
TLIN レファレンス・コード

(LIN コンソシアム仕様に基づくレファレンス・コード)

ユーザ・ガイド

Revision 1.1

2007 年 3 月 1 日

株式会社 OTSL



Edition History

発行日	バージョン	作成者・加筆者	改版の内容
2006/03/09	1.0	日野	初版
2007/01/31	1.1	日野	● 図表番号索引追加 ● Bus No Activity イベントの定義変更 ● Physical Bus Error イベントの追加 ● Synch Break Delimiter イベントの削除 ● Max Byte イベント記述の追加

目次

1. はじめに	9
1.1. 参照仕様書	9
1.1.1. LIN コンソシアム仕様	9
1.2. ソフトウェア・プラット・フォーム	9
1.2.1. 言語環境	9
1.2.2. OS 環境	10
1.3. ハードウェア環境	10
1.3.1. マイコン	10
1.3.2. ライン・ドライバ	10
2. TLIN REFERENCE CODE を用いた開発フロー	11
2.1. LDF ツール	12
2.2. エンティティ	13
2.3. ユーザ API	14
2.4. TLIN フレーム管理	14
2.5. LIN プロトコル・スタック	14
2.6. HAL	14
2.7. OSAL	14
3. TLIN API の概要	15
3.1. TLIN REFERENCE CODE で使用されるデータ型	15
3.2. マルチ・クラスタ対応	16
3.2.1. インタフェース・ハンドルとレスポンス・ハンドル	16
3.3. シグナル・ハンドル	17
3.3.1. NULL ハンドルの意味	18
3.4. 標準 API	19
3.4.1. システムとインタフェースの初期化／シャットダウン	19
3.4.2. スケジュール設定と駆動(マスタのみ)	21
3.4.2.1. tlin_ifc_sch_set(スケジュール設定関数)	22
3.4.2.2. tlin_ifc_sch_tick(スケジュール駆動関数)	22
3.5. シグナル・アクセス API	24
3.6. NETWORK MANAGEMENT API	25
3.7. ダイアグ API	26
4. TLIN REFERENCE CODE のタスク構成	28
4.1. TLIN PROTOCOL STACK のタスク構成	28
5. HAL	29

5.1. ハードウェアのモデル	29
5.2. イベント設計	29
5.3. HAL のチャンネル	30
6. OSAL	31
付録 A	32
1. 設計プロセスのモデル	32
1.1. プロセス・フローの概要	32

図表索引

図 1	TLIN Reference Code を用いた Application 構成図	11
図 2	LDF ツールによるエンティティ・ソース・コードの生成	12
図 3	エンティティを介したデータ通信	13
図 4	TLIN Reference Code のデータ型	15
図 5	LIN の Multi Cluster 接続	16
図 6	インタフェース・ハンドル, レスポンス・ハンドルの定義文の例	17
図 7	シグナル・ハンドルの定義文の例	17
図 8	その他のハンドルの定義文の例	18
図 9	システム初期化の状態遷移	19
図 10	初期化の例	20
図 11	スケジュール設定の例	22
図 12	tlm_ifc_sch_tick の駆動タイミング	23
図 13	Master Node の Diag シーケンス	26
図 14	Slave Node の Diag シーケンス	27
図 15	TLIN Reference Code のタスク構成	28
図 16	インタフェース・ハンドルと HAL チャンネル割付テーブル	30
図 17	設計プロセス・フロー	33

1. はじめに

この文書は、この文書とともに提供される『Toppers プロジェクトの一部として供給される、LIN コンソシアム仕様にに基づくレファレンス・コード』（以降、『*TLIN Reference Code*』と記す）の構造および設計の概要を述べるものである。

TLIN Reference Code は、実際のシングルチップ・マイクロコントローラ（以下、『マイコン』）の上で動作試験を行ったものであるが、あくまで、トヨタ標準 LIN 仕様書の動作説明を目的にして設計されたものである。

また、本書は、LIN コンソシアム仕様を参照する際に、LIN コンソシアムが規定する LIN ネットワークの動作を、イベント駆動型状態遷移モデルを用いて説明する事を、目的としている。

同時に、コンソシアム LIN 仕様に基づく ECU の実際のインプリメンテーションの参考資料として作成したものである¹。

1.1. 参照仕様書

1.1.1. LIN コンソシアム仕様

LIN コンソシアムが発行している“LIN Specification Package Revision 1.3 Dec. 12, 2002”²に含まれる“LIN Protocol Specification Revision 1.3 December 12, 2002”³（以下『*Spec 1.3*』と述べる）をベースにしている。

1.2. ソフトウェア・プラット・フォーム

1.2.1. 言語環境

TLIN Reference Code は、C 言語で記述されている。理由は、C 言語が現在組み込みシステムにおける最も標準的な言語であることと、同時に安価で多くのソフトウェア・ベンダから開発環境が提供されていることである。従って、言語は C 言語である必要は無く、また C コンパイラや、それに付随する開発環境 (IDE など) の種類やバージョンを問わない。

なるべく多くの読者の都合を考え、C 言語の記述は、ANSI C に準拠している。

本 *TLIN Reference Code* で使用したコンパイル環境は、Renesas M3T-NC30WA V.5.10 Release 1 Entry (C compiler)と、KD30 V.3.30 Release 1 (Remote Debugger)である。

¹ 参照資料であることに注意。インプリメンテーションは、使用するマイコンなどの部品や、ECU に要求される仕様などにより、異なるものである。

² <http://www.lin-subbus.org/>

³ コンソシアムの提供する LIN 仕様は、既に 2.0 になっているが、日本における実際のインプリメンテーションは、1.3 仕様が一般的であるので、これをベースにした。

1.2.2. OS 環境

OS は必ずしも使用しなくても良いものとする。しかし、プログラム・コードの再利用性や、マイナー・チェンジへの対応柔軟性を考えると、RTOS(Real Time Operating System)の使用が望ましい。

TLIN Reference Code は、基本的に、『イベント駆動型』の『ステート・マシン・モデル』の考え方で記述されている。本来、通信プログラムは『イベント駆動型』であることと、**TLIN**のネットワーク・マネージメントの記述が、『状態遷移』の考え方で記述されていることなどから、『イベント駆動型』の『ステート・マシン・モデル』が、最もなじむ手法であると考えられるためである。

RTOS としては、OSEC をイメージして記述されている。しかし、使用するサービス・コールは、イベント・フラグ、メール・ボックス、セマフォなどの基本的なものに留めた上、OSAL (Operating System Abstraction Layer)を設けたので、他の RTOS や、RTOS 無しのインプリメンテーションにも比較的簡単に適用可能である⁴。

1.3. ハードウェア環境

TLIN Reference Code を適用するハードウェアは、必要なペリフェラルを持つマイコンと LIN ライン・ドライバのみである。

1.3.1. マイコン

マイコンは、2005 年現在、組み込み市場で使われている平均的なマイコンをターゲットにしている。**TLIN Reference Code** では、最低限以下の仕様を満たすマイコンへの適用を推奨する。

内部 8 ビットまたは 16 ビット・データ・バス幅のマイコン(もちろん、それ以上のバス幅でも可)

内部バスクロック(CPU の1サイクル・クロック)で、8MHz~20MHz 程度

- プログラム・メモリ領域 32K バイト以上
- データ・メモリ領域(スタックを含む)4K バイト以上
- UART(Transmitter/Receiver のセット): 1
- 標準的な 16 bit timer⁵ : 2
- BUS monitor 用外部割込み端子: 1

上記は、あくまで目安とすること。アプリケーションにより、この値は大きく変わる。

本 **TLIN Reference Code** では、評価用に Renesas 製 M16C/6N, M32C/85 マイコンを使用した。

1.3.2. ライン・ドライバ

TLIN Reference Code では、ライン・ドライバの仕様については関知しない。**Spec 1.3** および **TLIN** で規定される特性を満たせば良いものとする。

⁴ 実際のところ、開発の初期においての動作検証は、 μ ITRON 3.0 を用いて行なった。

⁵ 現在市場に出ているマイコンのタイマには、多くの機能が内蔵されているが、本インプリメンテーションにおいては、レフェレンス・コードにより汎用性を与えるために、プリミティブな機能のみを使用する方針で行った。

2. TLIN Reference Code を用いた開発フロー

図 1 に TLIN Reference Code を用いた際の、アプリケーション・プログラムの構成を示す。

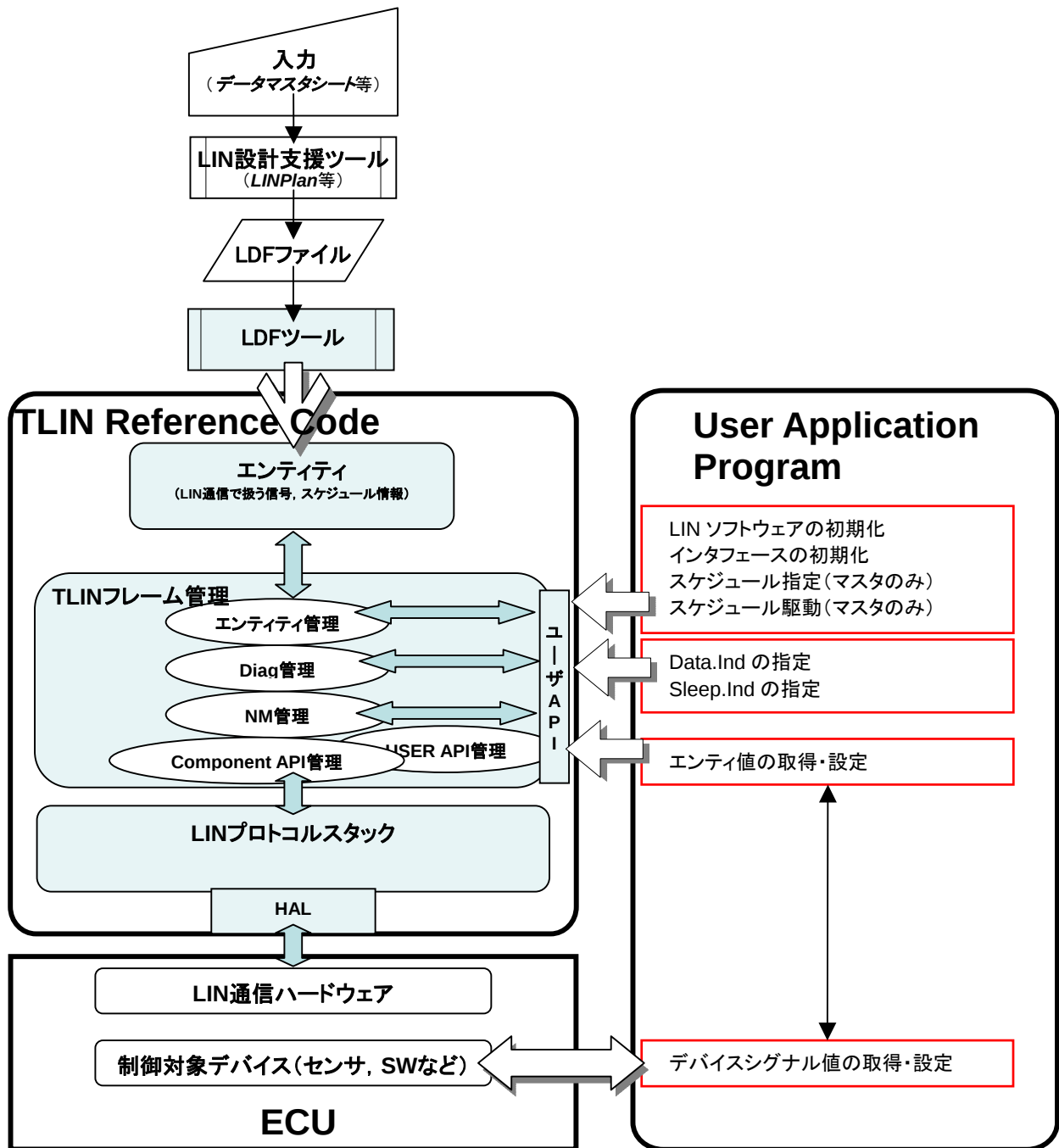


図 1 TLIN Reference Code を用いた Application 構成図

図 1 において、赤い四角で囲んだ部分が、ユーザが記述する部分である。図 1 の各要素について、以下に処理の概要を説明する。

2.1. LDF ツール

LIN ネットワークの設計においては、ECU ごとに、必要な情報 (LIN 用語では、シグナル) の記載を行い、データ・マスタ・シートとして作成する。データ・マスタ・シートの情報は、LIN バス・マネージャのもとに集められ、スケジュール・テーブルおよびビット・アサイン表などの作成により、LIN ネットワークの設計が行われる。

上記のバス・マネージャの作業は、全て手作業で行うこともあるが、通常は **LIN Plan**⁶などの LIN ネットワーク設計サポートツールを用いて行うのが通常である。**LIN Plan** は、出力のひとつとして、LDF (LIN Description File : LIN 記述ファイル) を生成するのが一般的である。

TLIN Reference Code の一部として提供する LDF ツールは、LDF ファイルを解析し、スケジュール・テーブル情報、フレーム情報、シグナル情報などを、C ソース・コード (*LDF_file_name.H*, *LDF_file_name.C*) として出力する⁷。ここで、“*LDF_file_name*” はユーザが LDF ツール使用時に指定するファイル名称です。

LDF_file_name.H には、スケジュール・テーブル、フレーム、シグナルのハンドル名称と値が定義される。*LDF_file_name.C* は、これらの情報テーブルの実体が記述される。これは、図 1 におけるエンティティの実体でもある。TLINType.h ヘッダファイルに、*LDF_file_name.C* で使用されるデータ・タイプが記述されている。

詳細については、“TLIN LDF Tool 仕様書”を参照すること。

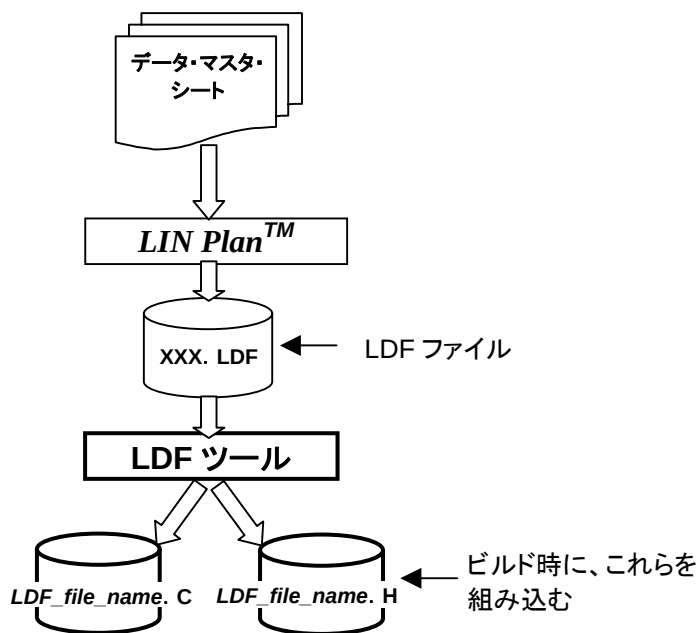


図 2 LDF ツールによるエンティティ・ソース・コードの生成

⁶ LINPlan は、TTTech 社のソフトウェア製品の登録商標です。

⁷ OSEC の oil をベースに作成した SG (System Generator) を供給する予定である。

2.2. エンティティ

エンティティとは、LIN 通信で送受信される信号(シグナル)や、スケジュール・テーブル情報、フレーム情報などの総称である。LDF ツールにより、LDF から自動生成される C ソース・コードをコンパイルすることにより、TLIN Reference Code の一部として取り込まれる。

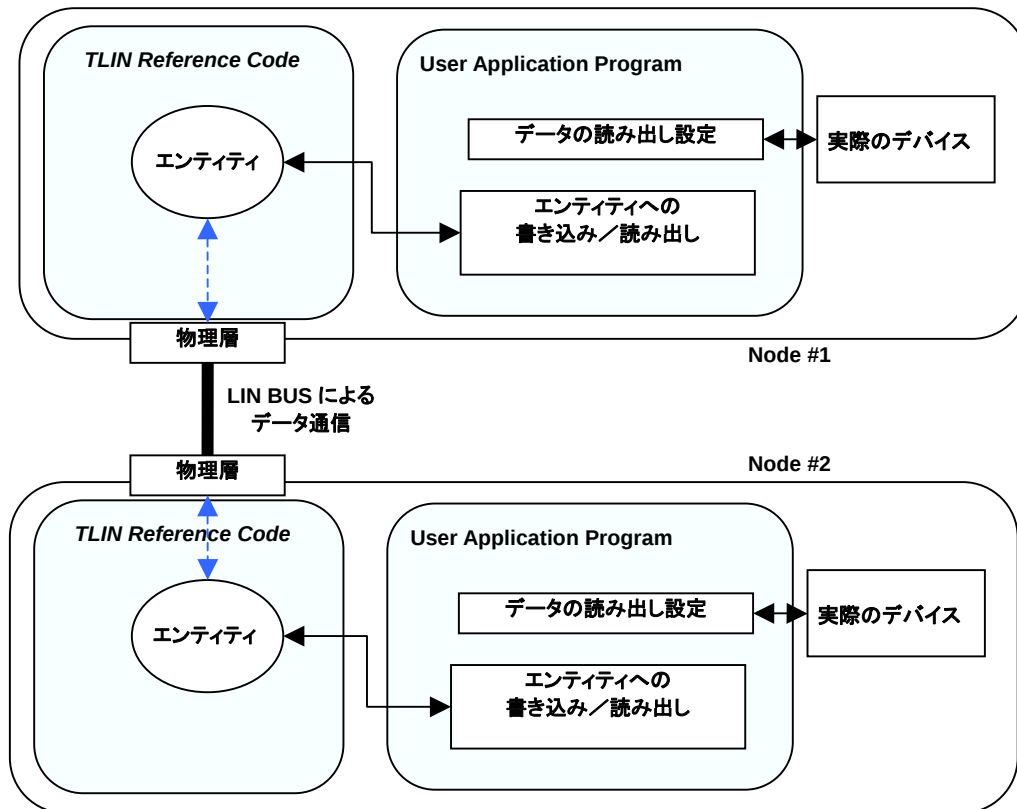


図 3 エンティティを介したデータ通信

図 3 にエンティティを介した LIN ノード間の情報交換のイメージを示す。

ノード間で情報交換が必要なデータは、**TLIN Reference Code** の内部で、(LDF.C によって)エンティティとして格納領域が確保されている。アプリケーション・プログラムでは、ユーザ API を用いて、対応するシグナル・ハンドル名 (LDF.H で定義されている) を用いて、エンティティへの書き込み／読み出しを行うのみである。

互いのノード間のエンティティ・データは、スケジュール・テーブルとフレームのビット・アサイン情報に従って、自動的にデータが交換される。

2.3. ユーザ API

ユーザ・アプリケーション・プログラムから、TLIN Reference Code の各機能ブロックおよびエンティティにアクセスする場合は、全て、このユーザ API を介して行われる。

ユーザ API は、TLIN Reference Code 内部で使用されるコンポーネント API と区別する場合のみ、この名称で使われる。通常 API と記述された場合は、ユーザ API を意味する。

ユーザ API の詳細については、“TLIN API 仕様書”を参照すること。

2.4. TLIN フレーム管理

2.5 項に述べる LIN プロトコル・スタックと呼応して、LIN ネットワークのデータ送受信を管理する。この部分は、特にトヨタ標準 LIN 仕様書で規定される、トヨタ独自のインプリメンテーション仕様を実現する。(トヨタ Network Management 仕様, Diag 仕様)

また、ユーザ API は、全て、この層を経由して行われる。

2.5. LIN プロトコル・スタック

純粋にコンソシアム仕様に基づく LIN 通信プロトコルを実現している部分である。OSI 7 Layer model で述べる 1 層(物理層), 2 層(データ・リンク層)のみを実現する。通信プログラムで言う MAC 層に対応する。

2.6. HAL

Hardware Abstraction Layer。LIN 通信に必要なハードウェア機能を抽象化し、ハードウェア依存の部分を、この層に閉じ込めている。

2.7. OSAL

Operating System Abstraction Layer。OS 機能を、この層に閉じ込めている。

3. TLIN API の概要

詳細は、『TLIN API 仕様書』に述べられるため、ここでは、TLIN API の一覧および、その使い方の指標を示す。

3.1. TLIN Reference Code で使用されるデータ型

TLIN Reference Code では、独自のデータ型が定義されており、TLinTypes.H に記述されている。以下、その記述を示す。

```
/* **** */
/* < TLinTypes.h > :      基本データ型定義      */
/* **** */

/* 基本型 */
typedef unsigned char  TLinU08;
typedef unsigned short TLinU16;
typedef unsigned long  TLinU32;
typedef unsigned char  TLinBool;

#define TLIN_BOOL_TRUE      0x01
#define TLIN_BOOL_FALSE    0x00

#define TLIN_STATUS_OK      0x00
#define TLIN_STATUS_NG     0xFF

#define TLIN_FLAG_SETED     0x01
#define TLIN_FLAG_CLEAR    0x00

/* ハンドル形(エンティティ識別用) */
typedef unsigned short TLinSigHnd; /* シグナル用 */
typedef unsigned short TLinSchHnd; /* スケジュール用 */
typedef unsigned short TLinIfcHnd; /* インタフェース用 */
typedef unsigned short TLinFrmHnd; /* フレーム用 */
typedef unsigned short TLinResHnd; /* レスポンスフレーム用 */
typedef unsigned short TLinSigLnkHnd; /* シグナル・オフセット用 */
typedef unsigned short TLinSchRcdHnd; /* スケジュールレコード用 */
```

図 4 TLIN Reference Code のデータ型

各ハンドル型については、以下に説明する。

3.2. マルチ・クラスタ対応

3.2.1. インタフェース・ハンドルとレスポンス・ハンドル

ある ECU では、LIN 端子が複数存在し、個々の LIN 端子が異なる LIN Network Cluster に接続される場合がある。(図 5 参照)

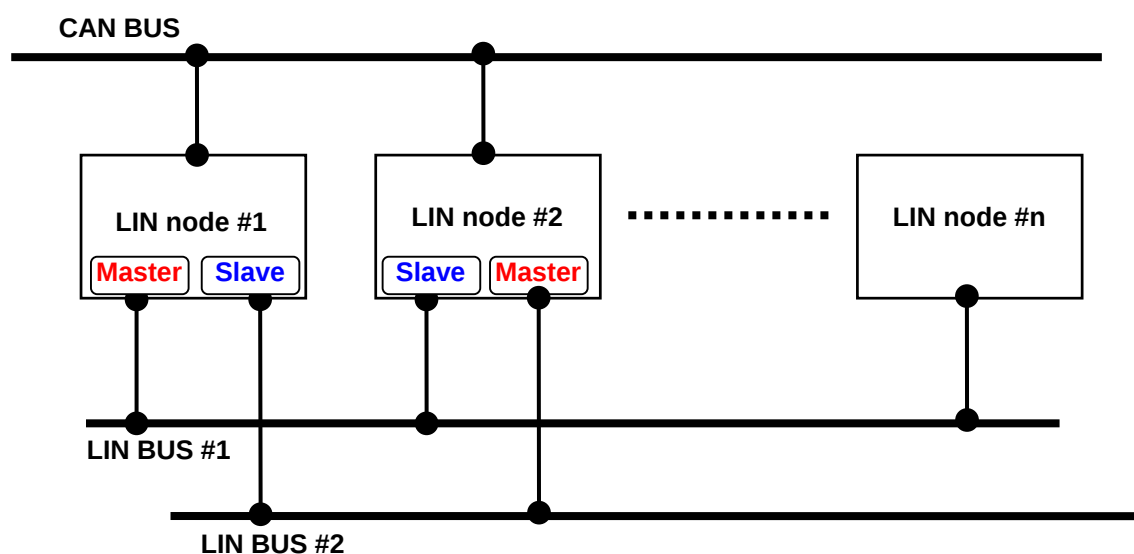


図 5 LIN の Multi Cluster 接続

TLIN Reference Code は、このような Multi-cluster 対応のために、インタフェース・ハンドル (Interface Handle) という概念を使用する。インタフェース・ハンドルは、個々の Cluster を識別するために用いられる Cluster ID である。

TLIN Reference Code 内部においては、インタフェース・ハンドルは独立して管理され、エンティティや、HAL の使用チャンネルも、各インタフェース・ハンドルごとに用意され、管理される。

例えば、あるノードを LIN BUS #1 に物理的に接続し、動作させることを、**TLIN Reference Code** では、『インタフェース・ハンドル#1 を connect する。』というように取り扱う。API.H ファイルにあるユーザ API 関数のプロトタイプは以下のようなものになっている。

```
TLinU08 tlin_ifc_connect(TLinIfcHnd hIfc);
```

実際のアプリケーション・プログラムで、ノードを LIN BUS #1 に接続するには、以下のような記述になる。

```
TLinU08 error_code;  
error_code = tlin_ifc_connect( TLIN_IFCHND_1 );
```

上記では、TLIN_IFCHND_1 は、LDF ツールが作成する LDF.H に #define 文として自動生成される。以下は、LDF.H のインタフェース・ハンドルと、下に説明するレスポンス・ハンドルの定義部分の例である。

```
#define TLIN_IFCHND_1 1
#define TLIN_RESND_1 1
#define TLIN_IFCHND_2 2
#define TLIN_RESND_2 2
```

図 6 インタフェース・ハンドル、レスポンス・ハンドルの定義文の例

上の記述で、TLIN_RESND_1, TLIN_RESND_2 という定義文がある。これは、レスポンス・ハンドル(Response Handle)の定義文である。

TLIN Reference Code が、通信を行う際には、エンティティ中のシグナルを、メッセージ・フレームのレスポンス・イメージ(2バイト/4バイト/8バイトのデータ配列)に展開する必要がある。**TLIN Reference Code** のエンティティ管理部は、各 LIN Network Cluster ごとに、レスポンス展開用のバッファを管理している。インタフェース・ハンドルと同様に、レスポンス・ハンドルによって、レスポンス展開用バッファは、独立に管理される。

3.1 項(図 4)に示した TLinIfcHnd と TLinResHnd は、この 2 つのハンドル型の定義である。

3.3. シグナル・ハンドル

3.2 項に述べたように LIN Network Cluster は、インタフェース・ハンドルとレスポンス・ハンドルによって識別される。

一方、エンティティ内部で管理されるシグナルは、LIN Network Cluster に関係無く、すべてのシグナルについて連番で管理される。

```
/*== LDF #1 signals ==*/
#define TLIN_SIGHND_NULL 0
#define TLIN_SIGHND_DSPW_MTR 1
#define TLIN_SIGHND_PSPW_MTR 2
#define TLIN_SIGHND_RSRPW_MTR 3
#define TLIN_SIGHND_RSLPW_MTR 4
#define TLIN_SIGHND_SRPW_MTR 5
#define TLIN_SIGHND_PW_MAIN_SW 6

/*== LDF #2 signals ==*/
#define TLIN_SIGHND_wiper_s_1 16
#define TLIN_SIGHND_wiper_s_2 17
#define TLIN_SIGHND_wiper_s_3 18
#define TLIN_SIGHND_wiper_s_4 19
#define TLIN_SIGHND_wiper_s_5 20
```

図 7 シグナル・ハンドルの定義文の例

インタフェース・ハンドルにより、物理的な LIN Network Cluster が識別可能であるため、エンティティのその他のリソース(スケジュール・テーブル、メッセージ・フレームなど)は、全て連番(間が跳ぶ場合はある)管理となる。以下にシグナル・ハンドル以外の LDF.H ファイルの定義文の例を示す。

#define TLIN_FRMHND_NULL	0
#define TLIN_FRMHND_Sensor1_out_1	1
#define TLIN_FRMHND_Sensor1_out_2	2
#define TLIN_FRMHND_Vehicle_State	3
#define TLIN_FRMHND_DiagFrame_M	4
#define TLIN_FRMHND_DiagFrame_S	5
#define TLIN_SCHHND_NULL	0
#define TLIN_SCHHND_Mode1	1
#define TLIN_SCHHND_Diag1	2

図 8 その他のハンドルの定義文の例

3.3.1. NULL ハンドルの意味

図 6, 7, 8 の定義文の最初は、必ず“**TLIN_XXXHND_NULL 0**”となっている。これは、NULL Handle と呼び、定義されていないハンドルであることを識別するものである。

API やエンティティ・アクセサ関数(コード内部使用で通常ユーザは使用しない)が、NULL Handle を返してきた場合は、そのノードに対して、指定されたリソースは存在しない(例えば、他ノードが反応すべきフレームなど)ことを意味する。

アプリケーション・プログラムでは、不用意に NULL Handle を用いてアクセスしないように注意する事。

3.4. 標準 API

3.4.1. システムとインタフェースの初期化／シャットダウン

以下にシステムとインタフェースの初期化に関する API 関数とプロトタイプを示す。

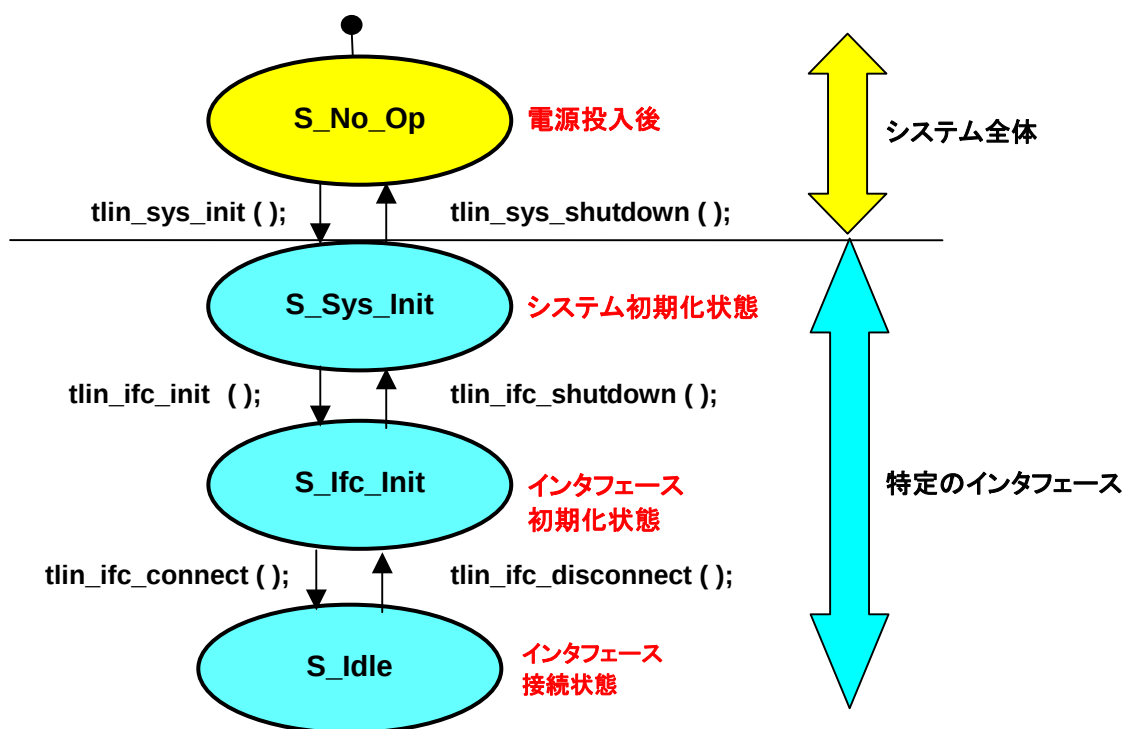
表 1 ユーザ API(システムとインタフェース初期化／シャットダウン)

分類	機能	API 名	概要
システム	初期化	tlin_sys_init	TLIN ソフトウェア全体の初期化
	終了	tlin_sys_shutdown	TLIN ソフトウェアの終了
インタフェース	初期化	tlin_ifc_init	NW インタフェースの初期化
	終了	tlin_ifc_shutdown	NW インタフェースの終了
	接続	tlin_ifc_connect	LIN バスとの接続
	切断	tlin_ifc_disconnect	LIN バスからの切断

```

TLinBool tlin_sys_init(void);
TLinBool tlin_sys_shutdown(void);
void tlin_ifc_init(TLinIfcHnd hIfc, TLinResHnd hRes, TLinBool bMaster);
void tlin_ifc_shutdown(TLinIfcHnd hIfc);
TLinU08 tlin_ifc_connect(TLinIfcHnd hIfc);
TLinU08 tlin_ifc_disconnect(TLinIfcHnd hIfc);
  
```

システムの初期化は、以下のようなシステム状態の遷移を前提にして、設計されている。



tlin_sys_init と tlin_sys_shutdown は、システム全体に対して適用される。したがって、これらの API 関数の引数は無し(void)なのである。

これに対して、インタフェースの初期化／接続／切断／シャットダウンは、各インタフェースごとに独立して行われる。したがって、これらの関数の引数には、インタフェース・ハンドル、レスポンス・ハンドルの他、マスタ／スレーブの指定が必要である。

例えば、TLIN_IFCHND_1 がマスタ、TLIN_IFCHND_2 がスレーブとして、Multi Cluster ノードの初期化を行うとすれば、以下のような記述になる。

```
/* Multi-cluster Node Initialization */
// Cluster #1 : Master, Cluster #2: Slave TlinBool is_OK;

/* System Initialization */
is_OK = TLIN_BOOL_FALSE;
while ( is_OK == TLIN_BOOL_FALSE ) {
    is_OK = tlin_sys_init( );
    if ( is_OK != TLIN_BOOL_FALSE ) {
        /* System Fault Confinement */
    }
}

/* Interface #1 Init & Connect */
tlin_ifc_init(TLIN_IFCHND_1, TLIN_RESND_1, TLIN_BOOL_TRUE);
tlin_ifc_connect(TLIN_IFCHND_1);

/* Interface #2 Init & Connect */
tlin_ifc_init(TLIN_IFCHND_2, TLIN_RESND_2, TLIN_BOOL_TRUE);
tlin_ifc_connect(TLIN_IFCHND_2);
```

図 10 初期化の例

tlin_sys_init を実行することにより、**TLIN Reference Code** に対して、使用するインタフェース・ハンドルとレスポンス・ハンドルを通知し、かつマスタとして動作するのか、スレーブとして動作するのかを伝える。この時点では、HAL の初期化は行われず、チャンネルが確保されるだけである。したがって、通信は行われず、バスのタイムアウト検出も行われない。

tlin_ifc_connect を実行することにより、初めて HAL が初期化され、関連する内部ハードウェア(タイマ, UART など)が動作する。すなわち、LIN BUS に接続され、そこで発生するアクティビティに対して反応する状態となる。

tlin_ifc_disconnect, tlin_ifc_shutdown は、上記の動作を、まったく逆方向に行う関数である。

インタフェースの初期化／接続／切断／シャットダウンは、インタフェース・ハンドルごとに独立して行われる。したがって、LIN Network Cluster #1(インタフェース・ハンドル #1)を動作させたまま、LIN Network Cluster #2(インタフェース・ハンドル #2)を切断／シャットダウンすることが可能である。この機能により、あるインタフェースが必要無い状態になった場合に、使用しているハードウ

エア・リソースを開放することが可能である。⁸

3.4.2. スケジュール設定と駆動(マスタのみ)

システムとインタフェースの初期化を終了した後で、スケジュール設定と、スケジュール駆動の API 関数を用いて、**TLIN Reference Code** の通信を制御する。また、システムに重大なエラーが生じた場合にアプリケーションに通知を行うコールバック関数が組み込まれている。

これらの関数は、インタフェースごとに独立して制御を行うために、インタフェース・ハンドルを引数に持つ。また、tlin_ifc_init 関数で、マスタに指定した場合のみ有効である。スレーブ指定した場合には、この関数を発行しても、**TLIN Reference Code** は無視する。

以下にスケジュール設定と駆動のために容易された API 関数と、プロトタイプを示す。

表 2 スケジュール設定と駆動のために容易された API

分類	機能	API 名	概要
インタフェース	スケジュール設定	tlin_ifc_sch_set	送信フレームテーブルの指定
	スケジュール駆動	tlin_ifc_sch_tick	送信契機の通知 次に送信されるフレーム番号を返す
コールバック	ハードエラー	tlin_cb_error	Fatal エラー検出時

```
void      tlin_ifc_sch_set(TLinIfcHnd hIfc, TLinSchHnd hSch, TLinU08 u08ent);
TLinU08   tlin_ifc_sch_tick(TLinIfcHnd hIfc);
void      tlin_cb_error(TLinIfcHnd hIfc, void* nay);
```

⁸ 論理的に、このようなインプリメンテーションが可能なロジックであるようにしたが、実際には、システムを不安定にする要因となるので、このような使用法は推奨しない。

3.4.2.1. tlin_ifc_sch_set(スケジュール設定関数)

tlin_ifc_sch_set によって、実行するスケジュール・テーブルとエントリ(開始位置)を指定する。

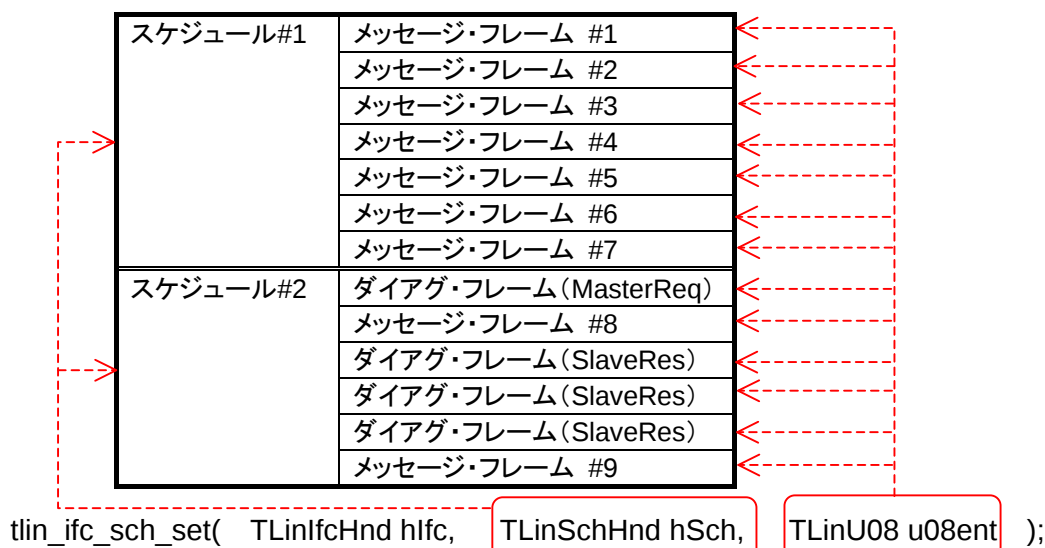


図 11 スケジュール設定の例

スケジュール・テーブルのイメージは、図 11 に示すようなものである。スケジュール・テーブルには複数のスケジュールが登録されており、どのスケジュールを実行するかを、第 2 引数の hSch (スケジュール・ハンドル) で指定する。また、してされたスケジュールの、どのメッセージ・フレームから開始するかを、u08ent(エントリ番号)で指定する。

tlin_ifc_sch_set が有効になるタイミング

tlin_ifc_sch_set は、インタフェース・コネクト後であれば、いつでも受け付けられる。

tlin_ifc_sch_set を発行した時点で、**TLIN Reference Code** がメッセージ・フレームの転送中であれば、そのフレームの転送が終了した時点で、スケジュール設定が有効になる。**TLIN Reference Code** が、メッセージ・フレームの転送中でなければ、即座に有効になる。

3.4.2.2. tlin_ifc_sch_tick(スケジュール駆動関数)

tlin_ifc_sch_tick は、スケジュール設定が発行された後であれば、いつでも、何回でも発行可能である。

tlin_ifc_sch_tick をトリガとして、**TLIN Reference Code** は、現在のエントリ番号が示すメッセージ・フレームの転送を開始する。tlin_ifc_sch_tick は、引数として駆動するインタフェースのハンドルを持ち、戻り値は tlin_ifc_sch_tick により駆動されたフレームのエントリ番号となる。

スケジュールが設定されていない状態や、スリープ中に tlin_ifc_sch_tick を発行した場合は無視され、戻り値は 0 となる。(エントリ番号は、必ず 1 から始まるため、エラーであることを確認できる。)

tlin_ifc_sch_tick を **TLIN Reference Code** が認識し、指定されたフレームの転送を終了した時点（レスポンスの転送が終了した時点）で、**TLIN Reference Code** 内部で、エントリ番号が次のエントリに更新される。スケジュールの最後のエントリまで実行した場合は、エントリ番号は最初の値に戻される。すなわち、スケジュールに含まれるメッセージ・フレームをリング状に実行する。

tlin_ifc_sch_tick により、あるエントリ番号に対応するメッセージの転送が開始され、レスポンス転送が終了する前に、新しい tlin_ifc_sch_tick を発行した場合、その要求は 1 回分のみ保持される。この場合、戻り値は、現在実行中のエントリ番号となる。**TLIN Reference Code** は、現在のエントリ番号のフレーム転送が終了した時点で、tlin_ifc_sch_tick 要求が保持されていれば、エントリを次の値に更新した後、スロット時間を経過後、（新規に tlin_ifc_sch_tick が無くても）即座にメッセージ転送を始める。

TLIN Reference Code は、LDF.C で与えられるスロット時間(delay)を認識し、必ずスロット幅を確保した後に、次のフレーム転送を行う。

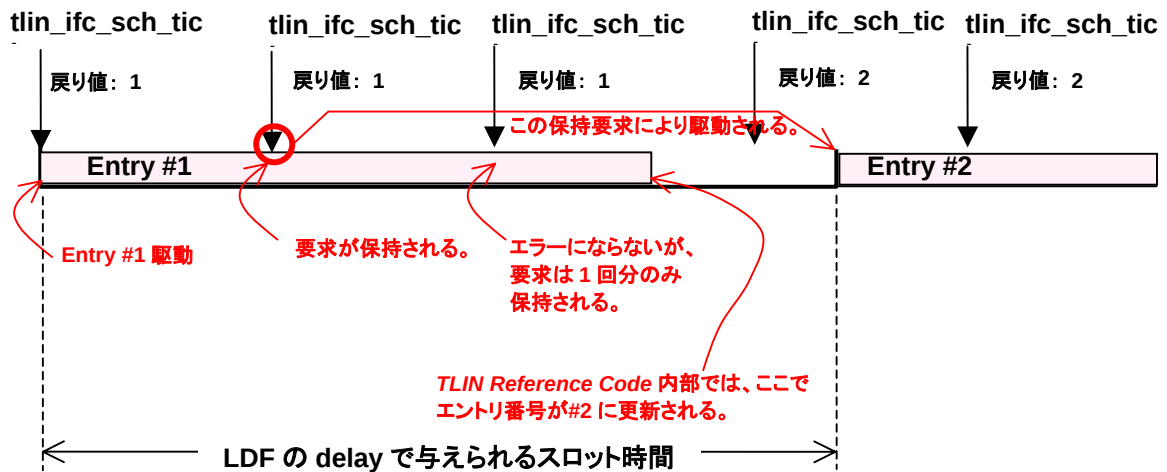


図 12 tlin_ifc_sch_tick の駆動タイミング

tlin_ifc_sch_tick の発行タイミング

基本的に tlin_ifc_sch_tick は、どのようなタイミングで発行しても構わない。しかし、CPU に余分な負荷をかけないために、以下の点に注意すること。

- ① システムが規定する jitter 時間より短い間隔で発行しないこと。
- ② delay で規定されるスロット時間より短い間隔で発行すること。

一般的には、最も短いフレームの送信時間（例えば、9600bps の場合、約 7～10msec）に合わせて発行することにより、スロット間に遅れのない転送が実現できる。

3.5. シグナル・アクセス API

シグナル・アクセス用に以下の API が用意されている。

表 3 シグナルアクセス API

分類	機能	API 名	概要
シグナル	値取得	<code>tlin_sig_read</code>	シグナル値の取得
	値設定	<code>tlin_sig_write</code>	シグナル値の設定
	更新フラグ取得	<code>tlin_sigflg_test</code>	シグナル値受信更新フラグ値の取得
	更新フラグクリア	<code>tlin_sigflg_clear</code>	シグナル値受信更新フラグ値のクリア

```
TLinBool tlin_sig_read_bool(TLinSigHnd hSig);
TLinU08  tlin_sig_read_u08(TLinSigHnd hSig);
TLinU16  tlin_sig_read_u16(TLinSigHnd hSig);
void      tlin_sig_write_bool(TLinSigHnd hSig, TLinBool bVal);
void      tlin_sig_write_u08(TLinSigHnd hSig, TLinU08 u08Val);
void      tlin_sig_write_u16(TLinSigHnd hSig, TLinU16 u16Val);
TLinBool  tlin_sigflg_test(TLinSigHnd hSig);
void      tlin_sigflg_clear(TLinSigHnd hSig);
```

前述したように、エンティティ内のシグナルにアクセスする関数である。

1 ビット長の `bool` 型, 8 ビット長の `u08` 型, 16 ビット長の `u16` 型について、`read` と `write` が準備されている。また、`bool` 型については、`test` と `clear` の論理フラグ型のアクセス関数も用意されている。

詳細は、『TLIN API 仕様書』を参照すること。

3.6. Network Management API

Sleep 要求, Wakeup 要求に対応する API と、関数プロトタイプを以下に示す。

表 4 NM 対応 API

分類	機能	API 名	概要
NW 制御	Sleep 要求	tlin_ifc_sleep_req	スリープ要求
	Sleep 可否	tlin_ifc_sleep	スリープへの移行の可否を設定。システム起動時は、スリープ不可になっている。
	Wakeup 要求	tlin_ifc_wakeup	ウェイクアップ要求
コールバック	Sleep 成立	tlin_cb_sleep	スリープ状態成立コールバック API
	Wakeup 成立	tlin_cb_wakeup	Wakeup 状態成立コールバック

```
Void tlin_ifc_sleep(TLinIfcHnd hIfc, TLinBool bEnable);  
Void tlin_ifc_wakeup(TLinIfcHnd hIfc);  
Void tlin_ifc_SetDataInd(TLinIfcHnd hIfc, TLinBool bEnable);
```

tlin_ifc_sleep は、即座にスリープに移行することを要求するものではないことに注意。(スリープ要求は、tlin_ifc_sleep_req で行なう。)

第 2 引数・TLinBool bEnable により、当該ノードがスリープに移行することが可能であるか否かを指定するものである。

第 2 引数・TLinBool bEnable が、TLIN_BOOL_TRUE と指定された状態で、この関数が発行され、スリープ条件が成立した後、初めてスリープに移行する。スリープ条件とは、Bus No Activity が検出されるか、もしくは Sleep Command を送信／受信した場合である。(Master Request フレームで、第 1 バイトが 0 の場合)

3.7. ダイアグ API

ダイアグ API は、ダイアグ用スケジュール実行中に使用される API である。以下の API 関数が用意されている。

表 5 ダイアグ用 API

分類	機能	API 名	概要
ダイアグ	受信データ取得	tlin_diag_read	ダイアグデータの取得
	送信データ設定	tlin_diag_write	ダイアグデータの設定
	応答フラグ設定	tlin_diag_setResponse	ダイアグ応答の有無の指定
コールバック	ダイアグ受信通知	tlin_cb_diag_rcv	ダイアグメッセージの受信を通知
	ダイアグ送信通知	tlin_cb_diag_send	ダイアグメッセージの送信を通知

```

/** Diag-API */
Void tlin_diag_read ( TLinIfcHnd hHnd, TLinU08* pData )
Void tlin_diag_write ( TLinIfcHnd hHnd, TLinU08* pData );

/** Diag-API(callback) */
void tlin_cb_diag_rcv(TLinIfcHnd hIfc);
void tlin_cb_diag_send(TLinIfcHnd hIfc);

```

ダイアグ通信は、以下のようなシーケンスで行う。

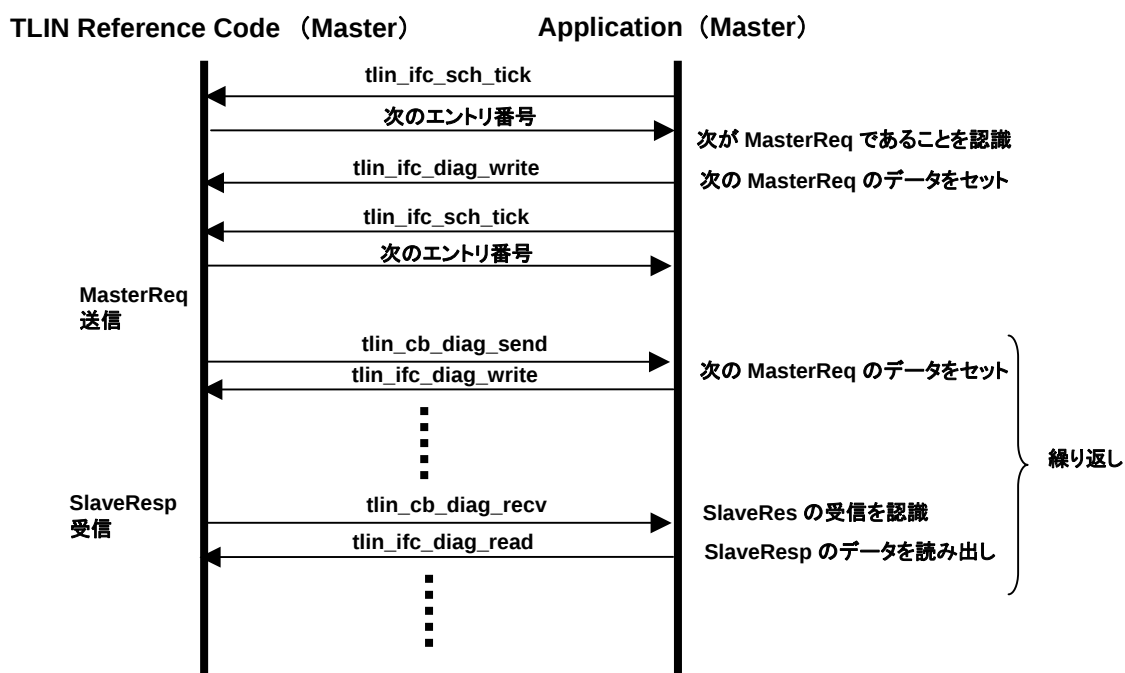


図 13 Master Node の Diag シーケンス

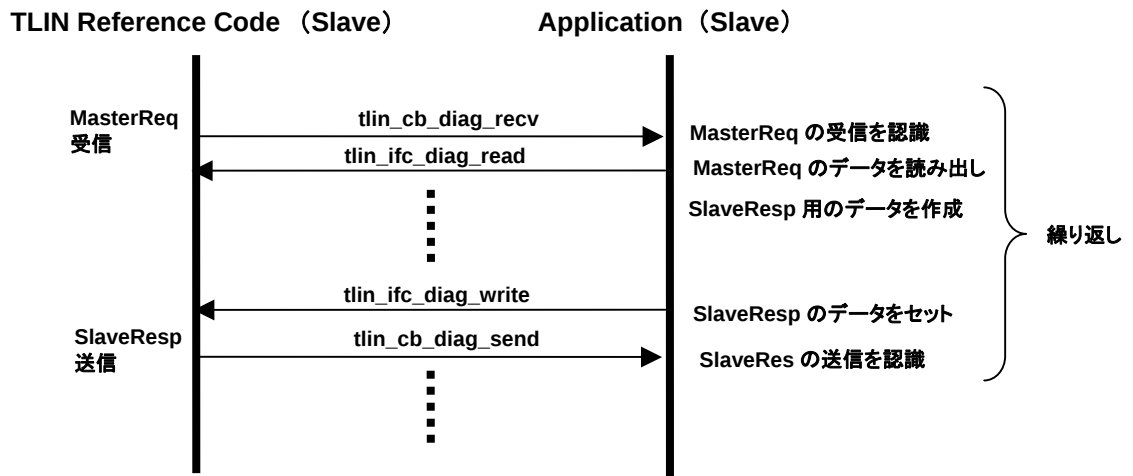


図 14 Slave Node の Diag シーケンス

4. TLIN Reference Code のタスク構成

ここでは、*TLIN Reference Code* のタスク構成について述べる。

4.1. TLIN Protocol Stack のタスク構成

TLIN Reference Code は、Real Time Operating System の下で動作するタスクとして記述されている。機能面での構成は以下のようになる。

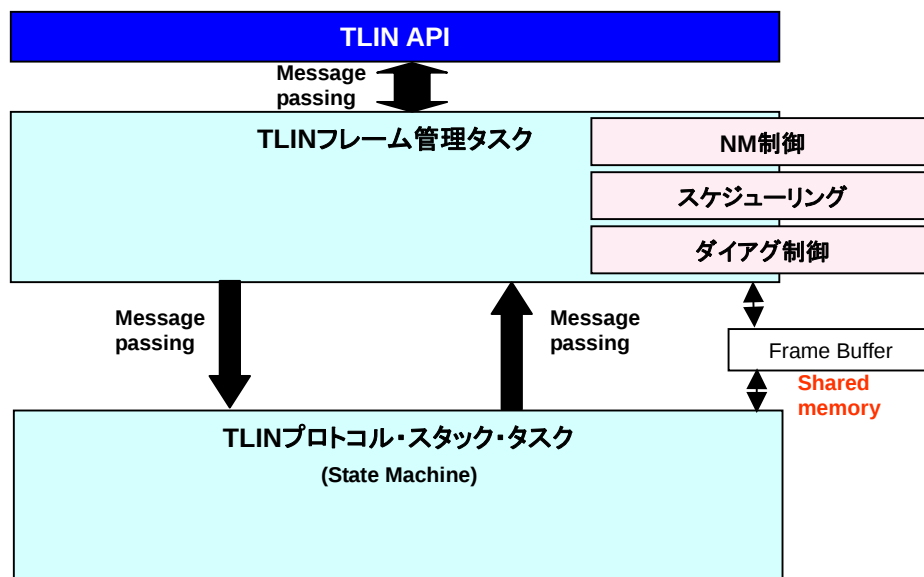


図 15 TLIN Reference Code のタスク構成

TLIN プロトコル・スタック・タスクは、純粋に LIN コンソシアム仕様の仕様に従うスタックである。TLIN 仕様で追加された NM 制御、ダイアグ制御は実現せずに、上位の TLIN フレーム管理タスクからのメッセージで、LIN プロトコル仕様 (Spec 1.3) に従う転送制御を行う。

TLIN 仕様で追加されている NM 制御や、ダイアグ制御は、上層の TLIN フレーム管理タスクで行う。また、このタスクは、API リクエストを直接受け付け、適当なフィルタリングを行った後に、下層の TLIN プロトコル・スタック・タスクにメッセージを送って、動作制御を行う。

さらに詳細なタスク設計仕様および内部メッセージ、状態遷移設計は、“TLIN Reference Code タスク設計資料”を参照すること。

5. HAL

TLIN Reference Code 設計においては、ハードウェア依存部分を、HAL (Hardware Abstraction Layer) として、分離している。

その機能およびインタフェースの詳細は、“T-LIN リファレンス HAL 仕様書”を参照すること。本章では、その設計コンセプトのみ述べる。

5.1. ハードウェアのモデル

HAL 設計の上で、使用するマイコンについては、必要最小限の機能のみを想定して、極力汎用性をあげてをねらった。想定ハードウェア機能は、以下の項目である。

- 8bit UART (Transmitter / Receiver のセット) が少なくとも 1 channel 保有
- UART は Receiver 受信時に、RxRDY (受信レジスタ・レディ) 割り込みを発生する。
- UART は Transmitter 送信完了時に TxEmp (送信レジスタ・エンプティ) 割り込みを発生する。
- 2 本の Auto Reload Timer を保有する。
- Timer は、最低限 0.5msec (UART 1bit 長の 1/2) 程度の時間計測精度を実現できること。
- Timer は、Overflow または Underflow 割り込みを持つこと。
- Synch Break / Physical Bus Error 検出トリガ用に、Up/Down edge 割り込み指定可能な外部割り込み端子を 1 本もつこと。

想定したハードウェア機能は、上記のみである。

5.2. イベント設計

上記のハードウェア・モデルの範囲内で、以下に述べるイベントを検出し、メッセージを TLIN プロトコル・スタックへ post する機能を組み込んだ。

- **RxRdy イベント**: マイコンの UART Receiver が 1 バイト分のデータを受信し終わり、Receiver のデータ・レジスタにデータがセットされた。
- **TxEmp イベント**: マイコンの UART Transmitter が 1 バイト分のデータを送出し終わり、Transmitter のデータ・レジスタが Empty であり、次のデータセット待ちである。
- **SynBrk Detected イベント**: LIN BUS が、10 ビット以上 dominant を継続した瞬間に発生する。LIN BUS が recessive に戻るのを待たない。UART イベントとは、まったく独立に動作する。
- **Physical Bus Error イベント**: LIN BUS が、 T_{TIMEOUT} 時間以上 Dominant (Ground level) であった場合に、このイベントが検出される。他のイベント検出とは、まったく独立して動作する。
- **BUS No Activity イベント**: LIN BUS が、 T_{TIMEOUT} 時間以上有効な信号が現れなかった場合に、このイベントが検出される。
- **T_{MAX_HEAD} イベント**: 最大ヘッダ幅超過の検出

- **T_{MAX_FRAME} イベント**: 最大フレーム幅超過の検出
- **T_{MAX_SLOT} イベント**: メッセージ・スロット幅終了の検出
- **T_{MAX_BYTE} イベント**: UART の送出エラー検出
- **T_{WUDEL} イベント**: Wake-up Delimiter 時間計測イベント
- **T_{WURTY} イベント**: Wake-up Retry 時間計測イベント
- **T_{T3BRK} イベント**: T3BRK 時間経計測イベント

RxRdy, TxEmp イベントは、UART の割り込みをそのまま使用。

次の **SynBrk Detected**, **Physical Bus Error** は、外部割り込み端子と、Timer 1 本で実現。残りのイベントは、全て物理 Timer 1 本を用いた仮想タイマにより計測するものとした。

5.3. HAL のチャンネル

LIN Network multi-cluster に対応して、HAL も複数のチャンネルに対応する構造にした。

tlin_ifc_init API 関数により、インタフェース・ハンドルが割り当てられると、フレーム管理タスクは、以下のテーブルに、インタフェース・ハンドルと、対応する実行モード(マスタ/スレーブ)と共に、使用する HAL のチャンネル番号を割り当てる。

HAL 関数およびプロトコル・スタックは、このテーブルを参照することにより、使用するチャンネルを知ることができる。

```

/*===== インタフェース・ハンドル構造体のテーブル =====*/

T_TLIN_IFC_HNDL tlin_ifc_hndl_tbl[] = {
/* Interface handl      Execution mode      HAL channel */
      0,                TMODE_TLIN_NON,      0xFF,
      0,                TMODE_TLIN_NON,      0xFF,
      0,                TMODE_TLIN_NON,      0xFF,
};

```

図 16 インタフェース・ハンドルと HAL チャンネル割付テーブル

6. OSAL

TLIN Reference Code で参照している OS サービスコールは、以下のものである。

- Message 送信
- Message 受信
- Event Flag のセット
- Event Flag のウェイト
- セマファの取得／開放

詳細は、別途“T-LIN リファレンス OSAL 仕様書”に示す。

付録 A

1. 設計プロセスのモデル

TLIN Reference Code は、LIN プロトコル・スタック・ライブラリではない。

ライブラリは、特定の CPU のために用意されるものである。つまり、プロトコル・スタックを実現するために、マイコンのリソース(ペリフェラル、割り込みなど)を用いて実現するためのコードを用意するのが、ライブラリであり、必然的に対象となるマイコンのマシン語や、コンパイル環境、使用可能なペリフェラル(タイマや、ポート、UART など)の違いや、制御方式の違いによって、ライブラリは個々に異なるものとなる。

TLIN Reference Code は、異なるマイコンと、異なるコンパイル環境においても参照可能であることを目的とする。そのために、実際の ECU に対して、ある特定のマイコンとコンパイル環境および OS 環境を選定するための設計プロセスを前提とする。

つまり、アプリケーション作成のためのフレームワーク(プロファイルという表現もある)と考えていただきたい。

1.1. プロセス・フローの概要

以下に一般的な通信プログラムの設計プロセス・フローを示す。

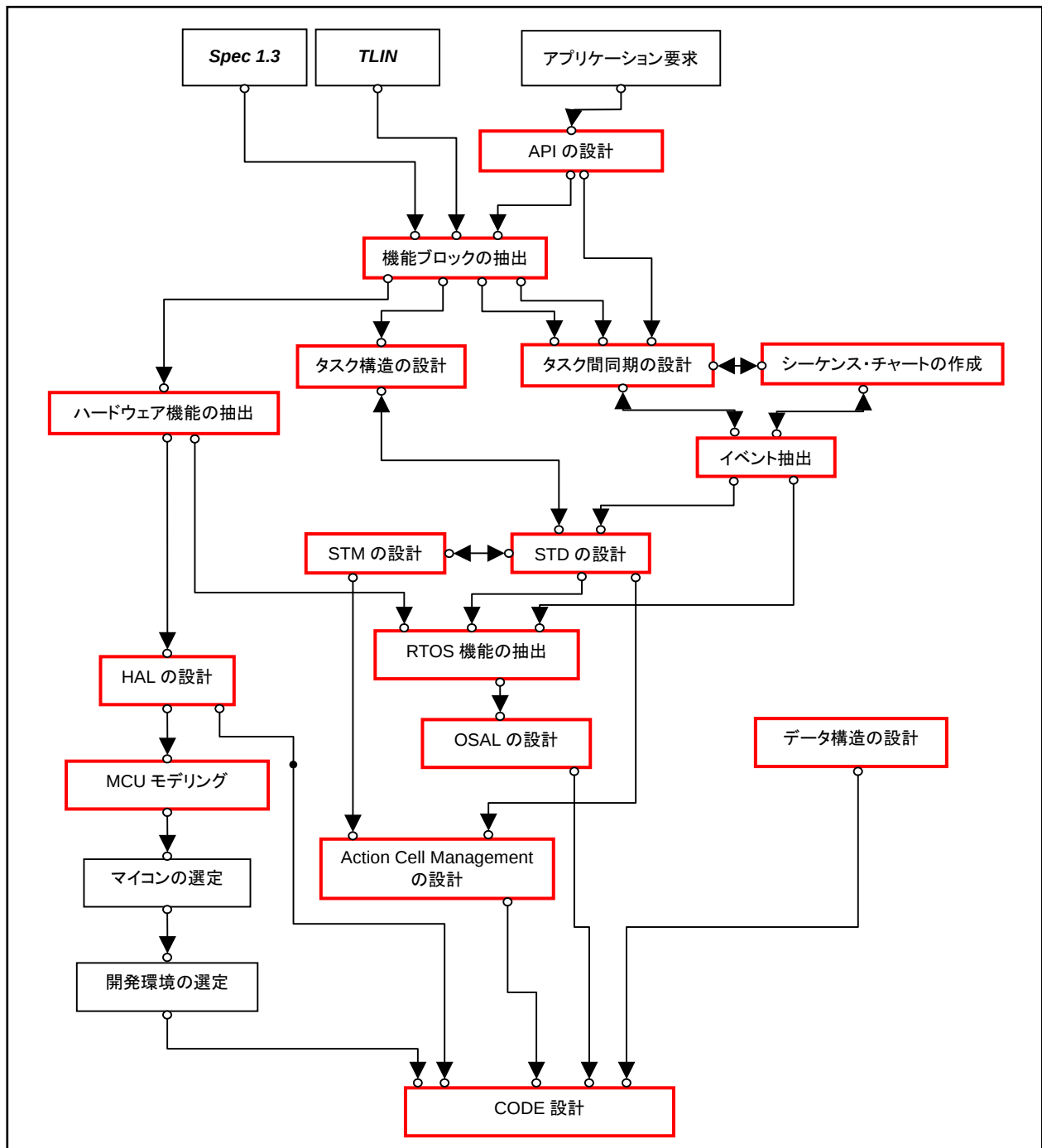


図 17 設計プロセス・フロー

STD : State Transition Diagram

STM : State Transition Matrix

HAL : Hardware Abstraction Layer

OSAL : Operating System Abstraction Layer

設計プロセス・フローは、時系列的に上部から下部へ流れる。ひとつのプロセスに、複数の矢印が入って来るのは、そのプロセスが複数の上流のプロセスの結果を反映することを示す。また、矢印が両方向を向いている(◄►)のは、下流プロセスの結果によっては、上流プロセスへのフィードバックが行われることを示す。

図 17 の赤い線のブロック()が、本 TLIN Reference Code で提供する部分である。