TOPPERS 活用アイデア・アプリケーション開発コンテスト

部門　：　アプリケーション開発部門

作品のタイトル　：　SOLID for Raspberry Pi を用いた RTOS と Linux が共存する環境での自律移動システムの開発

作成者　：　岡田 航季(苫小牧工業高等専門学校)

共同作業者　：　山本 椋太(苫小牧工業高等専門学校)

作品の対象者　：　SOLID for Raspberry Pi 使用者
Raspberry Pi を使用し RTOS、Linux を使用したロボットを作成したい方

使用する開発成果物　：　SOLID for Raspberry Pi(TOPPERS/FMP3)

目的・狙い

- 車載システムの高性能化による ECU の統合の流れを受け、RTOS と Linux とがワンボード上で共存させる環境で自律移動システムを開発し、統合 ECU における自律移動システムに求められる要件・システム構成について考察する。

アイデア/アプリケーションの概要

RTOS と Linux を Raspberry Pi 上で共存させる環境である SOLID for Raspberry Pi を用いて、自律移動を行うシステムを開発した。作成したシステムでは RTOS 側に SLAM を実装し、Linux 側では ROS 上にシステムを構築し、外部デバイスとの通信を担当する。

1. 背景

今日、自動運転を意識した車載システムの高性能化が進んでいる。しかし、高性能化に伴う電子制御装置（ECU）やセンサ量の増加、配線量の増加による車体重量の増加、車内空間のひっ迫が問題となっている[1]。近年、機能毎で独立していた ECU を単一の ECU に統合し、ECU の搭載数を減らす流れがあり、仮想マシン（VM）を用いた手法が提案されている[2]。しかし、仮想化では、VM の切替えや仮想化による実行オーバーヘッドによってリアルタイム性の保証が懸念される。

本研究では、SOLID for Raspberry Pi を用いて、高い信頼性とリアルタイム性を持つ RTOS と、豊富なソフトウェア資源を持つ Linux とをワンボードに集約し、各 OS が複数の役割を持ったデュアル OS 環境での自動運転システムを作成し、共存する環境の性能評価を目的とする。

2. 研究環境

開発環境として SOLID を使用し、RTOS と Linux を共存させる環境である SOLID for Raspberry Pi[2]上で自律移動システムを構築する。SOLID は京都マイクロコンピュータ（KMC）社の提供する組込み開発プラットフォームである。Arm Cotex-A デバイスを対象とした RTOS である SOLID-OS と統合開発環境（IDE）、コンパイラツールチェーン、デバッガ等のツール群である SOLID-IDE とが一体化されていることが特徴である。RTOS と IDE が一体化することで、プログラムで使用するアドレスを事前に把握し、不正なメモリアクセスを直前に検出するアドレスサニタイザ機能、タスク状況や RTOS の資源の表示等の機能が提供されている[3]。SOLID の構成を図 1 に示す。

SOLID for Raspberry Pi は Raspberry Pi を用いた SOLID によるリアルタイムアプリケーション開発の体験を目的として配布されているソフトウェアパッケージである。本研究では無償版を使用する。ターゲットである Raspberry Pi の構成を図 1 に示す。SOLID for Raspberry Pi では、RTOS である SOLID-OS と汎用 OS である Raspberry Pi OS が 2 コアずつを使用し共存するデュアル OS 環境となる。SOLID for Raspberry Pi での Raspberry Pi OS は Debian11（Bullseye）64bit 版であり、SOLID 固有のアプリケーションやデバイスドライバが追加されている。また、SOLID-OS と Raspberry Pi OS の間に OS 間通信機構（OSCOM RPC）が提供されている。また、ファイルシステム・ネットワーク API コールを Linux システムコールに変換、実行する仕組みを持つため、SOLID-OS は Linux のファイルシステムやネットワーク等の機能が利用可能であり、RTOS によるリアルタイム処理も可能となっている。

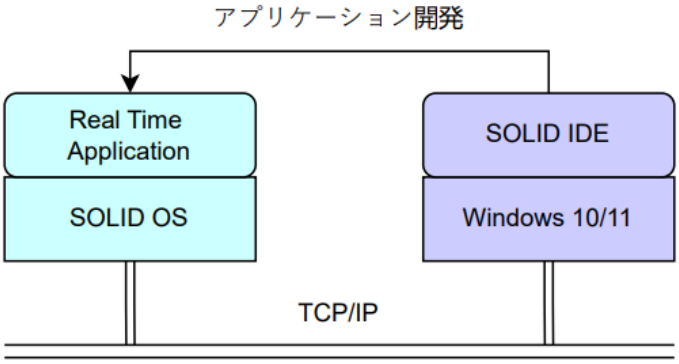

図 1 SOLID の構成[3]

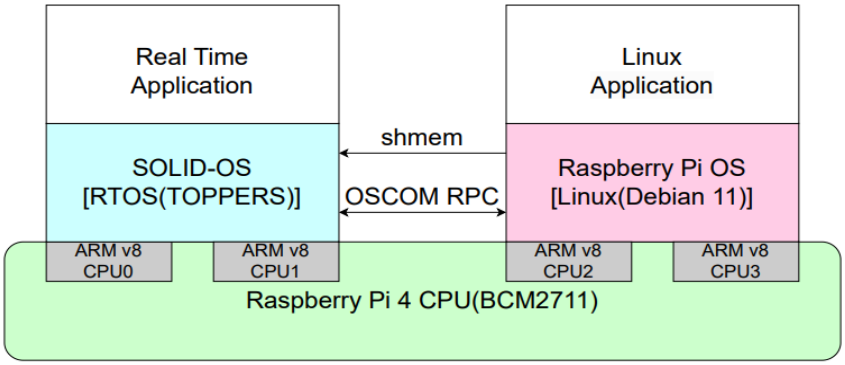

図 2 SOLID for Raspberry Pi の構成[3]

3. 設計

3.1. システム要件

作成するシステムの要件を考える。システムの要件は以下の 6 項目となる。

(1) 障害物を避け目的地へ走行すること。

(2) 周囲の環境に応じて走行経路を動的に導出し経路に沿って走行すること。

(3) 正確な自己位置推定と正確な環境地図の作成を行うこと。

(4) コントローラによって自律移動装置の作動及び停止が可能であること。

(5) 自律移動システムが正常に動作しない恐れがある場合には直ちに停止すること。

(6) 自律移動システムの作動状況を LED によって周囲に作動を知らせること。

要件(1)について、ロボットが目標地点に到達するためには障害物への衝突回避を行い安全に走行する必要がある。要件(2)について、ロボットの走行する環境は、対向車や歩行者など実環境を想定した、周囲の障害物の位置が変化する可能性がある動的環境とする。そのため、外部センサを用いて周囲の環境を把握することで、周囲の環境に応じた走行経路を適時変更する必要がある。要件(3)について、障害物を避け目標地点へ進行するために正確な自己位置の推定と正確な環境地図の作成が求められる。要件(4)(5)(6)は

自動運行装置について国土交通省より 2018 年 9 月に発表された自動運転車の安全技術ガイドライン[4]を参考に設定した。システム構成図を図 3 に示す。

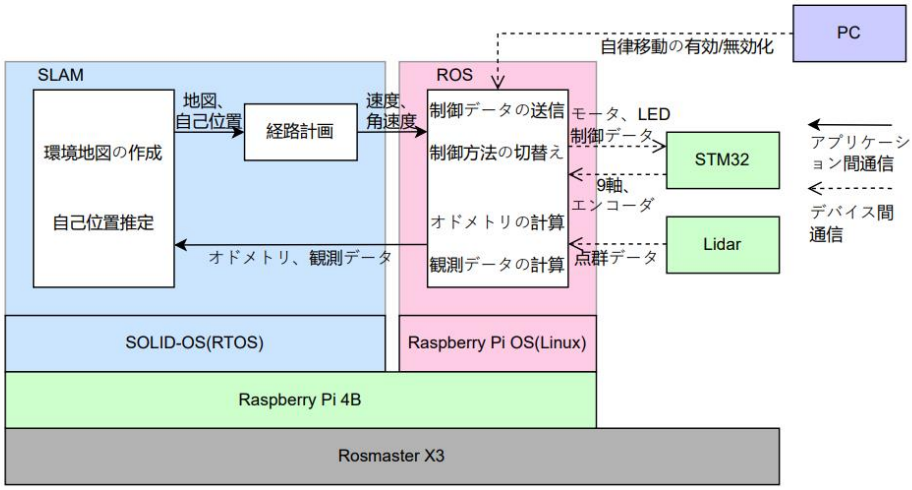

図 3 システムの構成図

3.2. RTOS 側アプリケーションの構成

RTOS 側は優先度ベースのスケジューリングでありリアルタイム性が高められるが、デバイスドライバは用意されていない。要件(4)(5)(6),(2)の外部センサからの環境データの取得は外部との通信が必要である。レジスタ操作による UART 通信を試みたが RTOS 側で外部デバイスと通信を実現することができなかった。Linux のみの環境では同様のプログラムで動作したため、SOLID 環境では何かしらの設定が行われているものと考察する。そのため RTOS 側では SLAM とその結果を用いた経路の導出、制御パラメータの決定を行う。本研究では 2 次元 LiDAR を使用するため、openslam_gmapping [5]を RTOS へ移植する。RTOS 側アプリケーションでは以下の 4 タスクを定義する。タスクの詳細を表 1 に示す。各タスクはイベントフラグによって同期をとる。

oscom_task は OS 間通信タスクである。ソケット通信のサーバを立ち上げ、Linux 側クライアントからのデータ更新通知をポーリング待ちで待つ。クライアントの接続をポーリング待ちで待つため、oscom_task で CPU1 を占有し応答性の改善を図る。slam_task は SLAM 計算タスクである。Linux 側で取得されたオドメトリと観測データを読み込み、自己位置の推定と環境地図の更新を行う。map_task は地図保存タスクである。走行データの記録として、地図を画像ファイルとして保存する。astar_task は A*アルゴリズムを用いて経路計画を行うタスクである。slam_task によって推定された自己位置と更新された環境地図を用いて最短経路の探索を行い、経路を追従するための走行パラメータの更新を行う。

表 1 タスクの詳細

項目	oscom_task	slam_task	map_task	astar_task
優先度	4	8	12	4
割付け CPU	1	2	2	2
待ちイベントフラグ		0x04	0x08	0x08
セットイベントフラグ	0x04	0x08		

3.3. Linux 側アプリケーションの構成

Linux 側は豊富なソフトウェア資源が利用可能であり、ラウンドロビンベースのスケジューリングによって CPU リソースが均等に割当たる[27]。

要件より、要件(4)(5)(6),(2)のデータ取得は STM32 マイコンへの制御データの送信、LiDAR との USB シリアル通信、遠隔停止用外部 PC とのソケット通信といった通信が必要である。Linux 側アプリケーションは ROS 上に構築する。ROS では実行単位である ROS ノードとノード間通信であるトピックを介したトピック通信による分散型システムを構築する。ROS ノードの構成図を図 4 に示す。

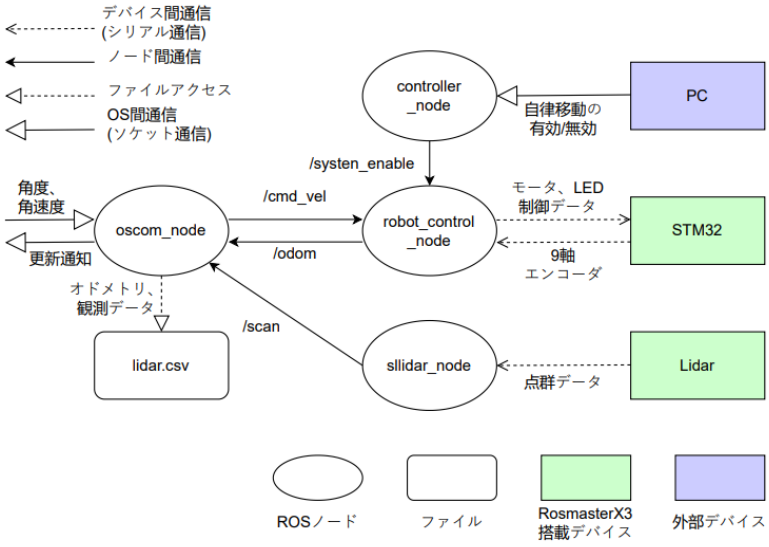

図 4 ROS ノードの構成図

4. 評価実験

4.1. 実験 1

実験 1 では、実装した SLAM アルゴリズムが正しく動作するかを調査する。SLAM の入力データであるオドメトリに対し x,y 方向に対して最大 0.4[m](自律移動ロボットの約 2 倍の大きさ)、機体角度に対して 0.1[rad]の測定誤差を含ませ、誤差がある場合でも正確に位置推定が行われるかを確認する。2 つの誤差パターンで各 10 回自己位置推定を行い、平均を推定される自己位置とする。

実験結果を表 3 に示す。結果より、作成する環境地図の解像度である 0.05m 以内の誤差に収まっていることは正確な自己位置推定ができていると評価できる。

表 3 誤差を含む自己位置推定の結果

	自己位置推定(1)			自己位置推定(2)		
	x[m]	x[m]	θ [rad]	x[m]	x[m]	θ [rad]
誤差	0.3	-0.1	0.1	0.4	-0.3	0.1
真値との差	-0.002	-0.021	0.037	0.008	0.012	-0.002

4.2. 実験 2

実験 2 では、経路上に障害物があることで目的地へ直接進行が出来ない場合に、経路計画を動的に行うことで目的地へ移動できるかを検証する。開始位置から見た目的地の座標のみを入力と与え、あらかじめ生成された環境地図等は与えない。障害物によって目的地の座標へ直接進行が出来ないため、動的な地図構築、経路探索が必要となる。

実験 2 の走行の様子を図 5 に示す。ロボットは図 5 に示すように障害物を回避する経路を走行した。しかし、大きく旋回する際に障害物に衝突し失敗した。これは経路追従において、角速度のみを決定するアルゴリズムで並進速度は一定であったためと考えられる。

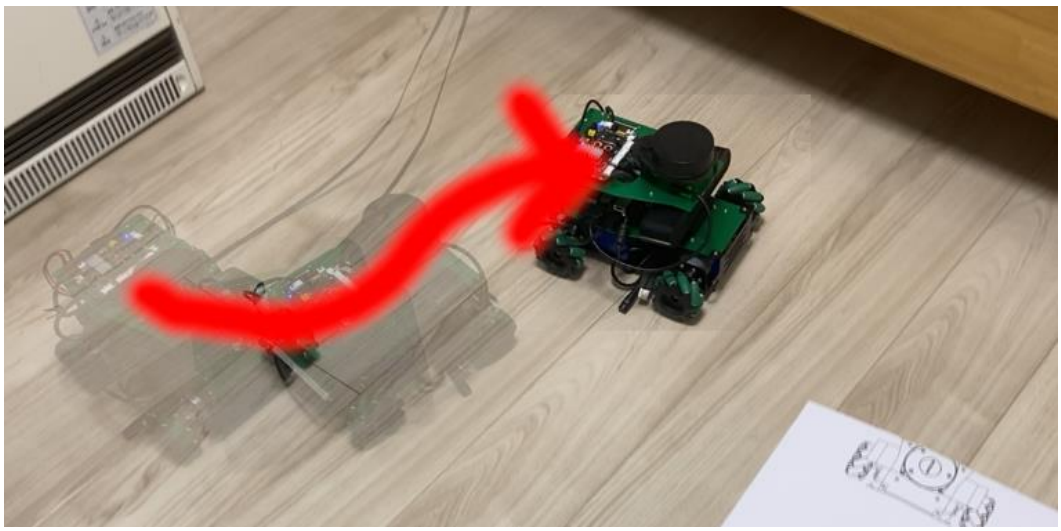

図 5 走行の様子

4.3. 実験 3

実験 3 では、RTOS と Linux とで SLAM 処理時間に差があるかを調査する。RTOS 側に実装した SLAM プログラムを Linux 側に移植し以下の項目について計測する。

- 計測 1 観測データの読込み時間の計測

- 計測 2 初期化処理の処理時間の計測
- 計測 3 SLAM の処理時間の計測

計測 1,2 ではそれぞれ 1000 回の実行を行い、平均を処理時間とする。

計測 3 では SLAM プログラムに自己位置推定と環境地図更新の処理を依頼してから結果を受け取るまでの時間を計算する。RTOS 側実装の計測では RTOS 側に更新通知を行い、推定された自己位置を Linux 側が受け取るまでの時間を計測する。Linux 側実装の計測ではデータ読み込みから自己位置推定、環境地図の更新終了を行う関数の実行時間を計測する。

計測結果を表 4 に示す。計測 1 のデータの読込みでは、RTOS 側実装では 1.53 倍遅い結果となった。これは SOLID-OS は SOLID API を介して OSCOM RPC による OS 間通信を行うことで Linux ファイルシステムを利用するため、RPC によるオーバーヘッドが影響し RTOS 側実装では遅くなったと考えられる。計測 2 の初期化処理では、RTOS 側実装が 2.15 倍速い結果となった。計測 3 の SLAM の処理時間では RTOS 側実装が 1.65 倍速い結果となった。RTOS 側実装では優先度ベースのスケジューリングによって SLAM タスクは CPU リソースを占有し実行される。対して Linux 側実装ではラウンドロビンベースのスケジューリングによって他タスクと CPU リソースを分け合って実行される。そのため単純な計算処理を行う計測 2、計測 3 では CPU リソースを占有する RTOS 側実装の方が処理時間が短くなったと考えられる。

表 4 SLAM の処理時間の比較

項目	RTOS 側処理時間[ms]	Linux 側処理時間[ms]
計測 1;データ読み込み	1.106	0.722
計測 2:初期化処理	0.624	1.344
計測 3:SLAM の処理時間の平均	1024.84	1689.666

5. 考察

5.1. 自律移動システムの考察

要件(1)(2)(3)は RTOS 側アプリケーションで実現する要件である。要件(1)(2)は 4.2 節より、障害物を避ける経路を導出し目的地へ進行する動作を確認した。しかし、走行パラメータが適切でなかったため、走行経路から外れ目的地へ到達することができなかった。要件(3)は 4.1 節、4.2 節より、ノイズ成分が大きい状態や走行状態でも正確な自己位置推定と正確な環境地図の作成が行われていることを確認した。

要件(4)(5)(6)は Linux 側アプリケーションで実現する要件である。要件(4)は 3.3 節より、外部デバイスとの通信を行う ROS ノードを実装し、受信したシステムの有効/無効を指定するデータに応じて送信する走行パラメータを切替えることで実現した。要件(5)はロボットの制御を行う ROS ノードでデータの更新間隔を確認し、一定時間更新が行

われない場合はロボットの停止を行う機能を実装した。また、要件(6)はシステムが無効化されている場合、車体後部にある LED ランプを点灯させることで実現した。

しかし、外部デバイスとの通信を Linux 側で行っているため、応答性は高くなく、通信間隔は ROS のタイマーコールバックに一任している。また、OS 間通信をソケット通信で実装しているため、OS 間通信タスクはポーリング待ちで Linux 側アプリケーションからの通信を待つ必要がある。そのため RTOS に割当たった 2 コアのうち 1 コアを占有する実装となってしまった。割込みとして OS 間通信を実現することができれば 2 コアを使用してより役割を持たせることが可能となる。

5.2. プラットフォームに関する考察

本研究では SOLID for Raspberry Pi 環境を用いて、自律移動システムを作成した。RTOS である SOLID-OS では Linux ファイルシステムやネットワークシステムが利用可能であり、OS 間で協調して動作するための仕組みが提供されている。そのため RTOS でありながら容易に OS 間通信を実現することができた。しかし、外部デバイスとの通信は提供されておらず、カーネル内部に関する情報が公開されていないことや知識不足もあり、実現することができなかった。Linux 側は ROS 推奨環境ではない Raspberry Pi OS となっており、非推奨環境に ROS を導入した。そのため一部の ROS パッケージが利用できない、正規の手順では ROS ノードをビルドできないといった問題が発生した。しかしながら、RTOS 側で Linux の機能が利用可能である点、コーディングからビルド、デバッグまでを一つのウインドウで行え、快適な開発を行える点は非常に有用である。また、SOLID for Raspberry Pi では機能の拡張のみが行われており TOPPERS FMP3、Raspberry Pi OS 共に既存の OS が持つ機能に関する制限事項はなく、割り付けられた CPU でそれぞれ動作するため、同時に両 OS へ接続しデバッグを行うことが可能であった。以上から、発生した問題点よりも開発において補助となった点は大きく、プラットフォームとして適当であったと言える。

5.3. 作成したシステムの課題

課題としては大きく 2 点ある。1 点目は目的地へ到達するための経路追従の見直しである。4.3 節より、走行パラメータが適切でなく走行経路から外れてしまい、目標地点へ到達できなかった。そのため適切な走行パラメータを決定するために経路追従プログラムを見直す必要がある。また、現状では走行パラメータをそのまま反映させるだけであり、適切に走行パラメータをロボットの走行へ反映させるために PID 制御といった制御手法の導入も求められる。

2 点目は時間制約についてである。自律移動システムの処理の流れを図 6 に示す。センサデータの収集は走行のために最新のセンサ情報を取得し利用する必要があり、ハードリアルタイムである。対して SLAM、走行パラメータの決定は処理に時間がかかる場

合に、ロボットはその場で停止すればよいためソフトリアルタイムとなる。また、アクチュエータの制御は走行パラメータを適切なタイミングで反映させる必要があるためハードリアルタイムである。

本研究で作成したシステムの処理の流れを図 7 に示す。図に示す通り、速度パラメータが反映されるタイミングは計算終了後のタイミングであり、反映される走行パラメータは計算時間分古いデータによって計算された走行パラメータとなる。また、動作環境によっては更なる精度が必要であり、精度を上げるほど処理時間が必要となる。

そのため、図 8 に示すような処理構成が望ましいと考察する。RTOS 側では、ハードリアルタイムであるセンサデータの一定間隔での収集やアクチュエータの制御を行う。また、LiDAR のデータから近距離に障害物が存在した場合は即時停止を行う。Linux 側では SLAM の前処理として、内部センサデータのフィルタリングを行う。また、外部デバイスによる SLAM で計算された経路情報から走行パラメータを決定する。正確な自己位置推定や環境地図の構築といった SLAM の処理、経路計画は計算量が多いため、計算能力の高い外部の計算用デバイスで処理を行うことで精度や処理速度を向上が見込める。現状の課題として、このようなシステムを実現するために RTOS 側での外部デバイスとの通信を実装する必要がある。

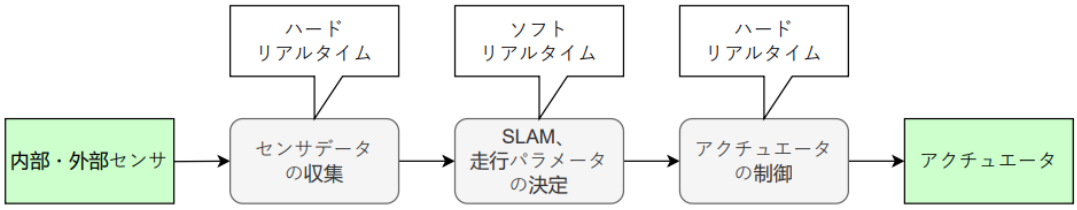

図 6 自律移動システムの処理の流れ

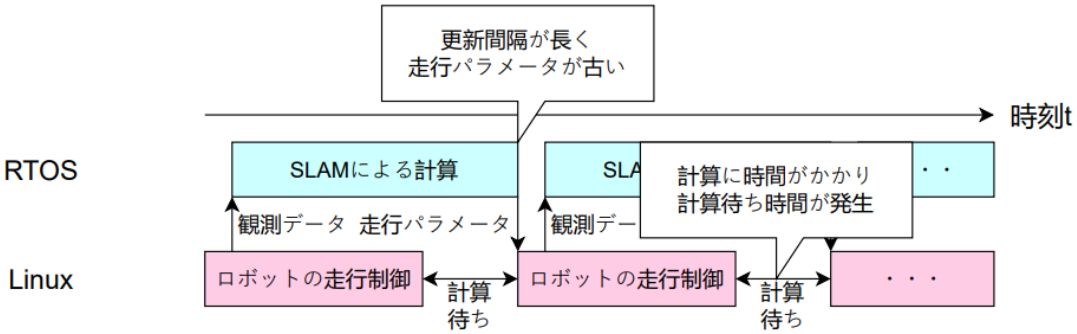

図 7 作成した自律移動システムの処理構成

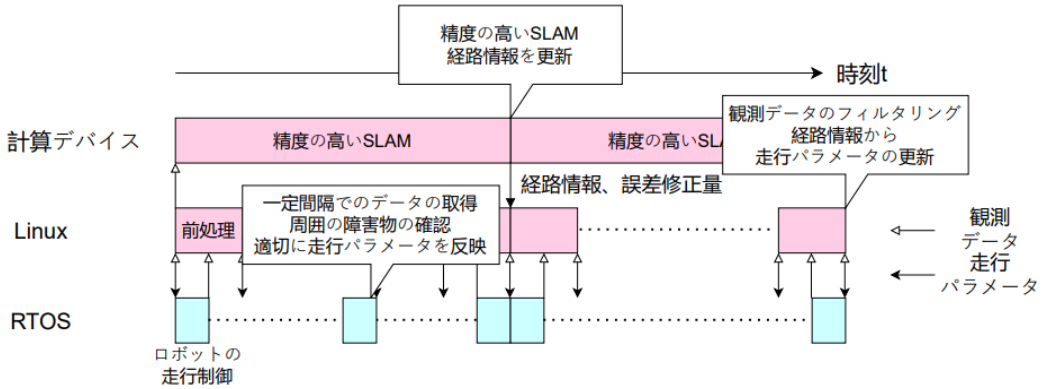

図 8 時間制約を意識した自律移動システムの処理構成

6. まとめ

本研究では、RTOS と Linux とを異なるコアで動作させ、共存させる SOLID for Raspberry Pi を用いて複数の役割を持ったデュアル OS 環境での自律移動システムの作成を行い、その評価を行った。SOLID for Raspberry Pi を開発プラットフォームとして使用した所感を述べ、デュアル OS 環境の実現と開発における有用性を考察し、用いることで効率的な開発を行うことができると分かった。作成したシステムでは、各 OS に役割を持たせることで SLAM の高速化効果の確認や様々なデバイスとの通信を行い自律移動システムを実現した。走行試験では正確な自己位置推定と環境地図の作成を行うことで目的地への経路を導出し、目的地へ進行する動作を確認できた。しかしながら、自律移動については、目的地へ到達することができなかった。

今後の課題と展望として、目的地へ到達するための経路追従の見直し、カルマンフィルタの適用を進める。また、RTOS 側で外部デバイスとの通信を実現し、時間制約を意識したシステムの構築を検討する。

7. 参考文献

[1] 藤澤行雄,品川雅臣,高島光,村上倫,石本裕介,米田真之, "諸解 車載ネットワーク CAN,CAN FD,LIN,CXPL,Ethernet の仕組みと設計のために", 初版,pp.9-14, 日刊工業株式会社,2022 年

[2] 本田晋也,山本椋太, "車載制御システム向け次世代プロセッサの仮想化支援機能を用いたハイパーバイザー" , 研究報告システムと LSI の設計技術(SLDM),情報処理学会,2019 年

[3] SOLID for Raspberry Pi 4 _ SOLID - enjoy Development, https://solid.kmckk.com/SOLID/solid4rpi4, 最終閲覧日:2024/9/7

[4] 自動運転車の安全技術ガイドライン, 国土交通省自動車局, 2018/9/12, https://www. mlit.go.jp/common/001253665.pdf, 最終閲覧日:2022/12/16

[5] openslam_gmapping, https://github.com/ros-perception/openslam_gmapping, 最終閲覧日:2024/9/7