanings.

Example

See also

Chapter 9 [Keywords], page 453, [const], page 296, [import], page 336, [private], page 389, [public], page 394, [published], page 395.

PtrCard

(Under construction.)

Synopsis

```
type
  PtrCard = Cardinal attribute (Size = BitSizeOf (Pointer));
```

Description

An unsigned integer type of the same size as a pointer.

Conforming to

'PtrCard' is a GNU Pascal extension.

Example

```
program PtrCardDemo;
var
  a: PtrCard;
  p: Pointer;
begin
  GetMem (p, 10);
  a := PtrCard (p);
  Inc (a);
  p := Pointer (a)
end.
```

See also

PtrDiffType

(Under construction.)

Synopsis

```
type
  PtrDiffType  { built-in type }
```

Description

'PtrDiffType' is a (signed) integer type to represent the difference between two positions in memory. It is not needed except for rather low-level purposes.

Conforming to

'PtrDiffType' is a GNU Pascal extension.

Example

```
program PtrDiffTypeDemo;
var
  a: array [1 .. 10] of Integer;
  d: PtrDiffType;
  p, q: ^Integer;
begin
  p := @a[1];
  q := @a[4];
  {$X+}
  d := q - p
end.
```

See also

PtrInt

(Under construction.)

Synopsis

```
type
  PtrCard = Integer attribute (Size = BitSizeOf (Pointer));
```

Description

A signed integer type of the same size as a pointer.

Conforming to

'PtrInt' is a GNU Pascal extension.

Example

```
program PtrIntDemo;
var
  a: PtrInt;
  p: Pointer;
begin
  GetMem (p, 10);
  a := PtrInt (p);
  Inc (a);
  p := Pointer (a)
end.
```

See also

PtrWord

(Under construction.)

Synopsis

```
type
  PtrWord = PtrCard;
```

Description

An unsigned integer type of the same size as a pointer.

Conforming to

'PtrWord' is a GNU Pascal extension.

Example

```
program PtrWordDemo;
var
  a: PtrWord;
  p: Pointer;
begin
  GetMem (p, 10);
  a := PtrWord (p);
  Inc (a);
  p := Pointer (a)
end.
```

See also

[PtrCard], page 392, Section 6.2.3 [Integer Types], page 64.

public

(Under construction.)

Synopsis

Description

GPC currently accepts but ignores the 'public' directive in object type declarations.

Conforming to

'public' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453, [private], page 389, [protected], page 391, [published], page 395.

published

(Under construction.)

Synopsis

Description

GPC currently accepts but ignores the ‘`published`’ directive in object type declarations.

Conforming to

‘`published`’ is a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [private], page 389, [protected], page 391, [public], page 394.

Put

(Under construction.)

Synopsis

```
procedure Put (var F: typed_file);
```

Description

Conforming to

‘`Put`’ is defined in ISO 7185 Pascal and supported by all known Pascal variants except UCSD/Borland Pascal and its variants.

Example

See also

qualified

(Under construction.)

Synopsis

Description

Import specification.

Conforming to

'`qualified`' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Random

(Under construction.)

Synopsis

Description

Conforming to

'`Random`' is a UCSD Pascal extension.

Example

See also

Randomize

(Under construction.)

Synopsis

Description

Conforming to

'`Randomize`' is a UCSD Pascal extension.

Example

See also

Re

Synopsis

```
function Re (z: Complex): Real;
```

Description

'Re' extracts the real part of the complex number 'z'.

Conforming to

'Re' is an ISO 10206 Extended Pascal extension.

Example

```
program ReDemo;
var
  z: Complex;
begin
  z := Cmplx (1, 2);
  WriteLn (Re (z) : 0 : 5)
end.
```

See also

[Cmplx], page 292, [Im], page 335, [Arg], page 269

Read

(Under construction.)

Synopsis

```
procedure Read (var F: typed_file; variable);
```

or

```
procedure Read (var F: Text; variables);
```

or

```
procedure Read (variables);
```

Description

Conforming to

'Read' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

ReadLn

(Under construction.)

Synopsis

```
procedure ReadLn (var F: Text; variables);
```
or
```
procedure ReadLn (variables);
```

Description

Conforming to

'ReadLn' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

ReadStr

(Under construction.)

Synopsis

```
procedure ReadStr (const S: String; variables);
```

Description

Conforming to

'ReadStr' is an ISO 10206 Extended Pascal extension.

Example

See also

Real

(Under construction.)

Synopsis

```
type
  Real  { built-in type }
```

Description

Conforming to

'Real' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program RealDemo;
var
  a: Real;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.4 [Real Types], page 68, [Int], page 342, [Frac], page 327, [Round], page 407, [Trunc], page 435.

record

Synopsis

In type definitions:

```
record_type_identifier = record
  field_identifier: type_definition
  ...
  field_identifier: type_definition
end;
```

or, with a variant part,

```
record_type_identifier = record
  field_identifier: type_definition
  ...
  field_identifier: type_definition
  case bar: variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
end;
```

or, without a variant selector field,

```
record_type_identifier = record
  field_identifier: type_definition
  ...
  field_identifier: type_definition
  case variant_type of
    selector: (field_declarations);
    selector: (field_declarations);
    ...
end;
```

Description

The reserved word '`record`' starts the definition of a new record type.

Records can be '`packed`' to save memory usage at the expense of speed.

The variants of a variant record may – but are not required to – share one location in memory (inside the record).

Sometimes variant records are used to emulate type casting in ISO 7185 Pascal. This is in fact a violation of the standard and not portable. There is intentionally *no* possibility in ISO 7185 Pascal to emulate type casting.

Conforming to

The reserved word '`record`' and record types are defined in ISO 7185 Pascal.

According to ISO Pascal, the variant type must be an identifier. GNU Pascal, like UCSD and Borland Pascal, also allows a subrange here.

Subranges in the variant fields, e.g. `case Integer of 2 .. 5`, are a GPC extension.

Example

```
program RecordDemo;

type
  FooPtr = ^Foo;

  Foo = record
    Bar: Integer;
    NextFoo: FooPtr;
    case Choice: 1 .. 3 of
      1: (a: Integer);  { These three choices may share }
      2: (b: Real);     { one location in memory. }
      3: (c: Char;
          d: Boolean);
  end;

  Int5 = Integer attribute (Size = 5);
  SmallFoo = packed record
    b: 0 .. 3;
    a: Int5;
    r: Boolean
  end;  { needs 1 byte }

var
  f: Foo;

begin
  f.b := 3.14;
  WriteLn (f.a)  { yields some strange number which is part of the   }
                 { internal representation of the real number 'f.b'. }
end.
```

See also

Chapter 9 [Keywords], page 453, [packed], page 381, Section 6.1.7.4 [case Statement], page 56

Release

(Under construction.)

Synopsis

```
procedure Release (P: Pointer);
```

Description

Conforming to

'`Release`' is a UCSD Pascal extension.

Example

See also

Rename

(Under construction.)

Synopsis

```
procedure Rename (var F: any_file; NewName: String);
```

Description

Conforming to

'`Rename`' is a Borland Pascal extension.

Example

See also

repeat

Synopsis

```
repeat
  statement;
  ...
  statement;
until boolean_expression;
```

Description

The 'repeat ... until' statement declares a loop. For further description see Section 6.1.7.7 [repeat Statement], page 59.

Conforming to

'repeat' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program RepeatDemo;
var
  Number, Sum: Integer;
begin
  WriteLn ('Black Jack for beginners.');
  WriteLn ('You can choose your cards yourself. :-)');
  Sum := 0;
  repeat
    Write ('Your card (number)? ');
    ReadLn (Number);
    Inc (Sum, Number);
    WriteLn ('You have ', Sum, '.')
  until Sum >= 21;
  if Sum = 21 then
    WriteLn ('You win!')
  else
    WriteLn ('You lose.')
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.6 [while Statement], page 58, Section 6.1.7.5 [for Statement], page 57.

Reset

(Under construction.)

Synopsis

```
procedure Reset (var F: any_file; [FileName: String;]
                                    [BlockSize: Cardinal]);
```

Description

'Reset' opens an existing file for reading. The file pointer is positioned at the beginning of the file.

Like 'Rewrite', 'Append' and 'Extend' do, 'Reset' accepts an optional second parameter for the name of the file in the filesystem and a third parameter for the block size of the file. The third parameter is allowed only (and by default also required) for untyped files. For details, see [Rewrite], page 405.

Conforming to

'Reset' is defined in ISO 7185 Pascal. The 'BlockSize' parameter is a Borland Pascal extension. The 'FileName' parameter is a GNU Pascal extension.

Example

```
program ResetDemo;
var
  Sample: Text;
  s: String (42);
begin
  Rewrite (Sample);  { Open an internal file for writing }
  WriteLn (Sample, 'Hello, World!');
  Reset (Sample);  { Open it again for reading }
  ReadLn (Sample, s);
  WriteLn (s);
  Close (Sample)
end.
```

See also

[Assign], page 272, [Rewrite], page 405, [Append], page 266, [Extend], page 318.

resident

Not yet implemented.

Synopsis

Description

Library export specification.

Conforming to

'resident' is a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

restricted

(Under construction.)

Synopsis

Description

Restricted type specification.

Conforming to

'`restricted`' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Result

(Under construction.)

Synopsis

Description

Please note: GPC *deactivates* '`Result`' by default because it is incompatible with standard program semantics. Only in '`--delphi`' mode or with the option '`--implicit-result`', it is activated. Explicit result variables provide a safer alternative that works in default mode.

Conforming to

'`Result`' is a Borland Delphi extension.

Example

See also

Return

(Under construction.)

Synopsis

Description

Conforming to

'Return' is a GNU Pascal extension.

Example

See also

ReturnAddress

(Under construction.)

Synopsis

Description

Conforming to

'ReturnAddress' is a GNU Pascal extension.

Example

See also

Rewrite

(Under construction.)

Synopsis

```
procedure Rewrite (var F: any_file; [FileName: String;]
                                       [BlockSize: Cardinal]);
```

Description

'Rewrite' opens a file for writing. If the file does not exist, it is created. The file pointer is positioned at the beginning of the file.

Like 'Reset', 'Append' and 'Extend' do, 'Rewrite' accepts an optional second and third parameter.

The second parameter can specify the name of the file in the filesystem. If it is omitted, the following alternative ways can be used to specify the name. There are so many different ways in order to be compatible to the idiosyncrasies of as many other Pascal compilers as possible. (If you know about yet other ways, let us know . . .)

- The 'Assign' procedure (see [Assign], page 272)
- The 'Bind' procedure (see [Bind], page 276)

The following ways are only available if the file is external, i.e. a global variable which is mentioned in the program header. Otherwise, the file will be internal, i.e. get no name in the file system (it may get a name temporarily, but will then be erased automatically again). This is useful to store some data and read them back within a program without the need for permanent storage.

- A command-line parameter of the form '`--gpc-rts=-n`*f*`:`*name*' where *f* is the identifier of the file variable.
- If the file was mentioned in the program header and the option '`--transparent file names`' (see Section 5.1 [GPC Command Line Options], page 35) was set, the file name will be identical to the identifier converted to lower-case.
- Otherwise, the user will be prompted for a file name.

The last optional parameter determines the block size of the file. It is valid only for untyped files. Often 1 is a reasonable value here. However, the existence of this parameter is a BP compatibility feature, and in BP it defaults to 128 because of historic misdesign. Therefore, GPC requires this parameter to be present. In '`--borland-pascal`' mode, it makes it optional (like BP does), but warns about the strange default if omitted.

Conforming to

'`Rewrite`' is defined in ISO 7185 Pascal. The '`BlockSize`' parameter is a Borland Pascal extension. The '`FileName`' parameter is a GNU Pascal extension.

Example

```
program RewriteDemo;
var
  Sample: Text;
begin
  Assign (Sample, 'sample.txt');
  Rewrite (Sample);
  WriteLn (Sample, 'Hello, World!');
  Close (Sample)
end.
```

See also

[Assign], page 272, [Reset], page 402, [Append], page 266, [Extend], page 318, [Update], page 441.

RmDir

Synopsis

```
procedure RmDir (Directory: String);
```

Description

'`RmDir`' removes the given *Directory* if its argument is a valid parameter to the related operating system's function. Otherwise a runtime error is caused.

Conforming to

'RmDir' is a Borland Pascal extension.

Example

```
program RmDirDemo;
var
  Foo: String (127);
begin
  WriteLn ('Enter directory name to remove: ');
  ReadLn (Foo);
  {$I-}  { Don't abort on I/O errors }
  RmDir (Foo);
  if IOResult <> 0 then
    WriteLn ('Directory ', Foo, ' could not be removed.')
  else
    WriteLn ('Okay.')
end.
```

See also

[ChDir], page 289, [MkDir], page 364

Round

Synopsis

```
function Round (x: Real): Integer;
```

Description

'Round' returns the nearest integer to 'x'. The result is of type integer. In the case of equidistance, the result is machine-dependent (or depends on the behaviour of the processor).

Conforming to

'Round' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program RoundDemo;
var
  Foo: Real;
begin
  Foo := 9.876543;
  WriteLn (Round (Foo));  { Prints 10 }
  Foo := 3.456789;
  WriteLn (Round (Foo));  { Prints 3 }

  WriteLn (Frac (12.345) : 1 : 5);  { 0.34500 }
```

```
  WriteLn (Int (12.345) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.345) : 1);  { 12 }
  WriteLn (Trunc (12.345) : 1);  { 12 }

  WriteLn (Frac (-12.345) : 1 : 5);  { -0.34500 }
  WriteLn (Int (-12.345) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.345) : 1);  { -12 }
  WriteLn (Trunc (-12.345) : 1);  { -12 }

  WriteLn (Frac (12.543) : 1 : 5);  { 0.54300 }
  WriteLn (Int (12.543) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.543) : 1);  { 13 }
  WriteLn (Trunc (12.543) : 1);  { 12 }

  WriteLn (Frac (-12.543) : 1 : 5);  { -0.54300 }
  WriteLn (Int (-12.543) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.543) : 1);  { -13 }
  WriteLn (Trunc (-12.543) : 1);  { -12 }
end.
```

See also

Section 6.2.4 [Real Types], page 68, [Real], page 398, [Int], page 342, [Frac], page 327, [Trunc], page 435.

RunError

(Under construction.)

Synopsis

```
procedure RunError (ErrorCode: Integer);
```

Description

Conforming to

'`RunError`' is a Borland Pascal extension.

Example

See also

Seek

(Under construction.)

Synopsis

procedure Seek (var F: *typed_file*; NewPosition: *index_type*);

Description

Conforming to

'Seek' is a UCSD Pascal extension.

Example

See also

SeekEOF

(Under construction.)

Synopsis

```
function SeekEOF ([var F: Text]): Boolean;
```

Description

Conforming to

'SeekEOF' is a Borland Pascal extension.

Example

See also

SeekEOLn

(Under construction.)

Synopsis

```
function SeekEOLn ([var F: Text]): Boolean;
```

Description

Conforming to

'SeekEOLn' is a Borland Pascal extension.

Example

See also

SeekRead

(Under construction.)

Synopsis

```
procedure SeekRead (var F: typed_file; NewPosition: index_type);
```

Description

Conforming to

'SeekRead' is an ISO 10206 Extended Pascal extension.

Example

See also

SeekUpdate

(Under construction.)

Synopsis

```
procedure SeekUpdate (var F: typed_file; NewPosition: index_type);
```

Description

Conforming to

'SeekUpdate' is an ISO 10206 Extended Pascal extension.

Example

See also

SeekWrite

(Under construction.)

Synopsis

```
procedure SeekWrite (var F: typed_file; NewPosition: index_type);
```

Description

Conforming to

'`SeekWrite`' is an ISO 10206 Extended Pascal extension.

Example

See also

segment

Not yet implemented.

Synopsis

Description

Segment specification.

Conforming to

'`segment`' is a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Self

(Under construction.)

Synopsis

Description

Conforming to

'`Self`' is an Object Pascal and a Borland Pascal extension.

Example

See also

set

Synopsis

In type definitions:

```
set of ordinal_type  { built-in type class }
```

Description

A set contains zero or more elements from an ordinal type, e.g. Char, a subrange of Char, or a subrange of an enumerated type or integers. The elements of a sets do not have any ordering (that is a set containing 'B' and 'A' is the same as a set containing 'A' and 'B'), nor can an element be included more than once. Sets simply store the information about which elements are included in the set.

Conforming to

'`set`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program SetDemo;

type
  TCharSet = set of Char;

var
  Ch: Char;
  MyCharSet: TCharSet;
begin
  MyCharSet := ['P','N','L'];
  if 'A' in MyCharSet then
    WriteLn ('Wrong: A in set MyCharSet')
  else
    WriteLn ('Right: A is not in set MyCharSet');
  Include (MyCharSet, 'A');  { A, L, N, P }
  Exclude (MyCharSet, 'N');  { A, L, P }
  MyCharSet := MyCharSet + ['B','C'];  { A, B, C, L, P }
  MyCharSet := MyCharSet - ['C','D'];  { A, B, L, P }
  WriteLn ('set MyCharSet contains:');
  for Ch in MyCharSet do
    WriteLn (Ch);
end.
```

Set also Section 6.10.7 [Set Operations], page 98 or examples of some of the many set operations.

See also

Chapter 9 [Keywords], page 453, Section 6.10.7 [Set Operations], page 98, [in], page 337, [Exclude], page 314, [Include], page 339.

SetFileTime

```
procedure SetFileTime (var f: any_file;
                       AccessTime, ModificationTime: UnixTimeType);
```

Synopsis

Description

Conforming to

'SetFileTime' is a GNU Pascal extension.

Example

See also

SetLength

Synopsis

```
procedure SetLength (var S: String; NewLength: Integer);
```

Description

'SetLength' explicitly assigns a new length 'NewLength' to the string parameter 'S'. The contents of the string elements (characters) are *not* changed; if the operation increases the length of the string, the characters appended at the end are *undefined*.

Conforming to

'SetLength' is a Borland Delphi 2.0 extension.

Example

```
program SetLengthDemo;

var
  S: String (26);

begin
  S := 'Hello, world!';
  SetLength (S, Length ('Hello'));
  WriteLn (S);                                          { 'Hello' }

  SetLength (S, 26);
  WriteLn (S);                      { 'Hello, world!(%$xy"!#&~+(/]' }
                                { undefined characters ~~~~~~~~~~~~~~  }

  { SetLength (S, 42);  GPC detects this overflow at compile time. }
end.
```

See also

[Length], page 347, [String], page 427.

SetType

Synopsis

```
procedure SetType (var SomeObject; VMT: PObjectType);
```

Description

The procedure 'SetType' explicitly assigns a value to the implicit VMT field of an object. This is normally done implicitly when a constructor is called.

You can use this to write a polymorphic I/O routine which reads an object from a file. In this case, you cannot reasonably use 'New' to allocate the storage, but you 'GetMem' it and initialize the object manually using 'SetType' before calling the constructor explicitly.

The only values you should assign to an object via 'SetType' are actual VMT pointers that were obtained via 'TypeOf'. In particular, declaring a record like the one shown in the description of 'PObjectType' and assigning a pointer to it to an object via 'SetType' will usually not work because the virtual method pointers are missing.

Since 'SetType' is a dangerous feature, it yields a warning unless '{$X+}' is given.

Conforming to

'SetType' is a GNU Pascal extension.

Example

```
program SetTypeDemo;

type
  BasePtr = ^BaseObj;

  BaseObj = object
    constructor Load;
  end;

  ChildObj = object (BaseObj)
    constructor Load;
  end;

constructor BaseObj.Load;
begin
end;

constructor ChildObj.Load;
begin
end;

{$X+}
```

```
{ This is somewhat fragmentary code. }
function GetObject (var InputFile: File) = Result: BasePtr;
const
  VMTTable: array [1 .. 2] of PObjectType =
    (TypeOf (BaseObj), TypeOf (ChildObj));
var
  Size: Cardinal;
  TypeID: Integer;
  VMT: PObjectType;
begin
  { Read the size of the object from some file and store it in ‘Size’. }
  BlockRead (InputFile, Size, SizeOf (Size));

  { Allocate memory for the object. }
  GetMem (Result, Size);

  { Read some ID from some file. }
  BlockRead (InputFile, TypeID, SizeOf (TypeID));

  { Look up the ‘VMT’ from some table. }
  { Explicit range checking wouldn’t be a bad idea here ... }
  { Without it, a wrong value just leads to a runtime error. }
  VMT := VMTTable[TypeID];

  SetType (Result^, VMT);

  { Now the object is ready, and the constructor can be called. }
  { Look up the correct constructor from some table and call it. }
end;

begin
end.
```

See also

[PObjectType], page 385, [TypeOf], page 438, Section 6.8 [OOP], page 86.

shl

Synopsis

```
operator shl (operand1, operand2: integer_type) = Result: integer_type;
```

or

```
procedure shl (var operand1: integer_type; operand2: integer_type);
```

Description

In GNU Pascal, ‘shl’ has two built-in meanings:

1. Bitwise shift left of an integer-type expression by another integer value. The result is of the type of the first operand.

2. Use as a “procedure”: ‘operand1’ is shifted left by ‘operand2’; the result is stored in ‘operand1’.

Conforming to

‘shl’ is a Borland Pascal extension.
Use of ‘shl’ as a “procedure” is a GNU Pascal extension.

Example

```
program ShlDemo;
var
  a: Integer;
begin
  a := 1 shl 7;  { yields 128 = 2 pow 7 }
  shl (a, 4)  { same as ‘a := a shl 4’ }
end.
```

See also

Chapter 9 [Keywords], page 453, [shr], page 419, Section 6.3 [Operators], page 82.

ShortBool

Synopsis

```
type
  ShortBool = Boolean attribute (Size = BitSizeOf (ShortInt));
```

Description

The intrinsic ‘ShortBool’ represents boolean values, but occupies the same memory space as a ‘ShortInt’. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

‘ShortBool’ is a GNU Pascal extension.

Example

```
program ShortBoolDemo;
var
  a: ShortBool;
begin
  ShortInt (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [MedBool], page 360, [WordBool], page 449, [LongBool], page 350, [LongestBool], page 351.

ShortCard

Synopsis

```
type
  ShortCard = Cardinal attribute (Size = BitSizeOf (ShortInt));
```

Description

'ShortCard' is an unsigned integer type which is not larger than 'Cardinal'. On some platforms it is 16 bits wide and thus has a range of '0 .. 65535'.

'ShortCard' in GNU Pascal is compatible to 'short unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'ShortCard' is a GNU Pascal extension.

Example

```
program ShortCardDemo;
var
  a: ShortCard;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

ShortInt

Synopsis

```
type
  ShortInt  { built-in type }
```

Description

'ShortInt' is a signed integer type which is not larger than 'Integer'. On some platforms it is 16 bits wide and thus has a range of '-32768 .. 32767'.

'ShortInt' in GNU Pascal is compatible to 'short int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'ShortInt' is a Borland Pascal extension. In Borland Pascal, 'ShortInt' is an 8-bit signed integer type ('ByteInt' in GNU Pascal).

Example

```
program ShortIntDemo;
var
  a: ShortInt;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

ShortReal

(Under construction.)

Synopsis

```
type
  ShortReal  { built-in type }
```

Description

Conforming to

'ShortReal' is a GNU Pascal extension.

Example

```
program ShortRealDemo;
var
  a: ShortReal;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

ShortWord

Synopsis

```
type
  ShortWord = ShortCard;
```

Description

'ShortWord' is an unsigned integer type which is not larger than 'Word'. On some platforms it is 16 bits wide and thus has a range of '0 .. 65535'. It is the same as [ShortCard], page 417.

'ShortWord' in GNU Pascal is compatible to 'short unsigned int' in GNU C.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'ShortWord' is a GNU Pascal extension.

'ShortWord' in GNU Pascal essentially corresponds to 'Word' in Borland Pascal and Delphi where it is a 16-bit unsigned integer type.

Example

```
program ShortWordDemo;
var
  a: ShortWord;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[ShortCard], page 417, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

shr

Synopsis

```
operator shr (operand1, operand2: integer_type) = Result: integer_type;
```

or

```
procedure shr (var operand1: integer_type; operand2: integer_type);
```

Description

In GNU Pascal, 'shr' has two built-in meanings:

1. Bitwise shift right of an integer-type expression by another integer value. The result is of the type of the first operand.
2. Use as a "procedure": 'operand1' is shifted right by 'operand2'; the result is stored in 'operand1'.

Conforming to

'shr' is a Borland Pascal extension.

Unlike the Borland compilers, GNU Pascal cares about the signedness of the first operand: If a signed integer with a negative value is shifted right, "one" bits are filled in from the left.

Use of 'shr' as a "procedure" is a GNU Pascal extension.

Example

```
program ShrDemo;
var
  a: Integer;
begin
  a := 1024 shr 4;  { yields 64 }
  a := -127 shr 4;  { yields -8 }
  shr (a, 2)  { same as 'a := a shr 2' }
end.
```

See also

Chapter 9 [Keywords], page 453, [shl], page 415, Section 6.3 [Operators], page 82.

Sin

Synopsis

```
function Sin (x: Real): Real;
```

or

```
function Sin (z: Complex): Complex;
```

Description

'`Sin`' returns the sine of the argument. The result is in the range '`-1 <= Sin (x) <= 1`' for real arguments.

Conforming to

The function '`Sin`' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program SinDemo;
begin
  { yields 0.5 since Sin (Pi / 6) = 0.5 }
  WriteLn (Sin (Pi / 6) : 0 : 5)
end.
```

See also

[ArcTan], page 268, [Cos], page 299, [Ln], page 349, [Arg], page 269.

Single

(Under construction.)

Synopsis

```
type
  Single = ShortReal;
```

Description

Conforming to

'Single' is a Borland Pascal extension.

Example

See also

SizeOf

Synopsis

```
function SizeOf (var x): SizeType;
```

Description

Returns the size of a type or variable in bytes.

'SizeOf' can be applied to expressions and type names. If the argument is a polymorphic object, the size of its actual type is returned.

Conforming to

'SizeOf' is a UCSD Pascal extension.

Example

```
program SizeOfDemo;
var
  a: Integer;
  b: array [1 .. 8] of Char;
begin
  WriteLn (SizeOf (a));        { Size of an 'Integer'; often 4 bytes. }
  WriteLn (SizeOf (Integer));  { The same. }
  WriteLn (SizeOf (b))         { Size of eight 'Char's; usually 8 bytes. }
end.
```

See also

[BitSizeOf], page 278, [AlignOf], page 261, [TypeOf], page 438.

SizeType

Synopsis

```
type
  SizeType  { built-in type }
```

Description

'SizeType' is an integer type (usually unsigned) to represent the size of objects in memory.

Conforming to

'SizeType' is a GNU Pascal extension.

Example

```
program SizeTypeDemo;
var
  a: array [1 .. 10] of Integer;
  Size: SizeType;
begin
  Size := SizeOf (a);
  WriteLn (Size)
end.
```

See also

SmallInt

Synopsis

```
type
  SmallInt = ShortInt;
```

Description

'SmallInt' is a signed integer type which is not larger than 'Integer'. On some platforms it is 16 bits wide and thus has a range of '-32768 .. 32767'. It is the same as 'ShortInt' (see [ShortInt], page 417).

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'SmallInt' is a Borland Delphi 2.0 extension.

Example

```
program SmallIntDemo;
var
  a: SmallInt;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[ShortInt], page 417, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

Sqr

Synopsis

```
function Sqr (i: integer_type): integer_type;
```
or
```
function Sqr (x: real_type): real_type;
```
or
```
function Sqr (z: complex_type): complex_type;
```

Description

Returns the square of the argument:

```
function Sqr (x: some_type): some_type;
begin
  Sqr := x * x  { or: x pow 2 }
end;
```

Conforming to

The function '`Sqr`' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program SqrDemo;

var
  i: Complex;

begin
  i := Cmplx (0, 1);
  WriteLn (Re (Sqr (i)) : 0 : 3)  { yields -1.000 }
end.
```

See also

[pow], page 388, [SqRt], page 424, [Abs], page 257, Section 6.3 [Operators], page 82.

SqRt

Synopsis

```
function SqRt (x: real_type): real_type;
```

or

```
function SqRt (z: complex_type): complex_type;
```

Description

Returns the positive square root of the argument.

For real arguments, it is an error if the argument is negative.

For complex arguments, 'SqRt' returns the principal value of the root of the argument, i.e. the root with positive real part, or, if the real part is zero, that one with positive imaginary part.

Conforming to

The function 'SqRt' is defined in ISO 7185 Pascal; its application to complex values is defined in ISO 10206 Extended Pascal.

Example

```
program SqRtDemo;

var
  m1: Complex;

begin
  m1 := Cmplx (-1, 0);  { -1 }
  WriteLn (Re (SqRt (m1)) : 6 : 3, Im (SqRt (m1)) : 6 : 3);
    { yields 1.000 -1.000, i.e. the imaginary unit, i }
end.
```

See also

[pow], page 388, [Sqr], page 423, Section 6.3 [Operators], page 82.

StandardError

(Under construction.)

Synopsis

Description

Conforming to

'StandardError' is a GNU Pascal extension.

Example

See also

StandardInput

(Under construction.)

Synopsis

Description

Conforming to

'`StandardInput`' is an ISO 10206 Extended Pascal extension.

Example

See also

StandardOutput

(Under construction.)

Synopsis

Description

Conforming to

'`StandardOutput`' is an ISO 10206 Extended Pascal extension.

Example

See also

StdErr

Synopsis

```
var
  StdErr: Text;
```

Description

The 'StdErr' variable is connected to the standard error file handle. To report errors, you should prefer 'WriteLn (StdErr, 'everything wrong')' over 'WriteLn ('everything wrong')'.

Conforming to

'StdErr' is a GNU Pascal extension.

Example

```
program StdErrDemo;
var
  Denominator: Integer;
begin
  ReadLn (Denominator);
  if Denominator = 0 then
    WriteLn (StdErr, ParamStr (0), ': division by zero')
  else
    WriteLn ('1 / ', Denominator, ' = ', 1 / Denominator)
end.
```

See also

[StandardError], page 424, [Output], page 380, [Input], page 342.

Str

(Under construction.)

Synopsis

procedure Str (x: *integer_or_real*; var Dest: String);

or

procedure Str (x: *integer_or_real* : *field_width*; var Dest: String);

or

procedure Str (x: Real : *field_width* : *precision*; var Dest: String);

or

procedure Str (*repeated_constructs_as_described_above*; var Dest: String);

Description

Conforming to

'Str' is a UCSD Pascal extension, generalized by Borland Pascal. The possibility to handle more than one variable in one 'Str' statement is a GNU Pascal extension.

ISO 10206 Extended Pascal defines 'WriteStr' instead of 'Str'.

Example

See also

[WriteStr], page 450.

String

(Under construction.)

Synopsis

Description

Conforming to

'`String`' is an Extended Pascal and a UCSD Pascal extension.

Example

See also

String2CString

(Under construction.)

Synopsis

```
function String2CString (const S: String): CString;
```

Description

Conforming to

'`String2CString`' is a GNU Pascal extension.

Example

See also

SubStr

Synopsis

```
function SubStr (S: String; FirstChar: Integer): String;
```

or

```
function SubStr (S: String; FirstChar, Count: Integer): String;
```

Description

'SubStr' returns a sub-string of *S* starting with the character at position *FirstChar*. If *Count* is given, such many characters will be copied into the sub-string. If *Count* is omitted, the sub-string will range to the end of *S*.

If 'Count' is too large for the sub-string to fit in *S* or if 'FirstChar' exceeds the length of *S*, 'SubStr' triggers a runtime error. (For a function returning the empty string instead, see [Copy], page 298.)

Conforming to

'SubStr' is an ISO 10206 Extended Pascal extension.

Example

```
program SubStrDemo;
var
  S: String (42);
begin
  S := 'Hello';
  WriteLn (SubStr (S, 2, 3));   { yields 'ell' }
  WriteLn (SubStr (S, 3));      { yields 'llo' }
  WriteLn (SubStr (S, 4, 7));   { yields a runtime error }
  WriteLn (SubStr (S, 42));     { yields a runtime error }
end.
```

See also

[Copy], page 298, Section 6.5 [String Slice Access], page 83.

Succ

Synopsis

`function Succ (i:` *ordinal_type*`):` *ordinal_type*`;`

or

`function Succ (i:` *ordinal_type*`; j:` *and_integer_type*`):` *ordinal_type*`;`

or, with extended syntax ('--extended-syntax' or '{$X+}'),

`function Succ (p:` *Pointer_type*`):` *Pointer_type*`;`

or

`function Succ (p:` *Pointer_type*`; j:` *and_integer_type*`):` *Pointer_type*`;`

Description

Returns the successor of the *ordinal_type* value 'i', or, if the second argument 'j' is given, its 'j'th successor. For integer values 'i', this is 'i + 1' (or 'i + j'). (No, 'Succ' does *not* work faster than plain addition. Both are optimized the same way, often to a single machine instruction.)

If extended syntax is on, the argument may also be a pointer value. In this case, the address is incremented by the size of the variable pointed to, or, if 'j' is given, by 'j' times the size of the variable pointed to. If 'p' points to an element of an array, the returned pointer will point to the ('j'th) next element of the array.

Conforming to

The 'Succ' function is defined in ISO 7185 Pascal. The optional second parameter is defined in ISO 10206 Extended Pascal. Application of 'Succ' to pointers is defined in Borland Pascal. The combination of the second argument with application to pointers is a GNU Pascal extension.

Example

```
program SuccDemo;

type
  Metasyntactical = (foo, bar, baz);

var
  m: Metasyntactical;
  c: Char;
  a: array [1 .. 7] of Integer;
  p: ^Integer;

begin
  m := Succ (foo);     { bar }
  c := Succ ('A', 4);  { 'E' }
  a[1] := 42;
  a[2] := Succ (a[1]);     { 43 }
  a[5] := Succ (a[2], 7);  { 50 }
  {$X+}
  p := @a[1];
  p := Succ (p);     { points to `a[2]' now }
  p := Succ (p, 3);  { points to `a[5]' now }
end.
```

See also

[Pred], page 388, [Inc], page 338, Section 6.6 [Pointer Arithmetics], page 84.

Text

(Under construction.)

Synopsis

```
type
  Text  { built-in type }
```

Description

Conforming to

'Text' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program TextDemo;
var
  t: Text;
begin
  Rewrite (t, 'hello.txt');
  WriteLn (t, 'Hello, world!')
end.
```

See also

[file], page 322, [AnyFile], page 265.

then

(Under construction.)

Synopsis

Description

Part of an '`if`' statement or part of the '`and then`' operator.

Conforming to

'`then`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program ThenDemo;
var
  i: Integer;
begin
  Write ('Enter a number: ');
  ReadLn (i);
  if i > 42 then
    WriteLn ('The number is greater than 42')
end.
```

See also

Chapter 9 [Keywords], page 453.

Time

Synopsis

```
function Time (T: TimeStamp): packed array [1 .. Time_Length] of Char;
```

Description

Date takes a `TimeStamp` parameter and returns the time as a string (in the form of a packed array of `Char`). *Time_length* is an implementation defined invisible constant.

Conforming to

'`Time`' is an ISO 10206 Extended Pascal extension.

Example

Set [TimeStamp], page 431.

See also

[TimeStamp], page 431, [GetTimeStamp], page 331, [Date], page 303, Section 6.10.8 [Date And Time Routines], page 99.

TimeStamp

Synopsis

```
type
  TimeStamp = packed record
    DateValid,
    TimeValid  : Boolean;
    Year       : Integer;
    Month      : 1 .. 12;
    Day        : 1 .. 31;
    DayOfWeek  : 0 .. 6;   { 0 means Sunday }
    Hour       : 0 .. 23;
    Minute     : 0 .. 59;
    Second     : 0 .. 61;  { to allow for leap seconds }
    MicroSecond: 0 .. 999999;
    TimeZone   : Integer;  { in seconds east of UTC }
    DST        : Boolean;
    TZName1,
    TZName2    : String (32);
  end;
```

Description

The `TimeStamp` record holds all the information about a particular time. You can get the current time with `GetTimeStamp` and you can get the date or time in a printable form using the `Date` and `Time` functions.

Conforming to

'`TimeStamp`' is an ISO 10206 Extended Pascal extension. The fields '`DateValid`', '`TimeValid`', '`Year`', '`Month`', '`Day`', '`Hour`', '`Minute`', '`Second`' are required by Extended Pascal, the other ones are GNU Pascal extensions.

Example

```
program TimeStampDemo;

var
  t: TimeStamp;

begin
  GetTimeStamp (t);
  WriteLn ('DateValid: ', t.DateValid);
  WriteLn ('TimeValid: ', t.TimeValid);
  WriteLn ('Year: ', t.Year);
  WriteLn ('Month: ', t.Month);
  WriteLn ('Day: ', t.Day);
  WriteLn ('DayOfWeek (0 .. 6, 0=Sunday): ', t.DayOfWeek);
  WriteLn ('Hour (0 .. 23): ', t.Hour);
  WriteLn ('Minute (0 .. 59): ', t.Minute);
  WriteLn ('Second (0 .. 61): ', t.Second);
  WriteLn ('MicroSecond (0 .. 999999): ', t.MicroSecond);
  WriteLn ('TimeZone (in seconds east of UTC): ', t.TimeZone);
  WriteLn ('DST: ', t.DST);
  WriteLn ('TZName1: ', t.TZName1);
  WriteLn ('TZName2: ', t.TZName2);
  WriteLn;
  WriteLn ('Date is: ', Date (t));
  WriteLn ('Time is: ', Time (t));
end.
```

See also

[GetTimeStamp], page 331, [Date], page 303, [Time], page 430, Section 6.10.8 [Date And Time Routines], page 99.

to

(Under construction.)

Synopsis

Description

Part of a '`for`' loop counting upwards or a '`to begin do`' or '`to end do`' module constructor or destructor.

Conforming to

'`to`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453.

to begin do

(Under construction.)

Synopsis

Description

Conforming to

'`to begin do`' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

to end do

(Under construction.)

Synopsis

Description

Conforming to

'`to begin end`' is an ISO 10206 Extended Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Trim

(Under construction.)

Synopsis

```
function Trim (S: String): String;
```

Description

Conforming to

'Trim' is an ISO 10206 Extended Pascal extension.

Example

See also

True

Synopsis

```
type
  Boolean = (False, True);  { built-in type }
```

Description

'True' is one of the two Boolean values and is used to represent a condition which is always fullfilled. For example, the expression `1 = 1` always yields the value 'True'. It is the opposite of 'False'. 'True' has the ordinal value 1.

Conforming to

'True' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program TrueDemo;

var
  a: Boolean;

begin
  a := 1 = 1;  { yields True }
  WriteLn (Ord (True));  { 1 }
  WriteLn (a);  { True }
  if True then WriteLn ('This is executed.')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [False], page 320, [Boolean], page 280.

Trunc

Synopsis

```
function Trunc (x: Real): Integer;
```

Description

'Trunc' returns the integer part of a floating point number as an integer. Use 'Int' to get the integer part as a floating point number.

Conforming to

'Trunc' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program TruncDemo;

begin
  WriteLn (Frac (12.345) : 1 : 5);  { 0.34500 }
  WriteLn (Int (12.345) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.345) : 1);  { 12 }
  WriteLn (Trunc (12.345) : 1);  { 12 }

  WriteLn (Frac (-12.345) : 1 : 5);  { -0.34500 }
  WriteLn (Int (-12.345) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.345) : 1);  { -12 }
  WriteLn (Trunc (-12.345) : 1);  { -12 }

  WriteLn (Frac (12.543) : 1 : 5);  { 0.54300 }
  WriteLn (Int (12.543) : 1 : 5);  { 12.00000 }
  WriteLn (Round (12.543) : 1);  { 13 }
  WriteLn (Trunc (12.543) : 1);  { 12 }

  WriteLn (Frac (-12.543) : 1 : 5);  { -0.54300 }
  WriteLn (Int (-12.543) : 1 : 5);  { -12.00000 }
  WriteLn (Round (-12.543) : 1);  { -13 }
  WriteLn (Trunc (-12.543) : 1);  { -12 }
end.
```

See also

Section 6.2.4 [Real Types], page 68, [Real], page 398, [Int], page 342, [Frac], page 327, [Round], page 407.

Truncate

(Under construction.)

Synopsis

```
procedure Truncate (var F: any_file);
```

Description

Conforming to

'Truncate' is a Borland Pascal extension.

Example

See also

type

Synopsis

As a type declaration:

```
type
  type_indentifier = type_definition;
```

or with initialization:

```
type
  type_indentifier = type_definition value constant_expression;
```

Description

The reserved word 'type' starts the declaration of a *type identifier* which is defined by *type_definition*. For further description see Section 6.1.4 [Type Declaration], page 50, Section 6.1.4 [Type Declaration], page 50, Section 6.2.1 [Type Definition], page 64, Section 6.2 [Data Types], page 64.

Conforming to

'type' is defined in ISO 7185 Pascal and supported by all known Pascal variants. Initializers are an ISO 10206 Extended Pascal extension.

Example

```
program TypeDemo;
type
  { This side is the }    { That side is the }
  { type declaration }    { type definition  }

  { array type }
  ArrayType                  = array [0 .. 9] of Integer;

  { record type }
  RecordType                 = record
                                 Bar: Integer
                               end;

  { subrange type }
```

```
    SubrangeType              = -123 .. 456;

    { enumeration type }
    EnumeratedType            = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

    { set type }
    CharSetType               = set of Char;

    { object type }
    ObjectType                = object
                                  constructor Init;
                                  procedure Method;
                                  destructor Done
                                end;

    { pointer type to another type identifier }
    PArrayType                = ^ArrayType;

    { an alias name for another type identifier }
    IntegerType               = Integer;

    { an integer which is initialized by 123 }
    InitializedInt            = Integer value 123;

    { a schema with discriminants x and y of type Integer }
    SchemaType (x, y: Integer) = array [x .. y] of Integer;

  { Dummy methods of the object type }
  constructor ObjectType.Init;
  begin
  end;

  procedure ObjectType.Method;
  begin
  end;

  destructor ObjectType.Done;
  begin
  end;

  begin
  end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.4 [Type Declaration], page 50, Section 6.2.1 [Type Definition], page 64, Section 6.2 [Data Types], page 64, Section 6.1.5 [Variable Declaration], page 51, [array], page 270, [record], page 399, [object], page 373, [set], page 412, [Pointer], page 386, [value], page 443.

type of

(Under construction.)

Synopsis

Description

Conforming to

Example

See also

Chapter 9 [Keywords], page 453.

TypeOf

Synopsis

```
function TypeOf (var x): PObjectType;
```

Description

Returns a pointer to the VMT of an *object* type or variable. This pointer can be used to identify the type of an object.

'TypeOf' can be applied to expressions of object type and to object type names. In the former case, the actual type of polymorphic objects is returned.

Conforming to

'TypeOf' is a Borland Pascal extension.

Example

```
program TypeOfDemo;
type
  FooPtr = ^Foo;
  BarPtr = ^Bar;

  Foo = object          { Has a VMT, though it doesn't }
    x: Integer;         { contain virtual methods.     }
    constructor Init;
  end;

  Bar = object (Foo)
    y: Integer;
  end;
```

```
constructor Foo.Init;
begin
end;

var
  MyFoo: FooPtr;

begin
  MyFoo := New (BarPtr, Init);
  if TypeOf (MyFoo^) = TypeOf (Bar) then  { True }
    WriteLn ('OK')
end.
```

See also

[BitSizeOf], page 278, [AlignOf], page 261, [PObjectType], page 385, [SetType], page 414, [SizeOf], page 421, Section 6.8 [OOP], page 86.

Unbind

(Under construction.)

Synopsis

```
procedure Unbind (var F: any_file);
```

Description

Conforming to

'Unbind' is an ISO 10206 Extended Pascal extension.

Example

See also

[Bind], page 276, [Binding], page 277, [BindingType], page 277, [bindable], page 276.

unit

(Under construction.)

Synopsis

Description

UCSD and BP style unit declaration.

Conforming to

'unit' is a UCSD Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Unpack

(Under construction.)

Synopsis

```
procedure Unpack (Source: packed_array;
                  var Dest: unpacked_array;
                  FirstElement: index_type);
```

Description

Conforming to

'Unpack' is defined in ISO 7185 Pascal and supported by all known Pascal variants except UCSD/Borland Pascal and its variants.

Example

See also

until

(Under construction.)

Synopsis

Description

'until' is part of the 'repeat ... until' loop statement.

Conforming to

'until' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

Chapter 9 [Keywords], page 453, [repeat], page 402, [while], page 447, [for], page 325.

UpCase

(Under construction.)

Synopsis

```
function UpCase (Ch: Char): Char;
```

Description

Conforming to

'`UpCase`' is a Borland Pascal extension.

Example

See also

Update

(Under construction.)

Synopsis

```
procedure Update (var F: any_file);
```

Description

Conforming to

'`Update`' is an ISO 10206 Extended Pascal extension.

Example

See also

uses

Synopsis

In a program:

```
program @@fragment foo;

uses
  bar1,
  bar2 in 'baz.pas',
  bar3;
```

```
  [...]
```

In a unit:

```
unit @@fragment Bar3;

interface

uses
  bar1,
  bar2 in 'baz.pas';

[...]

implementation

uses
  bar3,
  bar4 in 'qux.pas';

[...]
```

Description

The reserved word '`uses`' in the *import part* of a program or unit makes the program or unit import an interface.

The keyword '`in`' tells GPC to look for the '`unit`' in the specified file; otherwise the file name is derived from the name of the interface, coverted to lower-case, by adding first '`.p`', then '`.pas`'.

In BP, there must be at most one '`uses`' part in a program, and in a unit, there can be one import part in the interface part and one in the implementation part. GPC, as an extention, allows for several '`uses`' parts, and also allows '`uses`' parts in a module.

The imported interface needn't be a UCSD/Borland Pascal unit, it may be an interface exported by an Extended Pascal module as well.

Conforming to

ISO Pascal does not define '`uses`' and units at all. UCSD and Borland Pascal do, but without the '`in`' extension. Delphi supports '`uses`' like described above.

Example

See also

Chapter 9 [Keywords], page 453, [unit], page 439, [module], page 365, [import], page 336.

Val

(Under construction.)

Synopsis

```
procedure Val (const Source: String; var x: integer_or_real;
               var ErrorPosition: Integer);
```

Description

'Val' converts the integer or real number that is represented by the characters in the string 'Source' and places it into 'x'.

The 'Source' string can have a base prefix ('$' for hexadecimal or 'Base#'). The optional 'ErrorCode' will be set to the position of the first invalid character, or to a 0 if the entire string represents a valid number. In case an invalid character occurrs in 'Source', 'x' will be undefined.

Conforming to

'Val' is a Borland Pascal extension.

Example

```
program ValDemo;
var
  x, ec: Integer;
  l: LongInt;
  r: Real;
begin
  Val ('123', x, ec);              { x :=            123; ec := 0; }
  Val ('-123', x, ec);             { x :=           -123; ec := 0; }
  Val ('123.456', r, ec);          { r :=        123.456; ec := 0; }
  Val ('$ffff', x, ec);            { x :=          65535; ec := 0; }
  Val ('$F000', x, ec);            { x :=          61440; ec := 0; }
  Val ('-$ffff', x, ec);           { x :=         -65535; ec := 0; }
  Val ('12#100', x, ec);           { x :=            144; ec := 0; }
  Val ('-2#11111111', x, ec);      { x :=           -255; ec := 0; }
  { here we have the invalid character 'X' for base 16 }
  Val ('$fffeX', x, ec);           { x :=    <undefined>; ec := 6; }
  Val ('12#100invalid', x, ec);    { x :=    <undefined>; ec := 7; }
  Val ('36#Jerusalem', l, ec);     { l := 54758821170910; ec := 0; }
end.
```

See also

[ReadLn], page 398, [ReadStr], page 398, [WriteLn], page 450, [WriteStr], page 450, [Str], page 426.

value

Synopsis

Description

The reserved word 'value' is part of a type or var declaration. It can be replaced by ':=' or '='.

Conforming to

'`value`' is an ISO 10206 Extended Pascal extension. '`:=`' in this context is a VAX Pascal extension, and '`=`' is a Borland Delphi extension.

Example

See also

Chapter 9 [Keywords], page 453, [var], page 444, [type], page 436, Section 6.1.5 [Variable Declaration], page 51, Section 6.1.4 [Type Declaration], page 50.

var

Synopsis

As part of a variable declaration part or in a statement block:

`var` *variable_identifier*`:` *type_identifier*`;`

or

`var` *variable_identifier*`:` *type_definition*`;`

and with initializing value:

`var` *variable_identifier*`:` *type_identifier* `value` *constant_expression*`;`

or

`var` *variable_identifier*`:` *type_definition* `value` *constant_expression*`;`

As part of a parameter list (passing by reference):

`var` *var_parameter*`:` *type_identifier*`;`

or without type declaration

`var` *var_parameter*`;`

or protected (i.e., the called routine can't modify the parameter):

`protected var` *var_parameter*`:` *type_identifier*`;`

or without type declaration

`protected var` *var_parameter*`;`

Description

In a declaration part: The reserved word '`var`' declares a *variable_identifier* whose type is of *type_identifier* or which is defined by *type_definition*. For further description see Section 6.1.5 [Variable Declaration], page 51, Section 6.1.4 [Type Declaration], page 50, Section 6.2.1 [Type Definition], page 64, Section 6.2 [Data Types], page 64.

In a parameter list: see Section 6.1.6.4 [Subroutine Parameter List Declaration], page 53.

Conforming to

'`var`' is defined in ISO 7185 Pascal and supported by all known Pascal variants. Untyped '`var`' parameters in parameter lists are a UCSD Pascal extension. The ability to do '`var`' declarations in a statement block is a GNU Pascal extension.

Example

```
program VarDemo;

type
  FooType = Integer;

var
  Bar: FooType;
  ArrayFoo: array [0 .. 9] of Integer;   { array var definition }
  FecordFoo: record                      { record var definition }
               Bar: Integer
             end;
  CharsetFoo: set of Char;               { set var }
  SubrangeFoo: -123 .. 456;              { subrange var }
  EnumeratedFoo: (Mon, Tue, Wed, Thu, Fri, Sat, Sun);  {enumerated var }
  PointerBar: ^FooType;                  { pointer var }

procedure ReadFoo (var Foo: FooType);
begin
  ReadLn (Foo)
end;

begin
  var Bar: Integer;  { GNU Pascal extension }
  Bar := 42
end.
```

See also

Chapter 9 [Keywords], page 453, [type], page 436, [array], page 270, [record], page 399, [set], page 412, Section 6.2.11.1 [Subrange Types], page 70, [Pointer], page 386, [protected], page 391.

view

Not yet implemented.

Synopsis

Description

Object class view.

Conforming to

'`view`' is an Object Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

virtual

(Under construction.)

Synopsis

Description

Virtual object method declaration.

Conforming to

'virtual' is an Object Pascal and a Borland Pascal extension.

Example

See also

Chapter 9 [Keywords], page 453.

Void

(Under construction.)

Synopsis

```
type
  Void  { built-in type }
```

Description

Conforming to

'Void' is a GNU Pascal extension.

Example

```
program VoidDemo;

procedure p (var x: Void);
begin
end;

var
  i: Integer;
  s: String (42);

begin
  p (i);
  p (s)
end.
```

See also

while

Synopsis

```
while boolean_expression do
  statement
```

Description

The '`while`' statement declares a loop. For further description see Section 6.1.7.6 [while Statement], page 58.

Conforming to

'`while`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

```
program WhileDemo;
var
  Foo, Bar: Integer;
begin
  WriteLn ('Enter an descending series of integer numbers.');
  WriteLn ('Terminate by breaking this rule.');
  WriteLn ('Enter start number: ');
  Bar := MaxInt;
  ReadLn (Foo);
  while Foo < Bar do
    begin
      Bar := Foo;
      ReadLn (Foo)
    end;
  WriteLn ('The last number of your series was: ', Bar)
end.
```

See also

Chapter 9 [Keywords], page 453, Section 6.1.7.7 [repeat Statement], page 59, Section 6.1.7.5 [for Statement], page 57.

with

(Under construction.)

Synopsis

Description

Automatic '`record`' or object field access.

Conforming to

'`with`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

...

{ Note bar is shadowed by foo.bar } ...

See also

Chapter 9 [Keywords], page 453.

Word

Synopsis

```
type
  Word = Cardinal;
```

Description

'`Word`' is the "natural" unsigned integer type in GNU Pascal. On some platforms it is 32 bits wide and thus has a range of '`0 .. 4294967295`'. It is the same as [Cardinal], page 285, introduced for compatibility with other Pascal compilers.

There are lots of other integer types in GPC, see Section 6.2.3 [Integer Types], page 64.

Conforming to

'`Word`' is defined in Borland Pascal and Borland Delphi, where it is a 16-bit unsigned integer type.

Example

```
program WordDemo;
var
  a: Word;
begin
  a := 42;
  WriteLn (a)
end.
```

See also

[Cardinal], page 285, Section 6.2.3 [Integer Types], page 64, Section 6.2.11.1 [Subrange Types], page 70.

WordBool

Synopsis

```
type
  WordBool = Boolean attribute (Size = BitSizeOf (Word));
```

Description

The intrinsic '`WordBool`' represents boolean values, but occupies the same memory space as a '`Word`'. It is used when you need to define a parameter or record that conforms to some external library or system specification.

Conforming to

'`WordBool`' is a Borland Pascal extension.

Example

```
program WordBoolDemo;
var
  a: WordBool;
begin
  Word (a) := 1;
  if a then WriteLn ('Ord (True) = 1')
end.
```

See also

Section 6.2.9 [Boolean (Intrinsic)], page 69, [Boolean], page 280, [True], page 434, [False], page 320, [CBoolean], page 287, [ByteBool], page 282, [ShortBool], page 416, [MedBool], page 360, [LongBool], page 350, [LongestBool], page 351.

Write

(Under construction.)

Synopsis

`procedure Write (var F:` *typed_file*`;` *variable*`);`

or

`procedure Write (var F: Text;` *values_and_format_specifications*`);`

or

`procedure Write (`*values_and_format_specifications*`);`

Description

Conforming to

'`Write`' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

WriteLn

(Under construction.)

Synopsis

```
procedure WriteLn (var F: Text; values_and_format_specifications);
```

or

```
procedure WriteLn (values_and_format_specifications);
```

Description

Conforming to

'WriteLn' is defined in ISO 7185 Pascal and supported by all known Pascal variants.

Example

See also

WriteStr

(Under construction.)

Synopsis

```
procedure WriteStr (var Dest: String; values_and_format_specifications);
```

Description

Conforming to

'WriteStr' is an ISO 10206 Extended Pascal extension.

Example

See also

xor

Synopsis

```
operator xor (operand1, operand2: Boolean) = Result: Boolean;
```

or

```
operator xor (operand1, operand2: integer_type) = Result: integer_type;
```

or

```
procedure xor (var operand1: integer_type; operand2: integer_type);
```

Description

In GNU Pascal, '`xor`' has three built-in meanings:

1. Logical "exclusive or" between two '`Boolean`'-type expressions. The result of the operation is of '`Boolean`' type. (Logical '`foo xor bar`' in fact has the same effect as '`foo <> bar`'.)
2. Bitwise "exclusive or" between two integer-type expressions. The result is of the common integer type of both expressions.
3. Use as a "procedure": '`operand1`' is "xor"ed bitwise with '`operand2`'; the result is stored in '`operand1`'.

Conforming to

ISO Pascal does not define the '`xor`' operator; Borland Pascal and Delphi do.

Use of '`xor`' as a "procedure" is a GNU Pascal extension.

Example

```
program XorDemo;
var
  a, b, c: Integer;
begin
  if (a = 0) xor (b = 0) then
    c := 1  { happens if either 'a' or 'b' is zero,    }
            { but not if both are zero or both nonzero }
  else if (a xor b) = 0 then  { bitwise xor }
    c := 2  { happens if a = b }
  else
    xor (c, a)  { same as 'c := c xor a' }
end.
```

See also

Chapter 9 [Keywords], page 453, [and], page 262, [or], page 375, Section 6.3 [Operators], page 82.

9 Pascal keywords and operators supported by GNU Pascal.

This chapter lists all keywords understood by GNU Pascal. The keywords are taken from the following standards and dialects:

- ISO 7185 Pascal (CP)
- ISO 10206 Extended Pascal (EP)
- ANSI draft Object Pascal (OP)
- UCSD Pascal (UCSD)
- Borland Pascal 7.0 (BP)
- Borland Delphi (BD)
- Pascal-SC (PXSC, Pascal eXtensions for Scientific Calculations)
- VAX Pascal (VP)
- Sun Pascal (SP)
- Traditional Macintosh Pascal (MP)
- GNU Pascal extensions (GPC)

The table below lists all known keywords with short descriptions. The links point to the longer descriptions in the reference.

By default (using GPC extensions) all keywords are enabled. Only those are marked '`GPC`' in the table below that are valid *only* in the GPC extensions.

All keywords that are specific to some dialects (i.e., not marked "any") can also be used as identifiers (with a few exceptions, see below). Then, however, they generally cannot be used as keywords anymore. So you can compile code from dialects that use them as keywords and code that uses them as identifiers, i.e., you do not have to modify your correct ISO 7185 programs in order to compile them with GPC without any dialect option. Some words can even be used as keywords and identifiers in parallel, including '`forward`' (according to ISO 7185 Pascal), and '`near`' and '`far`' (according to Borland Pascal).

The exceptions are:

- '`Operator`' can't be used as a type, untyped constant or exported interface, i.e. when it would be followed by '`=`' (unless it's disabled as a keyword explicitly or by dialect options, see below). This is because of a conflict with a definition of the operator '`=`'. (It can be used as a typed constant, but it might be confusing if you later decide to make it untyped, so use with care.)
- The first statement after '`initialization`' (Delphi specific unit initialization) must not start with '`(`'. (Statements starting with '`(`' are uncommon, anyway, but not impossible.) This does not restrict the usage of '`Initialization`' as an identifier.
- The following keywords can't be used *immediately* after an '`import`' part: '`uses`', '`implementation`', '`operator`', '`constructor`', '`destructor`'. Using '`uses`' instead of '`import`', or putting some other declaration between '`import`' and the problematic keyword helps.

A dialect option turns off all keywords that do not belong to this dialect. Besides, any keyword can be enabled and disabled with the compiler options '`{$enable-keyword}`' and '`{$disable-keyword}`'.

absolute (BP, BD) (see [absolute], page 258)
: overloaded variable declaration

abstract (OP) (see [abstract], page 260)
: abstract object type or method declaration

all (all) (see [all], page 261)
: '`export`' (see [export], page 317) extension ('`export foo = all`')

and (any) (see [and], page 262)
: Boolean or bitwise '`and`' operator or part of the '`and then`' (see [and then], page 263) operator

and_then (EP, OP) (see [and_then], page 264)
: short-circuit Boolean '`and`' (see [and], page 262) operator

array (any) (see [array], page 270)
: array type declaration

as (OP, BD) (see [as], page 271)
: object type membership test and conversion

asm (BP, BD) (see [asm], page 271)
: GNU style inline assembler code

asmname (GPC) (see [asmname], page 271)
: DEPRECATED! linker name of routines and variables

attribute (GPC) (see [attribute], page 274)
: attributes of routines and variables

begin (any) (see [begin], page 275)
: begin of a code block, part of a '`to begin do`' (see [to begin do], page 433) module constructor

bindable (EP, OP) (see [bindable], page 276)
: external bindability of files

c (GPC) (see [c], page 284)
: DEPRECATED! declaration of external routine

case (any) (see [case], page 286)
: multi-branch conditional statement or variant '`record`' (see [record], page 399) type

c_language (GPC) (see [c_language], page 291)
: DEPRECATED! declaration of external routine

class (OP, BD) (see [class], page 291)
: OOE/Delphi style object class (not yet implemented)

const (any) (see [const], page 296)
: constant declaration or constant parameter declaration

constructor (OP, BP, BD) (see [constructor], page 297)
: object constructor

destructor (OP, BP, BD) (see [destructor], page 306)
: object destructor

div (any) (see [div], page 307)
: integer division operator

do (any) (see [do], page 308)
: part of a '`while`' (see [while], page 447) or '`for`' (see [for], page 325) loop, a '`with`' (see [with], page 447) statement, or a '`to begin do`' (see [to begin do], page 433) or '`to end do`' (see [to end do], page 433) module constructor or destructor

downto (any) (see [downto], page 309)
: part of a '`for`' (see [for], page 325) loop counting downwards

else (any) (see [else], page 310)
: alternative part of an '`if`' (see [if], page 334) statement, default '`case`' (see [case], page 286) branch, part of the '`or else`' (see [or else], page 377) operator

end (any) (see [end], page 311)
: end of a code block, end of a '`case`' (see [case], page 286) statement, end of a '`record`' (see [record], page 399) or '`object`' (see [object], page 373) declaration, part of a '`to end do`' (see [to end do], page 433) module destructor

export (export) (see [export], page 317)
: module interface export

exports (BP, BD) (see [exports], page 318)
: library export (not yet implemented)

external (UCSD, BP, BD, MP) (see [external], page 320)
: declaration of an external object

far (BP, BD) (see [far], page 321)
: BP directive (ignored)

file (any) (see [file], page 322)
: non-text file type declaration

finalization (BD) (see [finalization], page 324)
: unit finalization

for (any) (see [for], page 325)
: loop statement where the number of loops is known in advance

forward (any) (see [forward], page 326)
: declaration of a routine whose definition follows below

function (any) (see [function], page 329)
: function declaration

goto (any) (see [goto], page 331)
: statement to jump to a '`label`' (see [label], page 345)

if (any) (see [if], page 334)
: conditional statement

implementation (all except CP) (see [implementation], page 336)
: module or unit implementation part

import (EP, OP) (see [import], page 336)
: module interface import

in (any) (see [in], page 337)
: set membership test or part of a '`for`' (see [for], page 325) loop iterating through sets

inherited (OP, BP, BD, MP) (see [inherited], page 340)
: reference to methods of ancestor object types

initialization (BD) (see [initialization], page 340)
: unit initialization

interface (interface) (see [interface], page 344)
: module or unit interface part

interrupt (BP, BD) (see [interrupt], page 344)
: interrupt handler declaration (not yet implemented)

is (OP, BD) (see [is], page 345)
: object type membership test

label (any) (see [label], page 345)
: label declaration for a '`goto`' (see [goto], page 331) statement

library (BP, BD) (see [library], page 348)
: library declaration (not yet implemented)

mod (any) (see [mod], page 364)
: integer remainder operator

module (module) (see [module], page 365)
: EP style or PXSC style module

name (name) (see [name], page 366)
: linker name

near (BP, BD) (see [near], page 368)
: BP directive (ignored)

nil (any) (see [nil], page 370)
: reserved value for unassigned pointers

not (any) (see [not], page 371)
: Boolean or bitwise negation operator

object (BP, BD, MP) (see [object], page 373)
: BP style object declaration

of (any) (see [of], page 374)
: part of an '`array`' (see [array], page 270), '`set`' (see [set], page 412) or typed '`file`' (see [file], page 322) type declaration, a '`case`' (see [case], page 286) statement, a variant '`record`' (see [record], page 399) type or a '`type of`' (see [type of], page 438) type inquiry

only (EP, OP) (see [only], page 375)
: import specification

operator (PXSC) (see [operator], page 375)
: operator declaration

or (any) (see [or], page 375)
: Boolean or bitwise '`or`' operator or part of the '`or else`' (see [or else], page 377) operator

or_else (EP, OP) (see [or_else], page 378)
: short-circuit Boolean '`or`' (see [or], page 375) operator

otherwise (EP, OP, MP) (see [otherwise], page 379)
: default '`case`' (see [case], page 286) branch, default value in a structured value of '`array`' (see [array], page 270) type

packed (any) (see [packed], page 381)
: declaration of packed structured types ('`record`' (see [record], page 399), '`array`' (see [array], page 270), '`set`' (see [set], page 412), '`file`' (see [file], page 322)), also packed ordinal subranges

pow (EP, OP) (see [pow], page 388)
: exponentiation operator with integer exponent

private (private) (see [private], page 389)
: private object fields

procedure (any) (see [procedure], page 390)
: procedure declaration

program (any) (see [program], page 390)
: start of a Pascal program

property (OP, BD) (see [property], page 391)
: object properties (not yet implemented)

protected (protected) (see [protected], page 391)
: read-only formal parameters or module export and protected object fields

public (public) (see [public], page 394)
: public object fields

published (published) (see [published], page 395)
: published object fields

qualified (EP, OP) (see [qualified], page 395)
: import specification

record (any) (see [record], page 399)
: record type declaration

repeat (any) (see [repeat], page 402)
: loop statement

resident (BP, BD) (see [resident], page 403)
: library export specification (not yet implemented)

restricted (EP, OP) (see [restricted], page 404)
: restricted type specification

segment (UCSD) (see [segment], page 411)
: segment specification (not yet implemented)

set (any) (see [set], page 412)
: set type declaration

shl (BP, BD, MP) (see [shl], page 415)
: bitwise left shift operator

shr (BP, BD, MP) (see [shr], page 419)
: bitwise right shift operator

then (any) (see [then], page 430)
: part of an '`if`' (see [if], page 334) statement or part of the '`and then`' (see [and then], page 263) operator

to (any) (see [to], page 432)
: part of a '`for`' (see [for], page 325) loop counting upwards or a '`to begin do`' (see [to begin do], page 433) or '`to end do`' (see [to end do], page 433) module constructor or destructor

type (any) (see [type], page 436)
: type declaration or part of a '`type of`' (see [type of], page 438) type inquiry

unit (UCSD, BP, BD, MP) (see [unit], page 439)
: UCSD and BP style unit declaration

until (any) (see [until], page 440)
: end of a '`repeat`' (see [repeat], page 402) statement

uses (UCSD, BP, BD, MP) (see [uses], page 441)
: unit import

value (EP, OP) (see [value], page 443)
: variable/type initializer

var (any) (see [var], page 444)
: variable declaration or reference parameter declaration

view (OP) (see [view], page 445)
: object class view (not yet implemented)

virtual (OP, BP, BD, MP) (see [virtual], page 446)
: virtual object method declaration

while (any) (see [while], page 447)
: loop statement

with (any) (see [with], page 447)
: automatic '`record`' (see [record], page 399) or object field access

xor (BP, BD, MP) (see [xor], page 451)
: Boolean or bitwise '`exclusive or`' operator

10 Where to get support for GNU Pascal; how to report bugs.

Here you can find information on where to find the most up-to-date information about GPC, how you can get support (professional or voluntary), how to use GPC's Test Suite, and how to report new bugs you might encounter.

If you have problems with GNU Pascal, please read the relevant sections of the GPC Manual. The most current version of the manual is available at `http://www.gnu-pascal.de/gpc/index.html` for online browsing. You can also download the complete manual in HTML format (tar.bz2) ('`gpc-html.tar.bz2`' on GPC's WWW home page). The manual is also available in PDF ('`gpc.pdf`' on GPC's WWW home page), PostScript (compressed with bzip2) ('`gpc.ps.bz2`' on GPC's WWW home page) and DVI (compressed with bzip2) ('`gpc.dvi.bz2`' on GPC's WWW home page) formats for printing or online reading.

If the manual doesn't help you, here is what we recommend you to do:

Chapter 3 [FAQ], page 15 Check the GPC FAQ

10.1 The GPC Mailing List

There is a mailing list devoted to GNU Pascal. You can write to the mailing list, e.g. if you have problems installing GPC or if you found a problem with GPC (please see Section 10.6 [Reporting Bugs], page 461). You can also use it to discuss suggestions for improving GPC and, most welcome, to offer your help or contribute code, documentation or other work. Mails to the list should be in English.

To subscribe to the mailing list, send the command

```
subscribe gpc your@email.address
```

in the body of a mail to `majordomo@gnu.de` (**not** to '`gpc@gnu.de`'!). The subject is ignored. (Please replace '`your@email.address`' with your real email address.) For more info, send a line '`help`' to `majordomo@gnu.de`.

After subscribing, you can send a message to the mailing list by sending email to the list address `gpc@gnu.de` as if it were a person.

To leave the mailing list, send the command

```
unsubscribe gpc your@email.address
```

to `majordomo@gnu.de`.

You can reach a human moderator at *gpc-owner@gnu.de*.

There is a separate mailing list for discussions about GPC documentation, `gpc-doc@gnu.de`. To subscribe, send the command

```
subscribe gpc-doc your@email.address
```

to `majordomo@gnu.de`.

There is also a (low-traffic) announce list, `gpc-announce@gnu.de` that you can subscribe to stay up-to-date. To subscribe to the list, write an email with

```
subscribe gpc-announce your@email.address
```

in the body to `majordomo@gnu.de`. If you like to announce a contribution, a service or an event related to GPC, you are invited to post to this list rather than '`gpc@gnu.de`', but please don't use the announce list for questions or discussions. Please note that all mail sent to the announce list is forwarded to the regular list, so you won't have to subscribe to both lists if you don't want to miss anything. For the same reason, please don't cross-post to both lists.

There is also a German speaking mailing list, `gpc-de@gnu.de`. To subscribe send the command

```
subscribe gpc-de your@email.address
```

in the body of a mail to `majordomo@gnu.de` (note the hints above).

10.2 The GPC Mailing List Archives

Perhaps your problem was already discussed on the list. There is a searchable archive of the mailing list on the WWW. It can be browsed or searched at

`http://www.gnu-pascal.de/crystal/gpc/en/`

The archive of the documentation list is at

`http://www.gnu-pascal.de/crystal/gpc-doc/en/`

The archive of the announce list is at

`http://www.gnu-pascal.de/crystal/gpc-announce/en/`

The archive of the German GPC list is at

`http://www.gnu-pascal.de/crystal/gpc-de/de/`

10.3 Newsgroups relevant to GPC

To get support, you can also ask the Usenet newsgroups for help. There are several Pascal related newsgroups, but none is dedicated just to GNU Pascal, so use the one which is most appropriate for your problem. For general Pascal questions, we recommend the following one:

`news://comp.lang.pascal.misc` Pascal in general and ungrouped Pascals.

Pascal syntax related questions may be appropriate in:

`news://comp.lang.pascal.ansi-iso` Pascal according to ANSI and ISO standards.

The next newsgroup is a haven for beginners, answering questions that would apply to almost any Pascal. However, if you have a GPC-specific question don't post there – use the GPC mailing list. And when in doubt use the GPC mailing list.

`news://comp.lang.pascal.borland` Borland Pascal questions.

Don't forget to give back what you have obtained. None of us is getting money for answering your questions (unless you pay us by yourself). Please do your part by answering the questions of others instead.

10.4 Where to get individual support for GPC

GPC is free software and comes **without any warranty**.

If you want to get professional support, you can hire an individual or a company for providing such a service.

G-N-U GmbH (`http://www.g-n-u.de`) is doing large parts of the development of GNU Pascal. This company offers special support for GPC and can be contacted at `info@g-n-u.de`.

More generally, to find a company or an individual who offers support and other consulting services for free software, look into the GNU Service Directory (`http://www.gnu.org/prep/service.html`).

10.5 If the compiler crashes . . .

If the compiler crashes, you have discovered a bug. A reliable compiler never crashes. To help the maintainers fix this bug, it is important that you send us a problem report.

If you're on Unix, you can find out where the compiler crashed if you enable coredumps, then load the compiler ('`gpc1`') plus the core file in the debugger ('`gdb /your_path_here/gpc1 core`'), then type '`backtrace`' to get a stacktrace. Include this stacktrace in your bug report.

10.6 How to report GPC bugs

If you encounter a bug with GPC, please check whether it is one of the known bugs(see Section 11.1 [Known Bugs], page 465). If not, please report it to the GNU Pascal mailing list(see Section 10.1 [Mailing List], page 459). That way, they always reach the maintainers. Please note the following points.

- Please send a description of the problem. Try to give as much information as possible, with the full text of any error messages encountered, or a description of how some output varies from the expected output. Always specify the operating system type with version and the machine type (try '`uname -a`' if unsure) as well as the version of GPC which you get by typing '`gpc -v`'.
- A good article on submitting bug reports can be found at either `http://www.chiark.greenend.org.uk/~sgtatham/bugs.html` or `http://freshmeat.net/news/2000/02/26/951627540.html`

  Another good article "How To Ask Questions The Smart Way" is available as `http://www.catb.org/~esr/faqs/smart-questions.html`

  Please note that the authors of these articles have no relation to GPC and will not help you with your problems! The articles contain general hints about how to report problems well.

If the problem is with the compiler itself, not an installation problem or something like this, please provide a test program to reproduce the problem, and note the following hints. You can also contribute test programs for features that are working in GPC to ensure they will not break in future releases.

- The test program should be as short as possible, but **by all means**, please send a **complete** program and **make sure** that it still reproduces the problem before you send it. Too often, users have sent code which contained obvious syntax errors far before the actual problem, or just code fragments that we could only make wild guesses about. This is unproductive for us and doesn't help you get your problem solved.

The preferred form for test programs is the form that the automated GPC Test Suite understands. Please, if at all possible, send your test programs in this form which should be easy to do, so we won't have to waste our time to bring them into this form, and can concentrate on fixing the problem.

- The file containing the main program *must* have a name ending with '`.pas`' and contain the keyword '`program`' (case-insensitively) and a '`;`' in the same line to be recognized by the Test Suite at all. Other files whose name ends in '`.pas`' (e.g., units or modules needed by the program), must not contain this.
- Since the Test Suite must run under very . . . nah . . . strange operating systems, too, file names must be distinguished in their first eight characters (case-insensitively) and should not contain anything but letters, numbers, hyphens, underscores and a single dot. Furthermore, any ancillary files (units, modules, includes, data files) should not be longer than "8+3" characters; the same applies to the names of unit/module interfaces (because GPC will create '`.gpi`' file names based on those). It is often a good idea to use your name, nickname or initials followed by a number as the file name.
- If your test program needs any units or modules, don't give them (or their interfaces in case of modules) common names like '`Test`', '`Foo`' or '`MyUnit`', unless you have very special reasons to, because there might be subtle problems if several test programs use the same name. Instead, it is recommended to prefix the unit/module/interface names with the name of the main test program or an abbreviation of it (if necessary because of the file name restrictions). Furthermore, please avoid the use of units and modules at all if the bug is not specific to them, in order to keep the test as simple as possible.

- The test program, when run, should produce a line of output consisting of the string 'OK' (followed by a newline) if everything went as expected, and something else (e.g. 'failed', possibly followed by the reason of failure) if something went wrong. In the latter case you might want to output additional information such as the values of important variables or an indication in which place the program failed if there are several possible places.
- However, if the program is intended to check whether GPC catches an (intentional) compile-time error in the program, place the string 'WRONG' somewhere in the test program, preferably in a comment in the line that contains the intentional error. 'WRONG' tests will be run with the option '-w' to suppress all warnings, so only real errors will be detected.

  Please note: While it is possible to combine several 'OK' tests in a single test program (if you make sure that it outputs 'OK' only if all tests pass), you cannot put several 'WRONG' tests into one test program. This is because the compiler will fail (and the test therefore be regarded as successful) already if *one* error occurs. So, for 'WRONG' tests, do only one check per test program. Also, try to keep such a test program as small and simple as possible, to avoid the risk that it will fail because of other problems (and therefore the test be mistakenly considered successful).
- If the test should merely provoke a GPC warning, use 'WARN' instead of 'WRONG'. This will run the test without '-w', but with '-Werror'. However, such tests will also appear to succeed if they produce a compiler error, not only a warning. Therefore, when checking for a warning, it is often a good idea to provide a complementary test (with expected success) and with '-w' in 'FLAG' or a compiler directive '{$W-}' to make sure that it's really just a warning, not an error.
- Runtime errors must be detected by the test itself. One way to do so is to insert code like the following into your test program:

```
uses GPC;

procedure ExpectError;
begin
  if ExitCode = 0 then
    WriteLn ('failed')
  else
    begin
      WriteLn ('OK');
      Halt (0)
    end
end;

begin
  AtExit (ExpectError);
  { Your code which should provoke a runtime error }
end.
```

- For a test that reproduces an existing problem (which is not expected to be fixed soon), please put a comment at the top that describes the problem in a few words, and start it with 'BUG'. This is not required by the test scripts, it's just to make it easier for those who will try to fix the problem to see immediately what the test is about. Tests for new (planned) features should not say 'BUG'.

The following special features of the Test Suite may be helpful for constructing slightly unusual tests:

- If the expected output is something else than 'OK', place it in a file '<basename>.out' (where '<basename>' is the name of the test program without the '.pas' extension).

- If the test program expects some input, place it in a file '`<basename>.in`'. It will automatically be redirected to the program's standard input.
- If the test needs some special flags to be given to the GPC command line, place them in a comment preceded by '`FLAG`', e.g.:

  ```
  { FLAG --extended-pascal -Werror }
  ```
- If the test program creates a file, use '`.dat`' as a file name suffix and no directory (the Makefiles will remove such files in the '`mostlyclean`' etc. targets) and do not assume that this file exists, does not exist, or anything else about it when the test starts. If possible, use an internal (unnamed) file, so these issues will not apply.
- The source file name of the test program will be passed as the first command-line argument to the test program run.
- If a test needs to be run in a special way, you can accompany the program with a script '`<basename>.run`' that will do the actual test after the test program was compiled. This script will be run by '`sh`' (regardless of its first line). In order to be portable, it should only use standard tools and features present in all '`sh`' compatible shells (e.g., '`ash`', '`bash`', but not necessarily '`csh`'). The source file name of the test program will be passed as the first command-line argument to the run script in this case. The compiled file is called '`./a.out`' on most systems, but, e.g., '`./a.exe`' on Cygwin. The environment variable '`A_OUT`' contains the base name (i.e., '`a.out`' or '`a.exe`', so you can always invoke the program as '`./"$A_OUT"`').
- If a test needs to be compiled in a special way (e.g., to decide whether to skip the test), place the commands in a script (preferably called '`<basename>.cmp`'), and put the file name of the script (without directory) in a comment preceded by '`COMPILE-CMD:`' into the source of the test program. The compile script will be run **instead** of the compiler and any other action otherwise done for this test program, so it gives you maximum flexibility to do whatever you need to do. This script will be run by '`sh`' (regardless of its first line). In order to be portable, it should only use standard tools and features present in all '`sh`' compatible shells (see above). The first command-line argument to the compile script will be the compiler to use, including all options. The second argument will be the source file name of the test program. For some typical tests, there are standard compile scripts, e.g. '`asmtest.cmp`' which will skip the test unless run on a platform supported by the few tests that contain 'asm' statements. Of course, it's generally better not to have to use such scripts when possible.
- In some cases you may want to write *randomized* tests. This is not usually recommended since it makes problems found harder to reproduce, but sometimes it might be useful (e.g., if you want to cover a large parameter space). In such a case, the following strategy can be used:

  ```
  ...

  uses GPC;

  var
    RandomSeed: Integer;

  begin
    RandomSeed := Random (MaxInt);
    SeedRandom (RandomSeed);

    ... { do your normal tests }

    { when printing an error message: }
  ```

```
    if ... then
      begin
        WriteLn ('failed (', RandomSeed, ') ',
                 ... { possibly additional information } );
        Halt
      end
  end.
```

This is a little unnatural since a random number is used to (re-)seed the random number generator, but there's currently no way to retrieve the internal state of the random number generator (and in fact, it's not represented by a single number, but by a large array).

Given the value of '`RandomSeed`' in an error message, it should then be possible to reproduce the problem by inserting this value in place of the '`Random (MaxInt)`'. Just be sure to print this value in **every** message of failure the program may produce.

10.7 Running the GPC Test Suite

Note: If you have installed a GPC binary distribution, you usually don't have the Test Suite installed (you can download the GPC source distribution to get it, however), so this section does not apply to you. Still, you can find in the section '`Contributing Tests to the Test Suite`' how to report GPC bugs in the form of new test programs so we can fix them as fast as possible.

The files in the test directory and subdirectories are for testing GPC only and should not be of any other use.

Note: A few of the tests do not make sense on all systems. They are equipped with checks and will be skipped if they find the system not suitable. Skipped tests do **not** indicate a GPC bug, unless you have a reason to be sure that the particular test should make sense on your system.

- To run the whole Test Suite, type '`make`' in the test directory (or '`make check`' in the '`p`' object directory after building GPC).
- The output will show all errors encountered (hopefully none) and tests skipped, and finally display a summary giving the number of successful, failed and skipped tests. Any failed test indicates a bug in GPC which should be reported to the GPC mailing list, `gpc@gnu.de`. Don't forget to mention the operating system you ran the test on and any other relevant information about what you did.
- You can also type '`make pascal.check-long`' to get a long output which is a sequence of file names followed by '`OK`' for successful tests, '`SKIPPED`' for skipped tests (both in capital letters), and anything else for failed tests.
- To run only some of the tests, you can type something like '`make MASK="foo[1-7]*.pas"`' or '`make MASK="foo42.pas" pascal.check-long`'.
- To clean up after running the tests, type '`make clean`'.

11 The GNU Pascal To-Do List.

This is the To-Do list for the GNU Pascal Compiler.

The GNU Pascal development team is constantly working on making GNU Pascal more reliable and more comfortable. However, there are some known bugs which will take some time to be fixed (any help welcome!), and we do not want to hide them from you. You might also be interested to know what features are planned to be implemented in the future. This list is meant to keep track of the known bugs and wanted features of GPC.

If you want to report a new bug or suggest a new feature, the best way to do it is in the mailing list. This way, other GPC users can provide additional ideas, perhaps work-arounds for bugs, and the GPC maintainers can sooner start to think about how to solve the problem. The GPC mailing list is `gpc@gnu.de`. To subscribe, send the command '`subscribe gpc your@email.address`' in the body of a mail to `majordomo@gnu.de` (the subject is ignored). An archive of the mailing list can be found at `http://www.gnu-pascal.de/crystal/gpc/en/`.

The list changes regularly. The present version refers to the current GPC snapshot, 20050331.

This list is part of the GNU Pascal Manual, included in GPC distributions and snapshots. You can always browse the most current version of the list on GPC's WWW page. If you check the To-Do list regularly you can decide if and when to try a new snapshort.

11.1 Known bugs in GPC

In this section, you can find information about GPC bugs that are known to the developers.

If you encounter a bug with GPC, please check whether it is one of the known bugs. If not, report it to the GNU Pascal mailing list. (But always report if you solve the problem! :–)

Solved problems are moved to "Fixed Bugs" (see Section 11.3 [Fixed Bugs], page 468), and implemented features to the "News" chapter (see Chapter 2 [News], page 9).

A message ID (like '`<42@foo.bar>`') refers to a message in the GPC mailing list or a newsgroup where the topic was discussed (often a bug report). A note of the form '`(xy20001231)`' refers to an internal message of the GPC developers. A file name like '`(foo42.pas)`' refers to the according program in the GPC test suite included in the GPC source distribution.

- '`setlimit`' is applied when it shouldn't be ((a) variable/constant declarations, (b) '`set of ShortCard`') ???
- dynamic sets don't work
- there are some bugs with mathematical functions; GPC fails, for example, the Paranoia test (`http://www.netlib.org/paranoia/index.html`)

11.2 Features planned for GPC

In the following sections, you can find informations about features which have been suggested and/or discussed for inclusion into GPC, usually in the GPC mailing list. The division into the sections and the names of the sections may change when this seems useful. The entries within each section are in no particular order.

Some of the entries may be a bit difficult to read. Double question marks ('`??`') within the entries mean that it's not yet clear how to do the thing. Triple question marks ('`???`') mean that it's not clear whether to do it at all. ;–)

11.2.1 Planned features: Strings

- const/var '`AnyString`' parameters and pointers (records internally, cf. gpc.pas) (GetCapacity; only for var parameters)
- '`SetLength`' for non-Pascal string types; use truncate-flag
- make work on all string types: string functions from '`rts/string*.pas`', '`Concat`'/'`+`' (Optimize '`s := s + ch`') (fh19971105)
- option '`--arrays-of-char-as-cstrings={none,zero-based,all}`' to allow assigning [zero-based] arrays of char to cstring variables (warning otherwise) (cstrini2.pas); make padding of those arrays in initialized variables and in assignments dependent on this switch (fh19970921) (fjf57.pas, ok) and if the length matches exactly, give a warning (fh19971013) (cstrassign.pas); pass those arrays as cstrings in read*, write*, str (P_*STRING; current length=-1 in two places); compile time warning (and treatment as non-zero-based) if it's not clear at compile time if a conformant array or schema is zero-based or not, and an operation depends on this in the zero-based state
- Short strings (Length, Capacity) and switches (tc19980326.2,tc19980327); allow '`type x=string`' (undiscriminated) when switch is set so that strings don't default to length 255 (fh19971113) (cf. fjf97.pas); don't add a #0 terminator when passing as a CString parameter (automatically use String2CString)
- automatically convert CStrings to Strings?
- general switch '`--bp-strings`'
- "wide" characters
- open array/string parameters ('`$P`', '`$V`' in BP; undocumented '`OpenString`' identifier in BP)
- variables of type undiscriminated schema/strings -> remove pointers to string constants; functions returning undiscriminated schemata (pg19980813.3)

11.2.2 Planned features: OOP

- Object Oriented Extensions draft, Delphi; '`class`'; allow classes and object types to inherit from each other; <01BD7A3A.6B187A20.prucha@helicon.co.at>; '`obpascal.zip`'
- error/exception handling (Java, Delphi?) <01BD7A3A.6B187A20.prucha@helicon.co.at> (tc20000623)
- method pointers/procedural variables ('`procedure/function of object`') (-> Delphi) ???
- destructor: reset VMT link to nil ???
- '*class* `is` *class*' (implemented in Pascal with '`Parent`')
- BP compatible dynamic method dispatcher ?? ???
- '`New`': Delphi syntax; Dispose?? (automatically?; set references automatically to nil) (-> FPC)
- '`--base-class=foo`', '`--base-object=bar`' ???
- properties <01BD7A3A.6B187A20.prucha@helicon.co.at>, <16131.199805071354@potter.cc.keele.ac.uk>
- VMTs (also for local classes (EP modules???)!) in GPI files, written to assembler file by main program; '`--cfiles=foo`': .s file with VMTs for C/assembler??? (also .h???)
- method overloading ??? ??
- interfaces (cf. OOE, Java); '*obj* `is` *interface*' (not so easy?) ???; variables of type pointer to more than one interface [and class] (also possible!); VMT problem with interfaces (my solution??? also with dynamic linking???)
- virtual constructors; in interfaces (load)???

- VMT: Children: list of pVMT
- VMT: object constants, class variables; virtual/static
- VMT: ClassID (64/128 bits?) default 0, error when duplicated, warning when 0 and inherited from class with `<>`0 ?, not allowed for abstract classes!; ClassName (also for abstract classes?); []; ProgrammerID, BaseClassID at beginning of unit (redeclarable?); find class by ClassID/ClassName (implementable in Pascal with '`Children`'); store in GPI: whether ClassID used, ClassName used, which classes are instantiated ???

11.2.3 Planned features: Other types

- variant records: EP: check that all values of tag-type have exactly one matching variant
- automatically detected upper array bounds in structured constants/intialized variables (-> PXSC)
- UCSD Pascal's '`Integer[n]`' ??? BCD?
- '`SELECTED_REAL_KIND`' for real types (cf. Fortran, drf) ?? ???
- general subtypes ???
- type-cast operators ???
- read/write operators ???
- read Booleans and enum types from files, write enum types to files ???
- sparse sets; sets of arbitrary types ?? ???
- initalization and finalization code for arbitrary types (e.g. GMP types)
- resize schemata (cf. '`SetLength`' in Delphi)
- '`ShortComplex`', '`LongComplex`' <Pine.GSO.4.44.0207151258290.5058-100000@bonsai.fernuni-hagen.de>
- keep files in FDRList while they live, not only while they're open, so '`DoneFDR`' can be applied to all of them ???
- improve TFDDs
- generic types (cf. OOE section C.1) (gentypes.pas) ???

11.2.4 Planned features: Misc

- checking of overflows, variants etc.; check that strings converted to CStrings don't contain #0 characters; initialize strings (length field), pointers, ordinal values and reals(?) with invalid values if checking is on
- warn about using uninitialized variables (kevin2.pas), especially for strings, also for '`for`'-loop counters after the loop (EP 6.8.1)
- transpose arrays (Fortran)? <918557mke1@nnrp1.deja.com>
- inline functions in GPI files
- options to warn about everything that could cause portability problems
- libraries (static/shared; DLL) <Pine.HPP.3.96.971110183550.7996B-100000@tea.geophysik.tu-freiberg.de>
- smart linking (split '`foo.s`', as '`foo*.s`', ar '`foo*.o`' or split '`foo.o`')
- overloading of unary operators ('`+`', '`-`', '`not`') ???
- function overloading (in units and static methods?? – with different parameters, override or overload?); linker name?? (perhaps: first occurrence of a procedure with normal name (=> all non-overloaded procedures get normal names)?); cf. Delphi methods without '`override`'
- '`for var: type = foo to bar do`' ???

- default parameters (cf. OOE section C.4; Delphi 4?) (iniparm[12].pas) <E183vio-000IyH-00@f12.mail.ru>
- variable number of arguments <32F9CFE7.5CB@lmemw.ericsson.se> ?? ???
- multithreading support ?? ???
- '`--wirth-pascal`' :—)
- PXSC standard . . . ('`pxsc.zip`', '`contrib/peter/pxsc`') (??)
- Pascal++ standard . . . (??) ???
- Delphi features: CompToCurrency, CurrencyToComp, Slice, TypeInfo ???, dynamic arrays (tc19991012)
- output column numbers in error messages ??
- debug info: '`with`' statements

11.2.5 Planned features: Utilities

- '`gp`' make utility to replace automake; '`external` *lib*' like '`$L`' (-> BP, Delphi) ?? (fh19990325.2)
- '`pas2texi`' <200301290441.FAA30843@goedel.fjf.gnu.de>
- C header to Pascal translator
- gdb: Pascal types (sets, files, subranges, schemata, . . .)
- '`indent`'-like source code formatter for Pascal
- AT&T <-> Intel assembler syntax converter (-> BP BASM) ???

11.3 Problems that have been solved

This section lists the bugs fixed since the last (non alpha/beta) GPC release, together with the date (YYYYMMDD) on which they were fixed, so you can check if some problem has already been solved in a recent release or developer version. For new features, see Chapter 2 [News], page 9.

- 20050325: check initializers of unused types (fjf898*.pas)
- 20050325: check identifier scopes stricter (az25.pas, fjf1059*.pas)
- 20050321: in '`--borland-pascal`' give only a warning, not an error, for duplicate '`case`' elements (drf6*.pas)
- 20050320: multi-dimensional '`packed`' arrays are not correctly packed (fjf149*.pas)
- 20050316: check arithmetical overflow in constants somewhat better (fjf206*.pas)
- 20050316: check duplicate external declarations stricter (fjf948.pas)
- 20050312: check that actual discriminants match in parameters and assignments, and in strict EP mode, check that actual discriminants of parameters within one identifier list match (berend4.pas, fjf104[45]*.pas, typeof[2-4].pas)
- 20050312: check pointer to schema types stricter (fjf130.pas, fjf614.pas, schema3a.pas affected)
- 20050311: allow importing the same value as a result of distinct expressions under the same name for EP modules for ordinal, pointer, string and set constants (fjf1040*.pas, stef1.pas)
- 20050307: allow attributes for procedural types (chief56.pas)
- 20050307: problem with string operations and '`goto`' (contour0.pas, contourbug.pas, martin1.pas, berend3.pas)
- 20050307: check '`goto`' targets more strictly <261020020000077022%gpaeper@empirenet.com>, <Pine.LNX.4.44.0210281004000.31943-100000@duch.mimuw.edu.pl> (az48.pas, fjf701*.pas, fjf1038*.pas, gale6.pas)

- 20050307: 'Result' is not completely Delphi compatible (fjf1022*.pas)
- 20050307: problem if a 'Write' or 'WriteLn' argument calls a routine which writes to the same file (fjf1036.pas)
- 20050302: 'Update' as well as writing after reading or checking 'EOF' can get the file position wrong in some cases (arp2*.pas, fjf159.pas, fjf160*.pas)
- 20050217: declaring a procedure in the implementation with the same name as an imported procedure doesn't work (chief18.pas), detect name collisions between imported EP modules (mod9.pas)
- 20050214: warn if an lhs cast decreases alignment (avo9.pas)
- 20050214: don't require result variables in forward declarations except in strict ISO mode (marku10*.pas)
- 20050213: writing to a file buffer at EOF doesn't work (eof1.pas)
- 20050211: don't allow 'f^' for unopened files 'f' (pcerror[ij].pas)
- 20050210: evalute initializers exactly as often as specified (fjf1019.pas)
- 20050205: initialized types in record and object fields and in schemata (inirec[24].pas, fjf1016*.pas)
- 20050130: in standard Pascal modes, 'EOF' is false on an empty file (tom7.pas)
- 20050127: wrong 'arithmetical overflow' with 'set of byte' (igor1.pas)
- 20050119: types and initializers containing variable expressions, including schemata with non-constant discriminants and global variables of dynamic size (couper[45].pas, fjf186.pas, fjf548.pas, fjf612a.pas, fjf997*.pas, fjf1003*.pas, john1a.pas)
- 20050117: 'pow', 'Trim', 'Card', set operations ('-', '*', '><') and comparisons as well as string comparisons and array slices with constant arguments cannot be used in constants (fjf998*.pas, fjf1000*.pas, fjf1009*.pas)
- 20050117: 'ParamStr' evaluates its argument twice (fjf963b.pas)
- 20050106: user-defined operators must not be applied to implicit operations (fjf991.pas and many other cases)
- 20041203: accessing components of a constant (EP) constructor (indexed by a constant for arrays) must yield a constant again (dave3*.pas)
- 20041202: GPC crashes when using two variables with identically-named fields in a single 'with' statement (withbug.pas)
- 20041125: GPC doesn't accept 'case' statements without case-list-elements (fjf982*.pas)
- 20041125: gcc-3.x: options are wrongly reordered (so, e.g., '--gnu-pascal -Wno-underscore' doesn't work as expected) (avo7.pas)
- 20041124: applying 'not' to a function result in parentheses doesn't work (avo6.pas)
- 20041123: packed array indices containing 'mod' don't work (avo4.pas)
- 20041120: GPC sometimes prints '???' instead of the actual file name in messages
- 20041028: function results (of record type) must not be allowed as 'with' elements (only a warning in '--delphi' and '--mac-pascal' modes for compatibility); fields of non-lvalue 'with' elements must not be lvalues either (fjf493*.pas)
- 20041022: value parameters of type 'String' (undiscriminated) must take the capacity of the actual parameter, according to EP (waldek11*.pas)
- 20041021: initialized types in arrays (fjf233.pas, fjf974*.pas)
- 20041020: initializers are ignored in 'New' and 'Initialize' (fjf967[j-l].pas)
- 20041020: the address of global routines is not allowed in initializers (avo2*.pas)
- 20041015: 'Index' and 'Pos' cannot be used in constant expressions (fjf970*.pas)

- 20041012: initializers of variant records don't work (fjf259.pas), (peter6.pas) <C1256791.0021F002.00@synln01.synstar.de>
- 20041012: initializers of packed arrays don't work (emil5.pas)
- 20041007: the '__FILE__' and '__BASE_FILE__' macros should return full paths
- 20041006: 'Sqr' sometimes evaluates its argument twice (fjf963a.pas)
- 20041004: memory leak in routines with a local variable of file type that are left via 'Exit' or 'Return' (fjf962.pas)
- 20040916: using a Boolean function as a condition in a 'repeat' loop doesn't work correctly (artur1*.pas)
- 20040913: overstrict type-checking in comparisons involving procedural types (fjf960*.pas)
- 20040908: 'Read', 'Write' etc. evaluate its arguments in the wrong order (az43*.pas)
- 20040908: 'Read' etc. evaluate references to string parameters twice (fjf958.pas)
- 20040907: on targets with 16 bit 'Integer' type, the 'MicroSecond' field of 'TimeStamp' causes a compilation error
- 20040907: character arrays indexed by non-integer ordinal types treated as strings cause internal compiler errors (fjf957*.pas)
- 20040906: '-W[no-]cast-align' does not work (fjf956*.pas)
- 20040903: in 'FormatTime' '%Z' and '%z' unless provided by system library ignore DST
- 20040710: powerpc: '--strength-reduce' doesn't work with 'for' loops (was kludged before, fixed in gcc-3.3.3)
- 20040710: AIX: 'ReturnAddress' doesn't work after use of dynmamic variables (backend bug, fixed in gcc-3.3.3)
- 20040622: functions returning sets are called twice if range-checking is on (inga1*.pas)
- 20040512: 'pow' and '**' are really EP conformant now (in particular 'x pow y' and 'x ** y' are an error if x = 0 and y <= 0) (emil27*.pas)
- 20040511: bug with 'absolute' variables in strange situations (waldek8.pas)
- 20040511: 'protected var' parameters must only accept references (unlike 'const' parameters) (gale5*.pas)
- 20040507: 'pack' must not pack the component type of arrays (fjf940[b-e].pas)
- 20040507: in some circumstances packed fields are allowed as 'var' parameters (fjf940a.pas)
- 20040427: parameters of procedural types don't support Standard Pascal procedural parameters, conformant/open arrays and 'type of' inquiries (fjf939*.pas)
- 20040422: bugs with nonlocal gotos out of routines declaring file variables (nonloc*.pas) (fix involved a change in the internal representation of file variables)
- 20040331: ''foo'#42' is rejected in '--borland-pascal' (chief53.pas)
- 20040330: records/objects with many fields are handled slowly
- 20040328: '--implementation-only' doesn't work correctly (bo4-19.pas)
- 20040325: messages referring to object methods point to the end of the object type rather than the method declaration
- 20040325: bug when slice-accessing a function result of type 'Pointer' type-casted to 'PString' (bo4-18.pas)
- 20040210: incorrect errors with packed arrays and records (fb[12].pas, gale[34].pas)
- 20040204: GPC crashes on 'else' after a missing semicolon in some circumstances (fjf926.pas)
- 20031007: spurious errors about 'result types in forward declared functions' in '--extended-pascal' etc. (dave2.pas)

- 20031004: do not allow disposing of `nil` pointers in standard Pascal modes (fjf917*.pas)
- 20031001: arithmetic expressions don't work as lower array/subrange bounds (fjf204.pas, fjf248.pas, fjf293.pas, fjf336.pas, fjf346a.pas, fjf622.pas)
- 20030925: initializers for types containing nested schemata don't work (fjf914*.pas)
- 20030830: open internal files with `O_EXCL` on `Rewrite` (as a protection against symlink attacks)
- 20030819: GPC accepts, but ignores, options with invalid suffixes (e.g. `--delphi-pascal`)
- 20030729: `pow` and `**` are EP conformant now (in particular `x pow y = (1 div x) pow (y)` if `y` is negative and `x <> 0`) (fjf908.pas)
- 20030714: `--enable-keyword`/`--disable-keyword` on the command-line makes GPC crash (david5.pas)
- 20030704: wrong type-error when applying `Inc` to a type-casted pointer (peter3.pas)
- 20030702: with range checking enabled, check dynamic subrange/array size (fjf222*.pas, fjf813*.pas, fjf900*.pas)
- 20030701: GPC allows modification of conformant array bounds, result of `High`/`Low` etc. (fjf897*.pas)
- 20030626: don't allow linker names starting with a digit (fjf894.pas)
- 20030625: `SubStr` with constant arguments doesn't work in constants (gale1.pas)
- 20030617: handle `BitSizeOf` correctly for packed array fields, don't allow `SizeOf` on them (fjf891*.pas)
- 20030612: System: `BPReal` must be a packed record <3EE8A26D.C919BE7D@flexim.de>
- 20030610: schema types with initializers (drf1.pas, fjf886*.pas)
- 20030610: `Return` doesn't work for sets (fjf885.pas)
- 20030609: bug with arrays as fields of `packed` records (waldek6.pas)
- 20030607: don't allow duplicate imports in a module interface and implementation (nick1b.pas)
- 20030604: compensate for parser read-ahead in the lexer, so compiler directives don't become effective too early and error messages refer more closely to the correct source position
- 20030603: bug when dividing two integers with `/` (fjf481.pas)
- 20030509: don't allow `absolute` in type definitions
- 20030502: subranges with variable limits (couper[23].pas)
- 20030502: Sparc with gcc-2.95.x: `goto` jumping out of two procedure nesting levels doesn't work (GCC bug; fixed in gcc-3) <200111170922.KAA09125@goedel.fjf.gnu.de> (fjf558[op].pas) (fixed with gcc-3 or when using `--longjmp-all-nonlocal-labels`)
- 20030502: the parser does not always recover well after a parse error <199911040915.KAA11168@humulus.daimi.au.dk> (fixed the case given in this report; if there are other cases, please report)
- 20030501: detect conflicts between object fields and local variables in methods
- 20030430: packed array/record fields don't work in `Read` etc. (tom5.pas)
- 20030430: file parameters must not automatically be bindable in `--extended-pascal` (fjf193[c-e].pas)
- 20030423: give an error rather than a warning when casting between types of different size in `{$X-}`
- 20030423: simplify code generated to compute size of dynamical variables if no bitfields are involved (ok with gcc-3)
- 20030422: initialized object variables don't work (fjf445*.pas)

- 20030422: declarations of a module interface are not visible in the implementation (kevin13.pas, mod12.pas) <Pine.BSI.3.96.971110210330.7570A-100000@malasada.lava.net>
- 20030422: detect invalid array slice access with constant indices at compile-time (peter2*.pas)
- 20030421: automatically close dynamically allocated files on 'Dispose' and files declared in a statement block at the end of the statement block <6r9ir5$7v5$1@nntpd.lkg.dec.com> (fjf219[a-d].pas, fjf502.pas)
- 20030421: initialize local static variables in the main constructor, not in each routine call (fjf458*.pas)
- 20030421: check parameter and result variable names in repeated forward etc. declarations <20010321204051.A611@plato> (fjf284.pas, markus8.pas, fjf850*.pas)
- 20030417: modifying 'for'-loop counters within the loop or in a subroutine is not allowed <200005240807.EAA05355@mail.bcpl.net>, <Pine.LNX.4.44.0210281004000.31943-100000@duch.mimuw.edu.pl> (az47.pas, fjf837*.pas)
- 20030417: possible stack overflow when using string concatenation in a loop (fjf419*.pas, fjf345e.pas, fjf460b.pas) – breaks berend3.pas (less important because strange test case, and just another instance of existing contourbug.pas)
- 20030416: some functions in boolean shortcuts are always called (fjf226*.pas)
- 20030414: label declarations must not be allowed in unit/module interfaces and module implementations (but in unit implementations, BP compatible, though we don't allow nonlocal 'goto's into the constructor) (fjf835*.pas)
- 20030321: variables declared in interfaces of modules are not initialized (capacity of strings etc.) (daj3.pas, sven14c.pas, nick1.pas)
- 20030321: subranges whose size is exactly one or two bytes are not packed in packed arrays (daj14a.pas)
- 20030321: 'prior parameter's size depends on 'Foo'' with 'const' string parameters in module interfaces (fjf667.pas)
- 20030313: operators don't always work across units (fjf803.pas)
- 20030312: overloading '<=', '>=', '<>' and some certain words doesn't work (fjf789.pas, fjf794*.pas, fjf800.pas, fjf802.pas, fjf804.pas)
- 20030311: when passing a schema variable as an untyped argument, the whole schema, including the discriminants is passed (fjf798.pas)
- 20030302: discriminant identifiers as variant record selectors
- 20030227: GPC crashes when taking the address of local variables in an initializer (nicola4*.pas)
- 20030225: the warnings about uninitialized/unused variables don't work for strings, objects, etc. (fjf779*.pas)
- 20030221: gcc-2.95.x: 'configure --silent' doesn't work (passes wrong options to subconfigures, so the subsequent make fails) (GCC bug; fixed in 3.x)
- 20030215: forward referencing pointers generate debug info that appears as generic pointers
- 20030202: count of parameters in error messages should not include 'Self' in methods or internal parameters for conformant or open arrays
- 20030129: check for unresolved 'forward', interface and method declarations (az32.pas, fjf758*.pas)
- 20030129: several standard conformance bugs (az{1..24,26..42,44..46}*.pas, emil23*.pas)
- 20030126: some bugs with complicated schema usage (emil22*.pas, fjf750*.pas)
- 20030122: subtraction of unsigned types with a negative result doesn't work (ml4.pas)

- 20021229: declaring huge enum types and exporting subranges of them is very slow (quadratic time behaviour); some bugs regarding exporting of subranges (fjf736*.pas)
- 20021213: Linux: '`crtscreen`' should react to '`SIGWINCH`'
- 20021120: '`Card`' doesn't work with set constructors; '`Include`' and '`Exclude`' should not accept set constructors (eike3*.pas)
- 20021105: type initializers are not type-checked until a variable of the type is declared (fjf704.pas); with gcc-2.x: bug when variables of a type with initializer are declared locally in more than one routine (couper13.pas)
- 20021105: '`packed object`' should not be allowed (fjf703.pas)
- 20021101: bug when replacing a non-virtual method by a virtual one in a descendant object type (fjf702.pas)
- 20021027: classic Pascal does not know the empty string <Pine.LNX.4.44.0210181332470.29475-100000@duch.mimuw.edu.pl> (fjf693*.pas)
- 20021027: relational and exponentiation operators have no associativity <Pine.LNX.4.44.0210210807410.18095-100000@duch.mimuw.edu.pl> (fjf692.pas, fjf566[k-m].pas)
- 20021002: '`gpc -B`*dir*' requires a trailing dir separator
- 20021001: constructors are accepted as the argument to '`Dispose`' (fjf674.pas)
- 20021001: align file fields in packed records on machines with strict alignment requirements (chief38*.pas)
- 20021001: bug on machines with strict alignment requirements <199906021618.MAA06228@sten27.software.mitel.com> (richard1.pas)
- 20020930: duplicate variable declarations are allowed
- 20020929: '`attribute`'s of variables are ignored (fjf673.pas)
- 20020929: '`volatile`' for '`external`' variables without '`asmname`' is ignored (fjf672.pas)
- 20020926: numbers with base specifiers are allowed as labels (fjf417*.pas)
- 20020923: System: '`MemAvail`'/'`MaxAvail`' can go into an endless loop
- 20020920: the number of times the preprocessor is invoked by the automake mechanism might grow exponentially with the number of units involved <02091610572303.14626@dutw54.wbmt.tudelft.nl>
- 20020918: importing '`StandardOutput`' etc. in the interface of a module doesn't work (sietse2*.pas)
- 20020904: comparisons between signed and unsigned integers sometimes give wrong results (eike2.pas, fjf664.pas, martin5.pas)
- 20020903: '`IOSelect`' fails with file handles >= 8 on some systems (e.g., Solaris) (fjf663.pas)
- 20020831: GPC creates wrong debug info for many built-in types <200208280012.g7S0CWj07637@mail.bcpl.net>
- 20020827: comparisons of '`packed`' subrange variables don't work right (martin4[ab].pas)
- 20020824: operators defined in units don't always work (maur11.pas)
- 20020824: object methods that contain an ISO style procedural parameter forget the implicit '`with Self do`' (fjf662a.pas)
- 20020615: '`if Pass[i] in 'A' .. 'Z'`' makes GPC crash (miklos6.pas)
- 20020603: compiling a program (not a unit or module) with '`--interface-only`' or '`--syntax-only`' segfaults (waldek1.pas)
- 20020603: '`--nested-comments`' fails without '`-Wall`' (waldek2.pas)
- 20020514: powerpc: '`--strength-reduce`' doesn't work with '`for`' loops [kludged now]
- 20020514: guarantee complete evaluation in '`{$B+}`' mode (fjf552*.pas)
- 20020514: spurious warning with '`for`' loops using a '`ByteCard`' counter (toby1.pas)

12 The GPC Source Reference

"The Source will be with you. Always."

This chapter describes internals of GPC. It is meant for GPC developers and those who want to become developers, or just want to know more about how the compiler works. It does not contain information needed to just use GPC to compile programs.

This chapter tells you how to look up additional information about the GNU Pascal compiler from its source code.

Please note: If you intend to modify GPC's source, please check the top of each file you're going to modify. A number of files are generated automatically by various tools. The top of these files will tell you by which tool and from what file they were generated. Modifying a generated file is pointless, since it will be overwritten the next time the tool is run. Instead, modify the original source (which will usually be easier in fact, e.g. a bison input file vs. the generated C code). This also holds for various documentation and other files.

Proprietary compilers often come with a lot of technical information about the internals of the compiler. This is necessary because their vendors want to avoid to distribute the source of the compiler – which is always the most definitive source of this technical information.

With GNU compilers, on the other hand, you are free to get the source code, look how your compiler works internally, customize it for your own needs, and to re-distribute it in modified or unmodified form. You may even take money for this redistribution. (For details, see the GNU General Public License, Appendix A [Copying], page 491.)

The following subsections are your guide to the GNU Pascal source code. If you have further questions, be welcome to ask them at the GNU Pascal mailing list (see Chapter 10 [Support], page 459).

All file paths mentioned in this chapter are relative to the GNU Pascal source directory, a subdirectory 'gcc/p' below the top-level GCC source directory.

The following sections roughly coincide with the order of the steps a Pascal source passes through during compilation (omitting the code generation which is the job of the GCC backend, and the assembler and linker steps at the end which are done by the programs 'as' and 'ld' of binutils and possibly other utilities like 'collect2'). Also missing here is the compiler driver 'gpc' which behaves very similarly to 'gcc' and whose main job is to invoke the other parts in the right order, with the right arguments etc.

Note, this chapter docuemnts only selected parts of the compiler. Many things are missing because nobody has yet had the time to write something about them. In any case, for real understanding of the inner workings, you should always refer to the source code.

For more information, see the manual of GCC internals, [Top], page 497.

12.1 The Pascal preprocessor

'gpcpp' is based on the C preprocessor, so it does everything 'cpp' does (see the cpp manual) and some more. In particular:

- Comments like 'cpp' does, but within '{ ... }' and '(* ... *)', also after '//' if 'delphi-comments' is active, never within '/* ... */'. Also mixed comments ('{ ... *)', '(* ... }') if enabled ('mixed-comments') and nested comments (e.g. '{ ... { ... } ... }') if enabled ('nested-comments')
- Macros and conditionals like 'cpp' does, but both case sensitive and insensitive ones; 'no-macros' to turn macro expansion off (e.g., for BP compatibility)
- 'ifopt' for short and long options
- Include files like 'cpp' does, but also with '{$I ...}' (BP style), which allows the file name extension to be omitted

- Recognize Pascal strings (to avoid looking for comments and directives within strings) enclosed in single (like Standard Pascal) or double quotes (like C).
- Option handling, sharing tables in '`gpc-options.h`' with the compiler:
  - Default option settings
  - Options can imply other options (e.g., '`borland-pascal`' -> '`no-macros`' etc.)
  - Short compiler directives
  - Short directive '`W`' (warnings) is disabled in '`borland-pascal`' and '`delphi`' because it has another meaning there
- Compiler directives ('`{$...}`' or '`(*$...*)`'):
  - pass them through, so the compiler can handle them
  - keep track of them for '`ifopt`'
  - handle those that affect the preprocessor (e.g., about comments)
  - allow comments within compiler directives if nested comments are enabled
  - local directives
  - case insensitive
- Slightly Pascal-like syntax for conditional compilation ('`not`' -> '`!`', '`and`' -> '`&&`', '`or`' -> '`||`', '`xor`' -> '`!=`', '`shl`' -> '`<<`', '`shr`' -> '`>>`', '`False`' -> '`0`', '`True`' -> '`1`', '`<>`' -> '`!=`', '`=`' -> '`==`')
- Line directives like '`cpp`' does, but recognize BP style ('`#42`' or '`#$f0`') character constants and don't confuse them with line directives (the latter seem to always have a space after the '`#`')

12.2 GPC's Lexical Analyzer

The source files '`gpc-lex.c`' and '`pascal-lex.l`' contain the so-called *lexical analyzer* of the GNU Pascal compiler. The latter file is processed with '`flex`' to produce '`pascal-lex.c`' which is not meant to be read by a human, but compiled.

This very first stage of the compiler is responsible for reading what you have written and dividing it into *tokens*, the "atoms" of a computer language. Here is, for example, where integer and real numbers such as '`42`' and '`3.14e-5`', string constants, symbols, keywords and identifiers etc. are recognized.

The main entry point is the function '`yylex`' which calls the flex-generated function '`lexscan`' which does the main work of token separation.

12.2.1 Lexer problems

Pascal is a language that's easy to lex and parse. Then came Borland ...

A number of their ad-hoc syntax extensions cause lexing or parsing problems, and even ambiguities. This lexer tries to solve them as well as possible, sometimes with clever rules, other times with gross hacks and with help from the parser. (And, BTW, it handles regular Pascal as well. ;-)

Some of the problems are (see also see Section 12.3.1 [Parsing conflicts], page 480):

- Real constants with a trailing '`.`'. Problem: They make the character sequence '`2.)`' ambiguous. It could be interpreted as '`2.0`', followed by '`)`' or as '`2`' and '`.)`' (which is an alternative for '`]`'). This lexer chooses the latter interpretation, like BP does, and the standard requires. It would be possible to handle both, by keeping a stack of the currently open parentheses and brackets and chosing the matching closing one, but since BP does not do this, either, it doesn't seem worth the trouble. (Or maybe later ... ;-)

- They also cause a little problem in the sequence '`2..`' (the start of an integer subrange), but this is easily solved by normal lexer look-ahead since a real constant can't be followed by a '`.`' in any Pascal dialect we know of.
- Missing token separators between integer or real constants and a following keyword (e.g. '`42to`'). It gets worse with hex numbers ('`$abcduntil`'), but it's not really difficult to lex. However, we don't allow this with Extended Pascal non-decimal integer constants, e.g. '`16#abcduntil`' where it would be a little more difficult (because it would depend on the base whether or not '`u`' is a digit). Since BP does not even support EP non-decimal constants, there's no point in going to such troubles.
- Character constants with '`#`'. They conflict with the Extended Pascal non-decimal integer number notation. '`#13#10`' could mean '`Chr (13) + Chr (10)`' or '`Chr (13#10)`'. This lexer chooses the former interpretation, since the latter one would be a mix of BP and Extended Pascal features.
- Last (but not least – no, certainly worst): Character constants with '`^`' (was this "feature" meant as an AFJ or something???). GPC tries to make the best out of a stupid situation, see the next section (see Section 12.2.2 [BP character constants], page 477) for details. It should be noted that BP itself fails in a number of situations involving such character constants, probably the clearest sign for a design bug.
- But also GPC's extension '`...`' for variadic external function declarations causes a problem in the sequence '`(...)`' which could mean '`(`', '`...`', '`)`', i.e., a parameter list with only variadic arguments, or '`(.`', '`.`', '`.)`'. Since the latter token sequence is meaningless in any Pascal dialect we know of, this lexer chooses the former one which is easily accomplished with normal look-ahead.

12.2.2 BP character constants

Borland-style character constants of the form '`^M`' need special care. For example look at the following type declaration:

```
type
  X = Integer;
  Y = ^X;        { pointer type }
  Z = ^X .. ^Y;  { subrange type }
```

One way one could attempt to resolve this is to let the parser tell the lexer (via a global flag) whether a character constant or the symbol '`^`' (to create pointer types or to dereference pointer expressions) is suitable in the current context. This was done in previous versions, but it had a number of disadvantages: First, any dependency of the lexer on the parser (see ⟨undefined⟩ [Lexical Tie-Ins], page ⟨undefined⟩) is problematic by itself since it must be taken care of manually in each relevant parser rule. Furthermore, the parser read-ahead must be taken into account, so the flag must usually be changed apparently one token too early. Using GLR (see ⟨undefined⟩ [GLR Parsers], page ⟨undefined⟩) makes this problem worse since it may read many tokens while the parser is split before it can perform any semantic action (which is where the flag could be modified). Secondly, as the example above shows, there are contexts in which both meanings are acceptable. So further look-ahead (within the lexer) was needed to resolve the problem.

Therefore, we now use another approach. When seeing '`^X`', the lexer returns two tokens, a regular '`^`' and a special token '`LEX_CARET_LETTER`' with semantic value '`X`'. The parser accepts '`LEX_CARET_LETTER`' wherever an identifier is accepted, and turns it into the identifier '`X`' via the nonterminal '`caret_letter`'. Furthermore, it accepts the sequence '`^`', '`LEX_CARET_LETTER`' as a string constant (whose value is a one-character string). Since '`LEX_CARET_LETTER`' is only produced by the lexer immediately after '`^`', with no white-space in between, this works (whereas otherwise, pasting tokens in the parser is not reliable due to white-space, e.g. the token sequence '`:`' and '`=`' could stand for '`:=`' (if '`:=`' weren't a token by itself), but also for '`: =`' with a space in between). With this trick, we can handle '^' followed by a single letter or underscore. The fact that this doesn't cause any conflict in the grammar tell us that this method works.

However, BP even allows any other character after '^' as a char constant. E.g., '^)' could be a pointer dereference after an expression and followed by a closing parenthesis, or the character 'i' (sic!).

Some characters are unproblematic because they can never occur after a '^' in its regular meaning, so the sequence can be lexed as a char constant directly. These are all characters that are not part of any Pascal tokens at all (which includes all control characters except white-space, all non-ASCII characters and the characters '!', '&', '%', '?', '\', '`', '|', '~' and '}' – the last one occurs at the end of comments, but within a comment this issue doesn't occur, anyway) and those characters that can only start constants because a constant can never follow a '^' in Pascal; these are '#', '$', ''', '"' and the digits.

For '^' followed by whitespace, we return the token 'LEX_CARET_WHITE' which the parser accepts as either a string constant or equivalent to '^' (because in the regular meaning, the white-space is meaningless).

If '^' is followed by one of the remaining characters (apart from one, see below), namely ',', '.', ':', ';', '(', ')', '[', ']', '+', '-', '*', '/', '<', '=', '>', '@', '^', the lexer just returns the tokens regularly, and the parser accepts these sequences as a char constant (besides the normal meaning of the tokens). (Again, since white-space after '^' is already dealt with, this token pasting works here.)

But '^' can also be followed by a multi-character alphanumeric sequence such as '^cto' which might be read as '^ cto' or '^c to' (since BP also allows omitting white-space after constants), or by a multi-character token such as '^<=' which could be '^ <=' or '^< ='. Both could be solved with extra tokens, e.g. lexing '^<=' as '^', 'LEX_CARET_LESS', '=' and accepting '^', 'LEX_CARET_LESS' in the parser as a string constant and 'LEX_CARET_LESS', '=' as equivalent to '<=' (relying on the fact that the lexer doesn't produce 'LEX_CARET_LESS' if there's white-space after the '<' because then the simple '^', '<' will work, so justifying the token-pasting once again). This has not been done yet (in the alphanumeric case, this might add a lot of special tokens because of keywords etc., and it's doubtful whether that's worth it).

Finally, we have '^{' and '^(*'. This is so incredibly stupid (e.g., think of the construct 'type c = Integer; foo = ^{ .. ^|; bar = {} c;' which would become ambiguous then), that perhaps we should not attempt to handle this ...

(As a side-note, BP itself doesn't handle '^' character constants in many situations, including many that GPC does handle with the mechanisms described above, probably the clearest sign for a design bug. But if we support them at all, we might just as well do it better than BP ... :–)

12.2.3 Compiler directives internally

Compiler directives are mostly handled in 'options.c', mostly in common with command-line options, using the definitions in 'lang-options.h' and the tables in 'gpc-options.h'.

A special problem is that the parser sometimes has to read tokens before they're used to decide what to do next. LALR(1) parsers would read at most one such token, but with GLR, the parser can split and consume tokens while not doing any actions. The number of such tokens is unbounded, though the relevant context can be determined by analyzing the grammar.

Reading look-ahead tokens is generally harmless, but if there is a compiler directive before such a look-ahead token, it would be handled apparently too early. This looks strange from the programmer's point of view – even more so since the programmer cannot easily predict when the parser needs to read ahead and when not, and therefore cannot be sure where exactly to place the directive. This is particularly important for local directives that are meant to have a scope as small as possible.

To solve this problem, GPC keeps those options that can be changed by directives in a linked list of '`struct options`'. There are several pointers into the list:

'`lexer_options`' are the options current to the lexer. These are always the ones read most recently. Compiler directives are applied here when read. Each directive causes a new '`struct options`' to be chained to the list.

'`compiler_options`' points to the options current for the compiler, i.e. seen before the last token handled in a parser rule. To facilitate this, we abuse Bison's location tracking feature (see ⟨undefined⟩ [Locations], page ⟨undefined⟩) and refer to the options seen before a token in the token's location ('`yylloc`'). Before each grammar rule is handled, the compiler options are pointed to those of the last token involved in the rules handled so far, using Bison's '`YYLLOC_DEFAULT`' feature. Actual locations, used for error messages etc., are handled the same way (according to the real purpose of Bison's location tracking), also distinct for the lexer and compiler.

Please Note: Tokens are not always handled in order. E.g., in '`2 + 3 * 4`', first '`3 * 4`' is evaluated, then '`2 + 12`', i.e., the tokens '`2`' and '`+`' are handled after the following ones. To avoid jumping back in the options, we store a counter, rather than a pointer, in '`yyloc`', so we can compare it to the current counter. This also allows us to free any '`struct options`' that '`compiler_options`' has advanced beyond because it can never go back.

Finally, the pointer '`co`' points to the current options which is '`lexer_options`' when we're in the lexer and '`compiler_options`' otherwise. All routines that use or set options refer to '`co`', so there is no problem when they may be called both from the lexer and from other parts of the compiler. (Previously, '`lookup_name`' was such a routine, but now the lexer doesn't call it anymore.)

Please Note: Some of the options are flags declared in the backend. Since we can't keep them in '`struct option`' directly, we have to copy them back and forth in '`activate_options`'. This is a little annoyance, but no real problem.

12.3 Syntax parsing: GPC's Parser

The file '`parse.y`' contains the "bison" source code of GNU Pascal's parser. This stage of the compilation analyzes and checks the syntax of your Pascal program, and it generates an intermediate, language-independent code which is then passed to the GNU back-end.

The *bison* language essentially is a machine-readable form of the Backus-Naur Form, the symbolic notation of grammars of computer languages. "Syntax diagrams" are a graphical variant of the Backus-Naur Form.

For details about the "bison" language, see the Bison manual (see [Top], page 497). A short overview how to pick up some information you might need for programming follows.

Suppose you have forgotten how a variable is declared in Pascal. After some searching in '`parse.y`' you have found the following:

```
simple_decl_1:
    ...
  | p_var variable_declaration_list
      { [...] }
  ;

variable_declaration_list:
    variable_declaration { }
  | variable_declaration_list variable_declaration
  ;
```

Translated into English, this means: "A declaration can (amoung other things like types and constants, omitted here) consist of the keyword (lexical token) '`var`' followed by a 'variable declaration list'. A 'variable declaration list' in turn consists of one or more 'variable declarations'."

(The latter explanation requires that you understand the recursive nature of the definition of 'variable_declaration_list'.)

Now we can go on and search for 'variable_declaration'.

```
variable_declaration:
    id_list_limited ':' type_denoter_with_attributes
      { [...] }
    absolute_or_value_specification optional_variable_directive_list ';'
      { [...] }
  ;
```

The '[...]' are placeholders for some C statements, the *semantic actions* which (for the most part) aren't important for understanding GPC's grammar.

From this you can look up that a variable declaration in GNU Pascal consists of an identifier list, followed by a colon, "type denoter with attributes", an "absolute or value specification" and an "optional variable directive list", terminated by a semicolon. Some of these parts are easy to understand, the others you can look up from 'parse.y'. Remember that the reserved word 'var' precedes all this.

Now you know how to get the exact grammar of the GNU Pascal language from the source.

The semantic actions, not shown above, are in some sense the most important part of the bison source, because they are responsible for the generation of the intermediate code of the GNU Pascal front-end, the so-called *tree nodes* (which are used to represent most things in the compiler). For instance, the C code in "type denoter" returns (assigns to '$$') information about the type in a variable of type 'tree'.

The "variable declaration" gets this and other information in the numbered arguments ('$1' etc.) and passes it to some C functions declared in the other source files. Generally, those functions do the real work, while the main job of the C statements in the parser is to call them with the right arguments.

This, the parser, is the place where it becomes Pascal.

12.3.1 Conflicts in the Pascal syntax

Some problematic parts of Pascal syntax in various dialects (see also see Section 12.2.1 [Lexer problems], page 476):

- BP style initializers for variables (or "typed constants", as BP likes to call initialized variables even though they are not constant) with '=' rather than 'value'. Problem: It makes initialized Boolean subrange variable declarations like 'Foo: False .. True = False = False' ambiguous. They could be interpreted as 'Foo: False .. (True = False) = False' or 'Foo: False .. True = (False = False)'. This lexer, like BP, chooses the latter interpretation. To avoid conflicts in the parser, this is done with the 'LEX_CONST_EQUAL' hack, counting parentheses and brackets so that in 'Foo: False .. (True = False) = True' the **second** '=' will become the LEX_CONST_EQUAL token.
- BP style array initializers in '(', ')'. When they consist of a single entry (without an index as required in EP), they conflict with expressions in parentheses. This is resolved in the parser using GLR with '%dprec', and in the following processing of initializers.
- Extended Pascal (and many dialects) allow the lower bound of a subrange declaration to be an arbitrary expression (Standard Pascal allows only a plain constant). This makes things like 'var Foo: (Bar ...' hard to parse, since 'Bar' could be part of an expression in parentheses as the lower bound of a subrange, or the beginning of an enumeration type declaration. BP can't handle this situation. In GPC, this is now solved with the GLR parser. In fact, this case is rather easy to handle for GLR, as it does not even need '%dprec' or '%merge' since the presence or absence of '..' sooner or later makes one alternative fail.

- Delphi's '`external [`*libname*`] [name` *name*`]`' construct where *libname* and *name* can be string expressions. Since '`name`' is not a reserved word, but an identifier, '`external name name name`' can be valid which is difficult to parse. It could be solved by the parser, by making '`name`' a special identifier whose special meaning is recognized after '`external`' only.

12.3.2 So many keywords, so many problems ...

Keywords can be potential problems since they are (generally) not available for use as identifiers. Only those keywords that are defined in ISO 7185 Pascal are unproblematic because no valid program should ever use them as identifiers.

To cope with this problem, GPC does several things:

- If a dialect option is set, only keywords of the specified dialect are enabled. All possible keywords, together with their dialects, are defined in '`predef.def`'. However, compiling with dialect options is usually not recommended, so this is no good general solution.
- The user can turn off individual keywords using the compiler directive '`{$disable-keyword}`'. This makes sure that every conflict with a user's identifier *can* be avoided, but with extra work on the part of the user.
- The parser used to enable and disable keywords in certain syntactic contexts. However, this was rather fragile since it interacts with the parser's read-ahead, and it requires attention on every related change in the parser. Therefore, this mechanism was removed.
- All non-ISO-7185 keywords are now treated as "weak". This means, they are only recognized as keywords if no current declaration of this name exists. However, so that this can work, it must be possible to create new declarations of this name in the first place – at this point, no declaration exists yet, so the name is recognized as a keyword.

  This is solved by listing these keywords in the '`new_identifier`' rule of the parser. This means, first the lexer recognizes them as keywords, then the parser "turns them back" into identifiers. The advantage, compared to explicit enabling and disabling of keywords, is that bison automatically finds the places in which to apply the '`new_identifier`' rule, i.e. treat them as plain identifiers.

  Of course, there is a catch. Since the keyword tokens are listed in '`new_identifier`', they can conflict with occurrences of the actual keywords (bison will find such cases as S/R or R/R conflicts). Such conflicts have to be sorted out carefully and either removed or left to GLR handling. Fortunately, for many keywords, removing the conflicts turned out quite easy – in some cases no conflicts arose at all.

12.3.3 '`forward`', '`near`' and '`far`' as weak keywords

'`forward`' is a little special in ISO 7185 in that it is no keyword, so it may be used as an identifier and a directive at the same time. That's more than what our weak keywords allow.

This problem would be easy to solve if we just parsed it as a plain identifier ('`LEX_ID`') and then check that it was in fact '`forward`'.

However, the same applies to the BP directives '`near`' and '`far`'. (At least so it seems – the BP documentation claims they're reserved words, but the compiler seems to think otherwise.)

Parsing all the three together as an identifier and then checking which one it was fails because '`forward`' is a remote directive, i.e. a routine declared so has no body, while '`near`' and '`far`' are not. So it makes a syntactical difference for what follows.

So we lex the three like regular (non-weak) keywords, but throw their tokens together with '`LEX_ID`' very early in the parser, in the '`id`' rule which is used everywhere an existing identifier is expected. But in the context of these three directives, no identifier is allowed, so the three tokens can be used without conflicts between each other or with '`id`'.

12.4 Tree Nodes

If you want really to understand how the GNU Pascal language front-end works internally and perhaps want to improve the compiler, it is important that you understand GPC's internal data structures.

The data structure used by the language front-end to hold all information about your Pascal program are the so-called "tree nodes". (Well, it needn't be Pascal source – tree nodes are language independent.) The tree nodes are kind of objects, connected to each other via pointers. Since the GNU compiler is written in C and was created at a time where nobody really thought about object-oriented programming languages yet, a lot of effort has been taken to create these "objects" in C.

Here is an extract from the "object hierarchy". Omissions are marked with "..."; nodes in parentheses are "abstract": They are never instantiated and aren't really defined. They only appear here to clarify the structure of the tree node hierarchy. The complete list is in '`../tree.def`'; additional information can be found in '`../tree.h`'.

```
(tree_node)
|
|--- ERROR_MARK  { enables GPC to continue after an error }
|
|--- (identifier)
|     |
|     |--- IDENTIFIER_NODE
|     |
|     \--- OP_IDENTIFIER
|
|--- TREE_LIST  { a list of nodes, also used as a
|                  general-purpose "container object" }
|
|--- TREE_VEC
|
|--- BLOCK
|
|--- (type)  { information about types }
|     |
|     |--- VOID_TYPE
|     |
|     |--- INTEGER_TYPE
|    ...
|     |
|     |--- RECORD_TYPE
|     |
|     |--- FUNCTION_TYPE
|     |
|     \--- LANG_TYPE  { for language-specific extensions }
|
|--- INTEGER_CST  { an integer constant }
|
|--- REAL_CST
|
|--- STRING_CST
|
|--- COMPLEX_CST
```

```
|
|--- (declaration)
|     |
|     |--- FUNCTION_DECL
|    ...
|     |
|     |--- TYPE_DECL
|     |
|     \--- VAR_DECL
|
|--- (reference)
|     |
|     |--- COMPONENT_REF
|    ...
|     |
|     \--- ARRAY_REF
|
|--- CONSTRUCTOR
|
\--- (expression)
      |
      |--- MODIFY_EXPR  { assignment }
      |
      |--- PLUS_EXPR  { addition }
     ...
      |
      |--- CALL_EXPR  { procedure/function call }
      |
      |--- GOTO_EXPR
      |
      \--- LOOP_EXPR  { for all loops }
```

Most of these tree nodes – struct variables in fact – contain pointers to other tree nodes. A 'TREE_LIST' for instance has a 'TREE_VALUE' and a 'TREE_PURPOSE' slot which can contain arbitrary data; a third pointer 'TREE_CHAIN' points to the next 'TREE_LIST' node and thus allows us to create linked lists of tree nodes.

One example: When GPC reads the list of identifiers in a variable declaration

```
var
  Foo, Bar, Baz: Integer;
```

the parser creates a chain of 'TREE_LIST's whose 'TREE_VALUE's hold 'IDENTIFIER_NODE's for the identifiers 'Foo', 'Bar', and 'Baz'. The function 'declare_variables()' (declared in 'declarations.c') gets this tree list as a parameter, does some magic, and finally passes a chain of 'VAR_DECL' nodes to the back-end.

The 'VAR_DECL' nodes in turn have a pointer 'TREE_TYPE' which holds a '_TYPE' node – an 'INTEGER_TYPE' node in the example above. Having this, GPC can do type-checking when a variable is referenced.

For another example, let's look at the following statement:

```
Baz := Foo + Bar;
```

Here the parser creates a 'MODIFY_EXPR' tree node. This node has two pointers, 'TREE_OPERAND[0]' which holds a representation of 'Baz', a 'VAR_DECL' node, and 'TREE_OPERAND[1]' which holds a representation of the sum 'Foo + Bar'. The sum in turn is represented as a ‘PLUS_EXPR’ tree node whose ‘TREE_OPERAND[0]’ is the ‘VAR_DECL’ node ‘Foo’, and whose ‘TREE_OPERAND[1]’ is the ‘VAR_DECL’ node ‘Bar’. Passing this (the ‘MODIFY_EXPR’ node) to the back-end results in assembler code for the assignment.

If you want to have a closer look at these tree nodes, write a line ‘{$debug-tree FooBar}’ into your program with ‘FooBar’ being some identifier in your program. This tells GPC to output the contents of the ‘IDENTIFIER_NODE’ to the standard error file handle in human-readable form.

While hacking and debugging GPC, you will also wish to have a look at these tree nodes in other cases. Use the ‘debug_tree()’ function to do so. (In fact ‘{$debug-tree FooBar}’ does nothing else than to ‘debug_tree()’ the ‘IDENTIFIER_NODE’ of the ‘Foobar’ identifier node – note the capitalization of the first character in the internal representation.)

12.5 Parameter Passing

GPC supports a lot of funny things in parameter lists: value and reference, ‘protected’ and ‘const’ parameters, strings and other schemata with specified or unspecified discriminants, conformant and open arrays, objects, procedural parameters, untyped reference parameters, etc. All this requires sophisticated type-checking; the responsible function is ‘convert_arguments()’ in the source file ‘typecheck.c’. Every detail can be looked up from there.

Some short notes about the most interesting cases follow.

Conformant arrays:
: First, the array bounds are passed (an even number of parameters of an ordinal type), then the address(es) of the array(s) themselves.

Procedural parameters:
: These need special care because a function passed as a parameter can be confused with a call to the function whose result is then passed as a parameter. See also the functions ‘maybe_call_function()’ and ‘probably_call_function()’ in ‘expressions.c’.

Chars: According to ISO 10206 Extended Pascal, formal char parameters accept string values. GPC does the necessary conversion implicitly. The empty string produces a space.

Strings and schemata:
: Value parameter strings and schemata of known size are really passed by value. Value parameter strings and schemata of unknown size are passed by reference, and GPC creates temporary variable to hold a copy of the string.

‘CString’ parameters:
: GPC implicitly converts any string value such that the address of the actual string data is passed and appends a ‘Chr (0)’ terminator when necessary.

‘const’ parameters:
: If a constant value is passed to a ‘const’ parameter, GPC assigns the value to a temporary variable whose address is passed. Exception: Small types (whose size is known and not bigger than that of a pointer) as well as all integer, real and complex types are passed by value.

Untyped parameters:
: These are denoted by ‘var foo’ or ‘var foo: Void’ and are compatible to C’s ‘void *’ parameters; the size of such entities is *not* passed. Maybe we will change this in the future and pass the size for ‘var foo’ parameters whereas ‘var foo: Void’ will remain compatible to C. (Same with ‘const’ instead of ‘var’.)

12.6 GPI files – GNU Pascal Interfaces

This section documents the mechanism how GPC transfers information from the exporting modules and units to the program, module or unit which imports (uses) the information.

A GPI file contains a precompiled GNU Pascal interface. "Precompiled" means in this context that the interface already has been parsed (i.e. the front-end has done its work), but that no assembler output has been produced yet.

The GPI file format is an implementation-dependent (but not *too* implementation-dependent ;−) file format for storing GNU Pascal interfaces to be exported – Extended Pascal and PXSC module interfaces as well as interface parts of UCSD/Borland Pascal units compiled with GNU Pascal.

To see what information is stored in or loaded from a GPI file, run GPC with an additional command-line option '`--debug-gpi`'. Then, GPC will write a human-readable version of what is being stored/loaded to the standard error file handle. (See also: Section 12.4 [Tree nodes], page 482.) **Please note:** This will usually produce *huge* amounts of output!

While parsing an interface, GPC stores the names of exported objects in tree lists – look for '`handle_autoexport`' in the GPC source files. At the end of the interface, everything is stored in one or more GPI files. This is done in '`module.c`'. There you can find the source of '`create_gpi_files()`' which documents the file format:

First, a header of 33 bytes containing the string '`GNU Pascal unit/module interface`' plus a newline.

This is followed by an integer containing the "magic" value 12345678 (hexadecimal) to carry information about the endianness. Note that, though a single GPI file is always specific to a particular target architecture, the host architecture (i.e., the system on which GPC runs) can be different (cross-compilers). Currently, GPC is not able to convert endianness in GPI files "on the fly", but at least it will detect and reject GPI files with the "wrong" endianness. When writing GPI files, always the host's endianness is used (this seems to be a good idea even when converting on the fly will be supported in the future, since most often, GPI files created by a cross-compiler will be read again by the same cross-compiler). "Integer" here and in the following paragraphs means a '`gpi_int`' (which is currently defined as '`HOST_WIDE_INT`').

The rest of the GPI file consists of chunks. Each chunk starts with a one-byte code that describes the type of the chunk. It is followed by an integer that specifies the size of the chunk (excluding this chunk header). The further contents depend on the type, as listed below.

For the numeric values of the chunk type codes, please refer to '`GPI_CHUNKS`' in '`module.c`'. Chunk types denoted with '`(*)`' must occur exactly once in a GPI file. Other types may occur any number of times (including zero times). The order of chunks is arbitrary. "String" here simply means a character sequence whose length is the chunk's length (so no terminator is needed).

'`GPI_CHUNK_VERSION`' (String) (*)
: The version of the GPI file which is the same as the GPC version. If '`USE_GPI_DEBUG_KEY`' is used (which will insert a "magic" value at the beginning of each node in the node table, see below, so errors in GPI files will be detected more reliably), '` D`' is appended to this version string. (Currently, '`USE_GPI_DEBUG_KEY`' is used by default.) Furthermore, the GCC backend version is appended, since it also influences GPI files.

'`GPI_CHUNK_TARGET`' (String) (*)
: The target system the GPI file was compiled for.

'`GPI_CHUNK_MODULE_NAME`' (String) (*)
: The name of the unit/module.

'`GPI_CHUNK_SRCFILE`' (String) (*)
: The name of the primary source file of the unit/module.

'GPI_CHUNK_IMPORT'
: The name of an interface imported by the current interface. This chunk consists of a string followed by the checksum of the imported interface's nodes, so the chunk length is the length of the string plus the size of an integer. Again, no terminator of the string is needed.

  The checksum is currently a simple function of the contents of the 'GPI_CHUNK_NODES' chunk's contents (see below). This might be replaced in the future by a MD5 hash or something else more elaborate.

'GPI_CHUNK_LINK' (String)
: The name of a file to link.

'GPI_CHUNK_LIB' (String)
: The name of a library to link (prefixed with '-l').

'GPI_CHUNK_INITIALIZER' (String)
: The name of a module initializer. For technical reasons, any such chunk must come *after* the 'GPI_CHUNK_MODULE_NAME' chunk.

'GPI_CHUNK_GPC_MAIN_NAME' (String)
: A 'gpc-main' option given in this interface. (More than one occurrence is pointless.)

'GPI_CHUNK_NODES' (*)
: The exported names and the objects (i.e., constants, data types, variables and routines) they refer to are internally represented as so-called *tree nodes* as defined in the files '../tree.h' and '../tree.def' from the GNU compiler back-end. (See also: Section 12.4 [Tree nodes], page 482.)

  The main problem when storing tree nodes is that they form a complicated structure in memory with a lot of circular references (actually, not a tree, but a directed graph in the usual terminology, so the name "tree nodes" is actually a misnomer), so the storing mechanism must make sure that nothing is stored multiple times.

  The functions 'load_node()' and 'store_node_fields()' do the main work of loading/storing the contents of a tree node with references to all its contained pointers in a GPI file. Each tree node has a 'TREE_CODE' indicating what kind of information it contains. Each kind of tree nodes must be stored in a different way which is not described here. See the source of these functions for details.

  As most tree nodes contain pointers to other tree nodes, 'load_node()' is an (indirectly) recursive function. Since this recursion can be circular (think of a record containing a pointer to a record of the same type), we must resolve references to tree nodes which already have been loaded. For this reason, all tree nodes being loaded are kept in a table ('rb.nodes'). They are entered there *before* all their fields have been loaded (because loading them is what causes the recursion). So the table contains some incomplete nodes during loading, but at the end of loading a GPI file, they have all been completed.

  On the other hand, for 'store_node_fields()' the (seeming) recursion must be resolved to an iterative process so that the single tree nodes are stored one after another in the file, and not mixed together. This is the job of 'store_tree()'. It uses a hash table (see 'get_node_id()') for efficiency.

  When re-exporting (directly or indirectly) a node that was imported from another interface, and a later compiler run imports both interfaces, it must merge the corresponding nodes loaded from both interfaces. Otherwise it would get only similar, but not identical items. However, we cannot simply omit the re-exported nodes from the new interface in case a later compiler run imports only one of them. The same problem occurs when a module exports several interfaces. In this case, a program that imports more than one of them must recognize their contents as identical where they overlap.

Therefore, each node in a GPI file is prefixed (immediately before its tree code) with information about the interface it was originally imported from or stored in first. This information is represented as a reference to an 'INTERFACE_NAME_NODE' followed by the id (as an integer) of the node in that interface. If the node is imported again and re-re-exported, this information is copied unchanged, so it will always refer to the interface the node was originally contained it. For nodes that appear in an interface for the first time (the normal case), a single 0 integer is stored instead of interface 'INTERFACE_NAME_NODE' and id (for shortness, since this information is implicit).

This mechanism is not applied to 'INTERFACE_NAME_NODE's since there would be a problem when the identifier they represent is the name of the interface they come from; neither to 'IDENTIFIER_NODE's because they are handled somewhat specially by the backend (e.g., they contain fields like 'IDENTIFIER_VALUE' which depend on the currently active declarations, so storing and loading them in GPI files would be wrong) because there is only one 'IDENTIFIER_NODE' ever made for any particular name. But for the same reason, it is no problem that the mechanism can't be applied to them.

'INTERFACE_NAME_NODE's are a special kind of tree nodes, only used for this purpose. They contain the name of the interface, the name of the module (to detect the unlikely case that different modules have interfaces of the same name which otherwise might confuse GPC), and the checksum of that interface. The latter may seem redundant with the checksum stored in the 'GPI_CHUNK_IMPORT' chunk, but in fact it is not. On the one hand, 'GPI_CHUNK_IMPORT' chunks occur only for interfaces imported directly, while the 'INTERFACE_NAME_NODE' mechanism might also refer to interfaces imported indirectly. On the other hand, storing the checksum in the 'GPI_CHUNK_IMPORT' chunks allows the import mechanism to detect discrepancies and refuse to load inconsistent interfaces, whereas during the handling of the 'GPI_CHUNK_NODES' chunk, the imported modules must already have been loaded. (It would be possible to scan the 'GPI_CHUNK_NODES' chunk while deciding whether to recompile, but that would be a lot of extra effort, compared to storing the checksum in the 'GPI_CHUNK_IMPORT' chunks.)

Finally, at the end of the 'GPI_CHUNK_NODES' chunk, a checksum of its own contents (excluding the checksum itself, of course) is appended. This is to detect corrupted GPI files and is independent of the other uses of checksums.

'GPI_CHUNK_OFFSETS' (*)

An offset table for the tree nodes. Each node in a GPI file is assigned a unique id (which is stored as an integer wherever nodes refer to other nodes). There are some special tree nodes (e.g., 'integer_type_node' or 'NULL_TREE') which are used very often and have fixed meanings. They have been assigned predefined ids, so they don't have to be stored in the GPI file at all. Their number and values are fixed (but may change between different GPC versions), see 'SPECIAL_NODES' in 'module.c'.

For the remaining nodes, the 'GPI_CHUNK_OFFSETS' table contains the file offsets as integers where they are stored within the (only) 'GPI_CHUNK_NODES' chunk. The offsets are relative to the start of that chunk, i.e. after the chunk header. After the table (but still in this chunk) the id of the main node which contains the list of all exported names is stored as an integer. (Currently, this is always the last node, but for the file format definition, this is not guaranteed.)

'GPI_CHUNK_IMPLEMENTATION'

This chunk contains no data (i.e., its size must be 0). Its only purpose is to signal that the module implementation or the implementation part of the unit has been compiled. (Stored, but not used currently.)

That's it. Now you should be able to "read" GPI files using GPC's '`--debug-gpi`' option. There is also a utility '`gpidump`' (built and installed with GPC, source code in the '`utils`' directory) to decode and show the contents of GPI files. It does also some amount of integrity checking (a little more than GPC does while loading GPI files), so if you suspect a problem with GPI files, you might want to run '`gpidump`' on them, discarding its standard output (it writes all error reports to standard error, of course).

If you encounter a case where the loaded information differs too much from the stored information, you have found a bug – congratulations! What "too much" means, depends on the object being stored in or loaded from the GPI file. Remember that the order things are loaded from a GPI file is the *reversed* order things are stored when considering *different* recursion levels, but the *same* order when considering the *same* recursion level. (This is important when using '`--debug-gpi`'; with '`gpidump`' you can read the file in any order you like.)

12.7 GPC's Automake Mechanism – How it Works

When a program/module/unit imports (uses) an interface, GPC searches for the GPI file (see Section 12.6 [GPI files], page 485) derived from the name of the interface.

Case 1: A GPI file was found.

Each GPI file contains the name of the primary source file (normally a '`.pas`' or '`.p`' file) of the module/unit, and the names of all interfaces imported. GPC reads this information and invokes itself with a command like

```
gpc foo.pas -M -o foo.d
```

This means: preprocess the file, and write down the name of the object file and those of all its source files in '`foo.d`'. GPC reads '`foo.d`' and looks if the object file exists and if the source was modified since the creation of the object file and the gpi file. If so, GPC calls itself again to compile the primary source file. When everything is done, the '`.d`' file is removed. If there was no need to recompile, all interfaces imported by the module/unit are processed in the same way as this one.

Case 2: No GPI file was found.

In this case, GPC derives the name of the source file from that of the interface by trying first '`interface.p`', then '`interface.pas`'. This will almost always work with UCSD/Borland Pascal units, but not always with Extended Pascal modules. The programmer can override this assumption using '`uses ... in`' or '`import ... in`'.

All this is done by the function '`gpi_open()`' which uses some auxiliary functions such as '`module_must_be_recompiled()`' and '`compile_module()`'.

Each time an object file is compiled or recognized as being up-to-date, its name is stored in a temporary file with the same base name as all the other temporary files used by GPC but the extension '`.gpc`'. When the top-level '`gpc`' is invoked (which calls '`gpc1`' later on), it passes the name of this temporary file as an additional command line parameter to '`gpc1`'. After compilation has been completed, the top-level '`gpc`' reads the temporary file and adds the new object files to the arguments passed to the linker.

The additional command '`--amtmpfile`' (not to be specified by the user!) is passed to child GPC processes, so all compiles use the same temporary file.

The source for this is merely in '`module.c`', but there are also some hacks in '`gpc.c`', additional command line options in '`lang-options.h`' and '`options.c`', and '`gpc.h`' contains declarations for the functions and global variables.

12.8 Files that make up GPC

The GNU back end (gbe) is used to convert RTL into assembler code. It is supposed to be language independent. Files are in the '`..`' directory (i.e., the directory called '`gcc`'). It also uses files in the '`../config`' subdirectories etc.

Unfortunately, some of them are not completely language independent and need patching for GPC. These patches (against all supported GCC versions) are in the '`diffs`' subdirectory.

The Pascal language implementation files are in the directory called '`p`'. Some of them were written from scratch. Others are hacked from GCC sources. Their roots, if any, are mentioned in the comment at their top.

12.9 Planned features

AnyStrings

```
GetCapacity (s):
  LongString            : s.Capacity
  UndiscriminatedString : MaxInt
  ShortString           : High (s)
  FixedString           : High (s) - Low (s) + 1
  CString (Array)       : High (s) - Low (s)
  CString (Zeiger)      : strlen (s)
  ObjectString          : s.GetCapacity

GetLength (s):
  LongString            : s.Length
  UndiscriminatedString : s.Length
  ShortString           : Ord (s[0])
  FixedString           : c := High (s);
                          while (c >= Low (s)) and (s[c] = ' ') do
                            Dec (c);
                          c - Low (s) + 1
  CString               : strlen (s)
  ObjectString          : s.GetLength

SetLength (s,n):
  if n > GetCapacity (s) then
    if TruncateFlag then
      n := GetCapacity (s)
    else
      Error;
  LongString            : s.Length := n
  UndiscriminatedString : if n > s.Capacity then
                            begin
                              tmp := @s;
                              { possibly round n up to m * 2^k
                                to avoid frequent reallocations }
                              New (@s, n);
                              Move (tmp^[1], s[1], Length (tmp^);
                              Dispose (tmp)
                            end;
```

```
                         s.Length := n
    ShortString          : s[0] := Chr (n)
    FixedString          : FillChar (s[Low (s) + n],
                             GetCapacity (s) - n, ' ')
    CString              : s[n] := #0
    ObjectString         : s.SetLength (n)

  GetFirstChar (s):
    LongString           : @s[1]
    UndiscriminatedString : @s[1]
    ShortString          : @s[1]
    FixedString          : @s[Low (s)]
    CString              : s
    ObjectString         : s.GetFirstChar
```

Anything else can be reduced to these, e.g. string assignment:

```
  SetLength (Dest, GetLength (Src));
  Move (GetFirstChar (Src) ^, GetFirstChar (Dest) ^, GetLength (Dest));
                                                                ^^^^
                                             (because of truncate!)
```

Note pointer CStrings because assignments to them (from long, undiscriminated (with appending #0) or CStrings, not from short, fixed or object strings) should set the pointer, not overwrite the memory pointed to.

Fully automatic C header translator

- C operators like '`+=`' (increment a variable and return the new value), or '`/`' (integer or real division, depending on the arguments). They could be emulated by special built-in functions in GPC which do the same . . .
- Types! C doesn't distinguish between pointers and arrays – and various other "jokes". E.g., a '`CString`' and a pointer to an array of bytes can both be '`char *`' in C. Solutions could be to introduce "special types" in GPC which behave like the C types (not so nice . . .)-:, or to let the translator choose one possible matching GPC type (by some heuristics perhaps), and leave it up to the user to type-cast when necessary (also not nice)-: . . .
- Name clashes. How to map '`foo`', '`FOO`', '`struct foo`', '`union foo`' etc. (which can potentially be totally different things in C) to Pascal identifiers in a reasonable way. Also, how to introduce identifiers for types when needed (e.g., typed used in parameter lists). Of course, that's solvable . . .
- Macros. Since GPC has a preprocessor, we can translate most of them, but some particularly strange ones are virtually impossible to translate. But there's hope that such strange macros are not being used in the libraries' headers . . .
- . . .

Appendix A GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place – Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

GPL Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year   name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items – whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Appendix B GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright © 1991, 1999 Free Software Foundation, Inc.
59 Temple Place – Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence the
version number 2.1.]

LGPL Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that

you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

   b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:
   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.
14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH

ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the library's name and an idea of what it does.
Copyright (C) year  name of author

This library is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2.1 of the License, or (at
your option) any later version.

This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307,
USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the library
'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

Appendix C DEMO COPYING

This is the common copying notice for all files found in '`demos/`' and '`docdemos/`' (unless stated otherwise in the file itself). They are distributed under the GNU General Public License with a notable exception:

Copyright (C) 1997-2005 Free Software Foundation, Inc.

Authors: See notice in the demo program. If not listed there, these are the authors of the GNU Pascal Compiler.

This demo program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This demo program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this demo program; see the file COPYING. If not, write to the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. Appendix A [Copying], page 491.

As a special exception, if you incorporate even large parts of the code of this demo program into another program with substantially different functionality, this does not cause the other program to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why it might be covered by the GNU General Public License.

Appendix D Contributors to GNU Pascal.

Jukka Virtanen
: invented GNU Pascal in March 1988, implemented the ISO 7185 and most of the ISO 10206 standard, etc.

Dr. Peter Gerwinski (`http://www.gerwinski.de/~peter/`)
: added Borland Pascal related and other extensions to GNU Pascal in summer 1995, ported GPC to EMX, did most of the development of the compiler from 1996 to 2001, created the WWW home page, etc.

Jan-Jaap van der Heijden
: ported GPC to DJGPP and to Microsoft Windows 95/NT, added ELF support in spring 1996, solved a lot of configuration and compatibility problems, created the GPC FAQ, etc.

Frank Heckenbach (`http://fjf.gnu.de`)
: rewrote and maintains the Run Time System since July 1997, does most of the development of the compiler since July 2001, wrote most of the units, demo programs, scripts and utilities distributed with GPC, wrote many test programs, maintains the GPC To-Do list (see Chapter 11 [To Do], page 465) and the WWW home page, etc.

Waldek Hebisch
: made GPC compatible with gcc-3.1.1 and later backend versions, fixed backend problems on various targets and improved the frontend in the areas of structured initializers, qualified identifiers, etc.

Prof. Abimbola A. Olowofoyeku ("The African Chief") (`http://www.bigfoot.com/~african_chief/`)
: created the original versions of many BP compatibility units in May 1997, contributed code to other units and the Run Time System, helped porting GPC and the units to Cygwin, mingw and MSYS, wrote a number of test programs, contributed a Borland Delphi-compatible '`SysUtils`' unit (`http://www.gnu-pascal.de/contrib/chief/`), etc.

Nicholas Burrett
: fixed some bugs and cleaned up GPC in May 1998, etc.

Dominik Freche
: improved and extended the GPC manual in August – September 1999 and wrote conversion routines for Borland compatible 6 Byte floating point numbers in December 1999.

Alexey Volokhov
: improved the performance of GPC's module/unit support in June 1997.

Bill Currie
: implemented more Borland extensions into GPC in July 1997.

Nicola Girardi
: wrote the GNU Pascal Coding Standards in November 2001, contributed a GPC unit (`http://www.gnu-pascal.de/contrib/nicola/`) for the '`svgalib`' graphics library for some platforms in February 2000, provided some portability enhancements to the RTS.

Eike Lange
: wrote an internationalization unit, translated the GNU Pascal Coding Standards into German and worked on the documentation.

Mirsad Todorovac
: translated the GPC documentation into Croatian and contributed code to the run time system.

Francisco Javier Fernandez Serrador
: translated the GPC documentation into Spanish.

Maurice Lombardi
: maintains the DJGPP port of GPC, improved the numerical routines for real and complex numbers and improved and extended the GMP real routines.

Emil Jerabek
: improved the numerical routines for real and complex numbers.

Adriaan van Os
: helped with the port of GPC to Mac OS X and set up a web site (`http://www.microbizz.nl/gpc.html`) with sources, binaries, patches and building instructions for this platform, and helped with Mac Pascal dialect support in GPC, since January 2003.

Neil Santos

James A. Morrison
: helps porting GPC to GCC-4.x backends since February 2005.

Russell Whitaker
: updated and maintained the GNU Pascal FAQ. (see Chapter 3 [FAQ], page 15)

Matthias Klose
: integrated GPC into EGCS and Debian GNU/Linux in May 1998, improved the installation process, etc.

Peter N Lewis
: added support for Mac Pascal dialect specific features and improved the documentation.

Orlando Llanes
: provided some contributions to the manual in May 1998.

The development of GNU Pascal profits a lot from independent contributions:

Anja Gerwinski
: maintains the GPC mailing list, since September 1999.

Berend de Boer
: wrote a lot of useful documentation about Extended Pascal in 1995.

Markus Gerwinski (`http://www.gerwinski.de/~markus/`)
: created the drawing showing a Gnu with Blaise Pascal and helped to design the WWW home page in October 1996.

Eike Lange
: is writing a book (`http://www.gnu-pascal.org/~eike/`) about GPC in German since March 2003.

Eike Lange
: wrote units (`http://www.gnu-pascal.de/contrib/eike/`) to access MySQL, GNU DBM and PostgreSQL databases in August 2000, and a unit (now part of GPC) and tools for internationalization in October – December 2001.

Eike Lange and Nicola Girardi
: together contributed a set of GTK units (`http://www.gnu-pascal.de/contrib/nicola/`) in February – May 2001.

Nicola Girardi
: wrote the GNU Pascal Coding Standards in English. Eike Lange translated them to German.

Prof. Phil Nelson
: created a bug reporting system for GPC in October 1996.

Robert Hhne
: wrote RHIDE (`http://www.rhide.com`), an integrated development environment for GNU compilers running under Dos (DJGPP) and Linux, and added support for GNU Pascal in autumn 1996.

Sven Hilscher
: wrote a mostly BP compatible '`Graph`' unit for several platforms in December 1996, now part of the GRX library (`http://www.gnu.de/software/grx/`).

Dario Anzani ("Predator Zeta") (`http://www.geocities.com/SiliconValley/Vista/6573`)
: contributed documentation about the use of assembler in GNU Pascal in May 1997. (see Section 7.2.3 [Assembler], page 238)

Dieter Schmitz
: set up a German mailing list for GPC, Section 10.1 [Mailing List], page 459, in March 2001.

(−:−−−−−−−−−−:−)
: This space is reserved for *your* name. ;−) Please contact us at the GPC mailing list, Section 10.1 [Mailing List], page 459, if you have something interesting for us.

We thank everybody who supports us by reporting bugs, providing feedback, contributing knowledge and good ideas, donating development tools, and giving us the opportunity to test GPC on a large variety of systems. We are particularly indebted (in alphabetical order, individuals first) to

Sietse Achterop, Jawaad Ahmad, Montaz Ali, Jamie Allan, Strobe Anarkhos, John P. R. Archer, Phil Armsdon, Geoffrey Arnold, Artur Bac, Steven J. Backus, Geoff Bagley, Andy Ball, Uwe Bauermann, Silvio a Beccara, Michael Behm, Ariel Bendersky, Pablo Bendersky, John Blakeney, Nicolas Bley, Philip Blundell, Preben Mikael Bohn, Ernst-Ludwig Bohnen, Nils Bokermann, Francesco Bonomi, J. Booij, Patrice Bouchand, Jim Brander, Frank Thomas Braun, Matthias Braun, Marcus Brinkmann, Steve Brooker, Doug Brookmann, J. David Bryan, Kev Buckley, Jason Burgon, Ricky W. Butler, Dr. E. Buxbaum, Andrew Cagney, Loris Caren, Theo Carr-Brion, Fernando Carrilho, Larry Carter, Fabio Casamatta, Janet Casey, Romain Chantereau, Emmanuel Chaput, Jean-Pierre Chevillard, Carl Eric Codere, Jean-Philippe Combe, Paolo Cortelli, F. Couperin, Nicolas Courtel, Miklos Cserzo, Tim Currie, Serafim Dahl, Paul Davidson, Martin G. C. Davies, Stefan A. Deutscher, Jerry van Dijk, Thomas Dunbar, Andreas Eckleder, Stephan Eickschen, Frank D. Engel Jr., Sven Engelhardt, Klaus Espenlaub, Toby Ewing, Chuck B. Falconer, Joachim Falk, Irfan Fazel, Carel Fellinger, Francisco Javier Fernandez, Christopher Ferrall, David Fiddes, Alfredo Cesar Fontana, Kevin A. Foss, B. Gayathri, Marius Gedminas, Philip George, Nicholas Geovanis, Jose Oliver Gil, Thorsten Glaser, Jing Gloria, Roland Goretzki, Morten Gulbrandsen, Gerrit P. Haase, Kocherlakota Harikrishna, Joe Hartley, Hans Hauska, Jakob Heinemann, Boris Herman, Arvid Herzenberg, Thorsten Hindermann, Honda Hirotaka, Stephen Hurd, Nick Ioffe, Mason Ip, Fredrik Ismyren, Richard D. Jackson, Daniel Jacobowitz, Grant Jacobs, Andreas Jaeger, Frank Jahnke, David James, Nathalie Jarosz, Sven Jauring, Niels Kristian Bech Jensen, Johanna Johnston, Achim Kalwa, Christine Karow, Tim Kaulmann, Thomas Keller, Clark Kent, Victor Khimenko, Russell King, Niels Ole Staub Kirkeby, Prof. Donald E. Knuth, Michael Kochiashvili, Tomasz Kowaltowski, David Kredba, Peter Ulrich Kruppa, Jochen Kuepper, Casper ter Kuile, Oliver Kullmann, Krzysztof Kwapien, Randy Latimer, Bernard Leak, Olivier Lecarme, Wren Lee, Martin Liddle, Kennith Linder, Stephen Lindholm, Orlando Llanes, Miguel Lobo, Benedict Lofstedt, Steve Loft, John

Logsdon, Dominique Louis, Dmitry S. Luhtionov, Jesper Lund, Martin Maechler, Muhammad Umer Mansoor, Claude Marinier, Ingvar Marny, Antony Matranga, Michael McCarthy, Michael Meeks, Clyde Meli, Axel Mellinger, Bryan Meredith, Jeff Miller, John Miller, Russell Minnich, Rudy Moddemeijer, Jason Moore, Scott A. Moore, Jeffrey Moskot, Pierre Muller, Adam Naumowicz, Nathanael Nerode, Andreas Neumann, Christian Neumann, Peter Norton, Adam Oldham, Gerhard Olejniczak, Alexandre Oliva, John G. Ollason, Marius Onica, Ole Osterby, Klaus Friis Ostergaard, Jean-Marc Ottorini, Michael Paap, Gale Paeper, Matija Papec, Miguel A. Alonso Pardo, Laurent Parise, Andris Pavenis, Robert R. Payne, Opie Pecheux, Jose M. Perez, Ronald Perrella, Bjorn Persson, Per Persson, Michael Pfeiffer, Pierre Phaneuf, Pascal Pignard, Tam Pikey, Nuno Pinhao, Philip Plant, Larry Poorman, Stuart Pope, Yuri Prokushev, Huge Rademaker, Shafiek Rasdien, Mike Reid, Leon Renkema, John L. Ries, Phil Robertson, Clive Rodgers, Jim Roland, Guillaume Rousse, Daniel Rudy, Marten Jan de Ruiter, Martin Rusko, Sven Sahle, Neil Santos, Carl-Johan Schenstrom, Robert B. Scher, Hartmut Schmider, Thomas D. Schneider, Dominique Schuppli, Egbert Seibertz, George Shapovalov, Richard Sharman, Patrick Sharp, Joe da Silva, Arcadio Alivio Sincero, Ian Sinclair, Kasper Souren, Tomas Srb, Anuradha Srinivasan, David Starner, Andrew Stribblehill, Alan Sun, Veli Suorsa, Matthew Swift, Mark Taylor, Paul Tedaldi, Robin S. Thompson, Ian Thurlbeck, Gerhard Tonn, Ivan Torshin, Bernhard Tschirren, Josef Urban, Luiz Vaz, Tom Verhoeff, Kresimir Veselic, Jean-Pierre Vial, Alejandro Villarroel, Bohdan Vlasyuk, Marco van de Voort, Raymond Wang, Nic Webb, Peter Weber, Francisco Stefano Wechsler, Christian Wendt, Benedikt Wildenhain, Gareth Wilson, Marc van Woerkom, David Wood, Michael Worsley, Takashi Yamanoue, George L. Yang, Salaam Yitbarek, Dafi Yondra, Eli Zaretskii, Artur Zaroda, Gerhard Zintel, Mariusz Zynel, the *BIP* at the *University of Birmingham*, UK, the *Institut fuer Festkoerperforschung (IFF)* at the *Forschungszentrum Juelich*, Germany, CARNet (Croatian Academic and Research NETwork), the Academy of Fine Arts and the Faculty of Graphic Arts at the University of Zagreb, Croatia, and everybody we might have forgotten to mention here. Thanks to all of you!

GNU Pascal is based on GNU CC by Richard Stallman. Several people have contributed to GNU CC:

- The idea of using RTL and some of the optimization ideas came from the program PO written at the University of Arizona by Jack Davidson and Christopher Fraser. See “Register Allocation and Exhaustive Peephole Optimization”, Software Practice and Experience 14 (9), Sept. 1984, 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the Vax machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Support wrote the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series cpu, the SPARC cpu and part of the Motorola 88000 cpu.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Randy Smith finished the Sun FPA support.
- Robert Brown implemented the support for Encore 32000 systems.
- David Kashtan of SRI adapted GNU CC to VMS.
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.

- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Alain Lichnewsky ported GNU CC to the Mips cpu.
- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Gary Miller ported GNU CC to Charles River Data Systems machines.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination.
- Richard Kenner and Michael Tiemann jointly developed reorg.c, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Mike Meissner at the Open Software Foundation finished the port to the MIPS cpu, including adding ECOFF debug support, and worked on the Intel port for the Intel 80386 cpu.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Torbjorn Granlund implemented multiply- and divide-by-constant optimization, improved long long support, and improved leaf function register allocation.
- Mike Stump implemented the support for Elxsi 64 bit CPU.
- John Wehle added the machine description for the Western Electric 32000 processor used in several 3b series machines (no relation to the National Semiconductor 32000 processor).

Appendix E Resources For Use With GPC.

Many of the programs mentioned here, plus some more, can be found at

`http://www.gnu-pascal.de/contrib/`

GNU Pascal Drawing

On the web you can find our GNU Pascal drawing as a small (1 KB) ('`images/GnuPascal-small.png`' on GPC's WWW home page) and a large PNG image (10 KB) ('`images/GnuPascal.png`' on GPC's WWW home page), as an EPS file (45 KB) ('`images/GnuPascal.eps`' on GPC's WWW home page), and as a PDF file (18 KB) ('`images/GnuPascal.pdf`' on GPC's WWW home page).

Due to patent problems (`http://www.gnu.org/philosophy/gif.html`) we do not use GIF files. Fortunately the PNG format (`http://www.libpng.org`), the successor of GIF, does not have this problem – and introduces better compression and more advanced features anyway.

By the way, the color gradient ('`images/gradient.png`' on GPC's WWW home page) that makes our page background is a PNG file of only 632 bytes. It was generated by a Pascal program compiled with GPC and '`pnmtopng`'. [Example (gradient.pas)]

PENG

PENG is an integrated development environment (IDE) for GNU Compilers and other purposes on any platform supported by GPC, written by Frank Heckenbach. The home page of PENG is

`http://fjf.gnu.de/peng/.`

RHIDE

RHIDE is an integrated development environment (IDE) for GNU Compilers on DOS (DJGPP) or Linux, written by Robert Hhne. The home page of RHIDE is

`http://www.rhide.com.`

DevPascal

DevPascal is an integrated development environment (IDE) for GNU Pascal on mingw32. The home page of DevPascal is

`http://www.bloodshed.net/devpascal.html.`

GRX

GRX is a graphics library for C and GNU Pascal, including a mostly BP compatible 'Graph' unit. It is available from

```
http://www.gnu.de/software/grx/
```

Although GRX originated on DJGPP, a DOS programming platform, it is portable to Linux with SVGAlib, to all Unix-like systems running the X11 window system, and to MS-Windows 9x/NT.

Internationalization

Units and tools for internationalization are available in

```
http://www.gnu-pascal.de/contrib/eike/
```

Database units

GNU Pascal units to access MySQL, GNU DBM and PostgreSQL databases are available in

```
http://www.gnu-pascal.de/contrib/eike/
```

GTK units

GNU Pascal units for the GTK+ and GTK+ GL libraries are available in

```
http://www.gnu-pascal.de/contrib/nicola/
```

Documentation

A book about GPC manual in German written by Eike Lange can be found in

```
http://www.gnu-pascal.org/~eike/
```

SysUtils unit

Prof. Abimbola A. Olowofoyeku ("The African Chief") wrote a Delphi-compatible (though a few routines are still missing) 'SysUtils' unit. It has been tested under Cygwin, mingw, Linux (Mandrake 7.0), and Solaris 7. It can be downloaded from http://www.gnu-pascal.de/contrib/chief/.

Crystal, a mailing list archive program

Crystal is a web based mailing list archive, written for GNU Pascal and used for the archives of GPC's mailing lists (see Section 10.2 [Mailing List Archives], page 460). The source code can be found at http://fjf.gnu.de/crystal/.

ISO standards

The Pascal standard specifications are available in PostScript format at

```
http://ftp.digital.com/pub/Digital/Pascal/
```

Alternative addresses are

```
ftp://ftp.europe.digital.com/pub/DEC/Pascal/
ftp://ftp.digital.com/pub/DEC/Pascal/
```

There are also copies at

```
http://www.moorecad.com/standardpascal/iso7185.ps
(ISO 7185 Pascal)
```

```
http://www.moorecad.com/standardpascal/iso10206.ps
(ISO 10206 Extended Pascal)
```

Note: These documents are a bit hard to navigate (e.g., in ghostview) because they are missing the so called "document structuring comments" (DSC). The GPC source distribution contains a little script '`ps2dsc`' to add the DSC again and make the documents easier to navigate. Note that for reasons of copyright, you are probably only allowed to do this for your own use and not to distribute the modified files.

You can find an easy-to-read introduction to Extended Pascal by Prospero Software at

```
http://www.prosperosoftware.com/epintro.html
```

Please note that Standard Pascal is **not** the same as Borland Pascal nor a subset of it. See Chapter 1 [Highlights], page 5 for examples of Standard Pascal features that are missing in Borland Pascal.

Scott A. Moore's ANSI-ISO Pascal FAQ (132 KB) (`http://www.moorecad.com/standardpascal/ansiiso.faq`) discusses the differences between both dialects in detail.

The draft standard "Object-Oriented Extensions to Pascal" can be found at

```
http://pascal-central.com/OOE-stds.html
```

Software Patents Kill Innovation

Programming activities of small companies and individuals are threatened by software patents. If you are a programmer, you are in danger, too! Your employer or yourself might be sued by a large company holding a patent on some *ideas* you are using in your programs. (You need not use foreign code in order to become vulnerable.)

For more information look at

```
http://swpat.ffii.org (Europe)
http://lpf.ai.mit.edu (USA)
```

Appendix F The GNU Project.

GNU Pascal is part of the GNU project which was founded by Richard Stallman in 1984. The aim of the GNU project is to provide a complete operating system with editors, compilers etc. as *Free Software*.

People often confuse *Free Software* with *public domain software* or have other wrong information about the GNU project. If you want to know it definitely, please read the **GNU General Public License** (`http://www.gnu.org/copyleft/gpl.html`).

For even more information, please consult the official **GNU home page** of the *Free Software Foundation (FSF)*, `http://www.gnu.org/` or one of its mirror sites.

Some small notes about common misunderstandings follow.

- It is legal to compile commercial, including non-free, programs written in Pascal with GNU Pascal. They do *not* automatically become Free Software themselves.
- "Free" is opposed to "proprietary", but *not* opposed to "commercial". Free Softare can be – and is in fact – distributed commercially for a real price. In contrast, most non-commercial software does *not* meet the open source criteria (`http://www.opensource.org/osd.html`) and thus does *not* qualify as Free Software.
- When you modify a free program released under the **GNU General Public License** (`http://www.gnu.org/copyleft/gpl.html`), e.g. the GNU Pascal compiler itself:
  - You can released and license your own modifications (as a separate entity) any way you like.
  - If you distribute or publish the whole work with your modifications, including or derived from GPL licensed parts, the whole work (including your modifications) must be licensed under the terms of the GPL. (Note: This does not contradict the previous point since you can license your work in several ways.)
  - You do not have to distribute or publish the whole work at all. In this case the question of license of the whole work does not arise.

  Please note: These are informal explanations which should not be construed as legal advice. The legally binding text is only the text contained in the license statement.
- When using libraries for writing proprietary programs, check the libraries' licenses carefully. The *GNU Lesser General Public License* (`http://www.gnu.org/copyleft/lesser.html`) allows linking a library to non-free software under certain conditions, the ordinary *GNU General Public License* does not.
- It is legal to charge a fee for distributing Free Software. If somebody sold you a copy of GNU Pascal you could have got without paying for it as well, that's in agreement with the *GNU General Public License*.
- However if somebody wants you to sign an agreement that you won't re-distribute the Free Software you have got, it would be illegal. That person would lose the right to use and distribute that Free Software.
- The preferred form to distribute Free Software is in source code. This ensures that everybody has the freedom to customize the software or to fix bugs by themselves. When we also distribute GNU Pascal binaries we do it only to simplify installation and to encourage its use.

F.1 The GNU Manifesto

The GNU Manifesto which appears below was written by Richard Stallman at the beginning of the GNU project, to ask for participation and support. For the first few years, it was updated in minor ways to account for developments, but now it seems best to leave it unchanged as most people have seen it.

Since that time, we have learned about certain common misunderstandings that different wording could help avoid. Footnotes added in 1993 help clarify these points.

For up-to-date information about the available GNU software, please see the latest issue of the GNU's Bulletin. The list is much too long to include here.

F.1.1 What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it.[1] Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TEX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the 'G' in the word 'GNU' when it is the name of this project.

F.1.2 Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

---

[1] The wording here was careless. The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies – and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

F.1.3 Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

F.1.4 How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

F.1.5 Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

F.1.6 How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

F.1.7 Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.[2]

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

F.1.8 Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.[3]

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for

---

[2] This is another place I failed to distinguish carefully between the two different meanings of "free". The statement as it stands is not false – you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.

[3] Several such companies now exist.

most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."
>
> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?[4]

> "My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.[5]

> "Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

> "Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

---

[4] The Free Software Foundation raises most of its funds from a distribution service, although it is a charity rather than a company. If *no one* chooses to obtain copies by ordering from the FSF, it will be unable to do its work. But this does not mean that proprietary restrictions are justified to force every user to pay. If a small fraction of all the users order copies from the FSF, that is sufficient to keep the FSF afloat. So we ask users to choose to support us in this way. Have you done your part?

[5] A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented – books, which could be copied economically only on a printing press – it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

> "Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies – such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

> "Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing – often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

F.2 Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers – the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU C compiler contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright (C) 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted
without royalty; alteration is not permitted.

Index-GPC

*

-

/

=

+

>

<

A

B

C

M

N

O

P

Q

R

S

T

U

V

W

X