Vol.2014-SLDM-165 No.9
Vol.2014-EMB-32 No.9
2014/3/15

組込みシステム向け軽量スクリプト言語へのアクセス制御機構の導入

熊谷 康太[1,a)]　石川 拓也[1]　本田 晋也[1]　高田 広章[1]

概要：　近年，組込みシステムの大規模化，複雑化が進んでいる．そのため，組込みソフトウェア開発における機能更新の容易性と生産性の向上が重要となっている．また，組込みシステムの大規模化と複雑化により，十分に検証することが困難なシステムが増加しており，安全性の担保も重要となっている．一方で，組込みシステム向け軽量スクリプト言語が開発されており，この言語は開発における機能更新の容易性と生産性が高いという性質を持っている．しかし，安全性を担保するには，まだ機能の面で不足する点が存在するという問題がある．本研究では，組込みシステム向け軽量スクリプト言語である mruby へアクセス制御機構を導入し，安全性を担保できる実行環境を提案する．提案するアクセス制御機構は，リアルタイム制御アプリケーションを用いて評価する．評価の結果から，mruby に対してアクセス制御機構を導入する影響は小さいと考える．

キーワード：mruby，アクセス制御，組込みシステム，軽量スクリプト言語

1. はじめに

近年，組込みシステムの大規模化，複雑化が進んでおり，組込みソフトウェア開発における機能更新の容易性と生産性の向上が重要となっている．組込みソフトウェア開発における機能更新の容易性とは，出荷後のシステムや，稼働しているシステムに対して，容易に機能を変更することができる性質のことを指す．さらに，開発したシステムの安全性を担保することも重要となっている．組込みシステムでは機器を制御することが多く，システムに不具合が発生して誤った操作が実行されると，制御機器そのものや，周囲を危険にすることがあるためである．従来は，十分にシステムを検証することで安全性を担保できたが，システムの大規模化と複雑化に伴い，十分に検証することが困難なシステムが増加している．そこで，十分に検証できないシステムであっても，安全性を担保できる性質が重要となっている．例えば，システムの機能の一部に不具合が発生したとしても，別の機能には影響を波及させない機構によって安全性の担保を実現できる．

一方で，組込みシステム向け軽量スクリプト言語 mruby が開発されている．mruby を用いた開発は，組込みソフトウェアの開発で広く用いられている C 言語よりも，機能更新の容易性と生産性が高いという性質を持つ．さらに，mruby は，C 言語と併用して機能を開発できるという性質も持つため，C 言語で開発してきた資産を再利用しつつ，mruby を新たな機能開発に利用することができる．しかし，mruby は，安全性を担保する機能が不足している．

本研究では，安全性の担保を要件とする組込みシステムを対象とし，mruby を用いた場合でも安全性の担保が可能な実行環境を提案する．はじめに，本研究で対象とするシステムと，システムが求める要件を定義し，次に， mruby を用いて，満足できない要件を明らかにする．そして，すべての要件を満足するために，mruby にアクセス制御機構を導入する．アクセス制御機構とは，アプリケーションなどのアクセス主体から，OS オブジェクトやハードウェアといった，リソースへのアクセス要求が生じた時に，リファレンスモニタと呼ばれる機構によって，要求されたアクセスを許可するか拒否するか判定することで，リソースへのアクセスを制御する機構である．そして，mruby にアクセス制御機構を導入した影響について，リアルタイム制御アプリケーションを用いて評価する．mruby にアクセス制御機構を導入しない場合と導入する場合を比較して，アプリケーションの実行時間の増加量を示す．

本論文の構成は，次のとおりである．まず 2 章で，mruby について述べ，3 章で， 本研究の対象システムと対象システムに求められる要件について述べる．そして 4 章で

[1] 名古屋大学
Nagoya University, 464–8601, Japan
[a)] kumagai@ertl.jp

mruby へのアクセス制御機構の導入について述べ，5 章で，導入したアクセス制御機構の評価実験について述べる．最後に 6 章で本論文をまとめる．

2. mruby

大規模，複雑化し，高品質，短納期，高い保守性が求められる組込みソフトウェア開発において，これらの課題を解決する目的で，mruby の開発が行われている [1]．mruby を C/C++ 言語で実装されたホストアプリケーションに組み込むと，そのホストアプリケーションの一部を Ruby で実装する事が可能となる．

2.1 mruby の構成

mruby は実行環境の内部が分割されており，組込み機器の多様な特性に応じて不必要なモジュールを取り外すなど，カスタマイズして利用できる．実行環境は， Ruby ソースコードの構文解析を行うパーサ，抽象構文木からバイトコードを生成するジェネレータ，バイトコードをデコードして実行する仮想マシン (mruby VM) からなる．

バイトコードをホスト PC 上でファイルに書き出すツール (バイナリ出力ツール) も mruby のパッケージに含まれる．mruby におけるバイトコードとは，メモリ上に展開されている状態のデータ形式を意味し，このままでは，バイトコードが実行される CPU のエンディアンやメモリ・アライメントなどに依存する．バイナリ出力ツールは，このバイトコードをシリアライズし，機種依存性のないバイナリファイル (mrb ファイル) として書き出す機能を持つ．mrb ファイルは，同じ mruby VM 上であれば，CPU や OS の種類に関係なく，同じように実行できる．

Ruby のソースコードを機密上，機器に直接搭載したくない場合などは，mrb ファイルを mruby VM と共に機器に組み込むことが可能である．ファイルシステムが利用できない場合は，mrb ファイルの情報を C 言語のソースコード中に直接埋め込むこともできる．この場合，パーサとジェネレータを実行環境に含まなくて良いため，省メモリを実現できる．また，パーサとジェネレータを実行環境に含まなくても，バイトコードを実行する機器とバイトコードを生成して発信する機器間で通信して，実行環境側の機器のバイトコードを更新することで動的に処理内容を変更できる．なお，実行環境が持つ各機能，例えば，パーサ，ジェネレータなどは C API を介して，C/C++ 言語で実装されたホストアプリケーションから呼び出せる．C API とは，mruby VM 上で動作するアプリケーションから C/C++ 言語で実装された機能を呼び出す，または，C/C++ 言語で実装されたアプリケーションから mruby VM の機能を呼び出すための API である．この C API を用いることで，C/C++言語で実装された関数を，mruby VM 上で動作するアプリケーションから呼び出すことができるメソッドとして，mruby VM に登録できる．登録が行われた mruby VM によりバイトコードを実行すると，バイトコード中で該当するメソッドが呼び出された時，対応する C 言語の関数が呼び出される．このため，mruby VM に対して，デバイスドライバや OS の API を mruby 上で動作するアプリケーションから呼び出せるメソッドとして登録しておくことで，mruby アプリケーションからのデバイスや OS オブジェクトへのアクセスが可能となる．

mruby VM は 多様な環境に組み込まれて利用されることが想定されているため，mruby VM を実行するために必要なライブラリ関数は多くない．しかし，mruby VM が演算を行うために必要なメモリ領域を確保，解放するために，malloc 関数 と free 関数は， 最低限提供する必要がある．ここで，mruby VM がアクセスするメモリ領域は，malloc 関数で確保された領域に限定されるという性質を持つ．

3. 対象システムと要件定義

本章では，まず本研究で対象とするシステムを定義する．次に，対象とするシステムに求められる要件を定義する．そして，既存の mruby を用いた実行環境では満足できない要件について整理する．

本研究で対象とするシステムは，ハードリアルタイムかつ，停止させられない機能 (HARD-APP) とソフトリアルタイムかつ，機能の更新が必要な機能 (SOFT-APP) が共存し，かつ OS を使用することが必須である組込みシステムとした．HARD-APP は，十分に検証されるため，信頼でき，かつ，安全に関する機能を実現し，SOFT-APP は，検証が困難なため，信頼性が無く，かつ，安全には直接影響しない機能を実現することを前提とする．対象システムには，分散ネットワークシステムを含み，SOFT-APP が動作する機器が，スタンドアロンで動作するシステムと，別の機器と通信するシステムの両方を含む．

次にシステムが求める要件は，次のように定義した．

1 SOFT-APP の機能を生産性高く開発できること
2 SOFT-APP の機能を動的に変更できること
3 メモリや CPU などの必要なハードウェアリソースが小さいこと
4 SOFT-APP の不具合による影響が HARD-APP に及ばないこと
  4-1 SOFT-APP がアクセスできるメモリ領域を限定できること
  4-2 SOFT-APP が HARD-APP の OS オブジェクトにアクセスしないこと
  4-3 HARD-APP の実行中に SOFT-APP が実行されないこと
5 SOFT-APP の不具合による影響が周辺機器 (制御対象や通信機器) に及ばないこと
  5-1 SOFT-APP が許可されていないハードウェアに

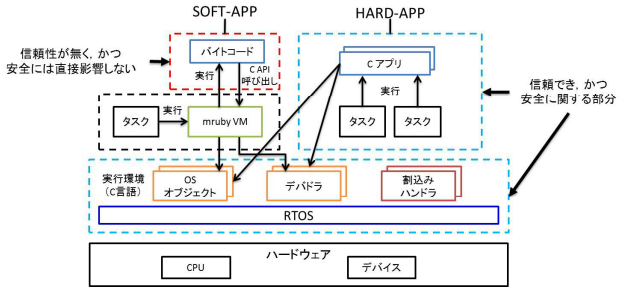

図 1 RTOS + mruby によるシステム構成
Fig. 1 System architecture with an RTOS and mruby

アクセスしないこと

5-2 SOFT-APP が許可されていない用途や時間間隔でハードウェアを使用しないこと

本研究では，mruby をリアルタイム OS（RTOS）上で動作させる実行環境（RTOS + mruby）を用いる．RTOS としては TOPPERS/ASP カーネル（ASP カーネル）を用いる．RTOS を使用する理由は，Linux のような汎用 OS では，HARD-APP を実現することが困難であるためである．対象システムを RTOS + mruby を用いて実現する場合のシステム構成を図 1 に示す．HARD-APP は，RTOS 上のタスクによって，C 言語で実装されたアプリケーションを実行することで，実現される．RTOS の API やデバイスドライバは C 言語で実装されているため，アプリケーションが直接呼び出すことが可能である．SOFT-APP は，RTOS 上のタスクが mruby VM を実行することで実現される．RTOS の API やデバイスドライバは，アプリケーションから C API を利用して呼び出すことが可能である．また，1 つの mruby VM は 1 つのタスク上で実行されることと，SOFT-APP の機能を更新するために，HARD-APP が mruby VM を実行するタスクを起動および，終了させる機能を持つことも前提とする．

以上の実行環境によって，満足できる要件を述べる．まず，要件 1 を満足できる．これは，SOFT-APP を Ruby を用いて開発できるためである．次に，要件 2 を満足できる．これは，バイトコードを更新することで，SOFT-APP の機能を更新できるためである．そして，要件 3 も満足できる．これは Linux に比べ，RTOS + mruby が必要とするメモリや CPU などのハードウェアリソースが小さいためである．また，要件 4-1 も満足できる．これは，mruby VM がアクセスするメモリ領域は malloc によって確保した領域に限定できるためである．

しかし，他の要件は満足できない．バイトコードに不具合が存在した場合，SOFT-APP が RTOS の API を誤って使用し，HARD-APP の OS オブジェクトにアクセスしたり (要件 4-2)，HARD-APP のタスクよりも SOFT-APP のタスクの優先度を高くする (要件 4-3) ことが起こりうる．また，mruby VM に登録されたデバイスドライバが許可されていない用途や時間間隔で使用される (要件 5-2)，例えば，SOFT-APP が制御機器を危険な状態にする値を引数としてデバイスドライバを呼び出すことが起こりうる．さらに，分散ネットワークシステムの場合，バイトコードを実行する機器上で動作する SOFT-APP が，デバイスドライバを使用して，許可されていない短い時間間隔でメッセージを送り，通信帯域を圧迫することなども起こりうる．また，機能の更新に伴い，SOFT-APP の仕様が変更され，変更前はアクセスが許可されていたハードウェアへのアクセスが，許可されなくなった場合，更新したバイトコードに古い機能が残っており，許可されていないハードウェアにアクセスする (要件 5-1) ことが起こりうる．

4. mruby へのアクセス制御機構の導入

本章では，mruby へのアクセス制御機構の導入について述べる．まず，前節で述べた全ての要件を満足させるアクセス制御機構の仕様について述べる．そして，アクセス制御機構の実装について述べる．最後に，アクセス制御で違反した mruby VM を実行するタスク（VM タスク）を強制的に終了するための機能について述べる．

4.1 アクセス制御機構の仕様

アクセス制御機構は，mruby VM が OS オブジェクトや，デバイスへアクセスする際に，リファレンスモニタと呼ばれるモニタ機構によって，アクセスを許可してよいか判定する．判定するための情報は，ポリシファイルとよばれるファイルが保持しており，リファレンスモニタはこれを参照する．また，アクセス制御機構は，mruby VM がアクセス制御で違反した場合に，VM タスクを強制的に終了させる機能も持つ．要件を満足するために，リファレンスモニタに求められる機能は以下のとおりである．

機能 1：実行できる関数を制御できること
機能 2：関数に渡す引数を制御できること
機能 3：関数の実行間隔を制御できること

まず，機能 1 により，許可されていないハードウェアにアクセスするデバイスドライバの呼出しを防止できる (要件 5-1)．次に，機能 2 により，mruby VM が RTOS の API 呼び出し時に渡した引数に応じてアクセスを制御することで，HARD-APP の OS オブジェクトへのアクセスを防止できる (要件 4-2)．また，同様に，HARD-APP の タスクの優先度より高い優先度に SOFT-APP のタスクの優先度を設定することを防止できる (要件 4-3)．さらに，mruby VM が，制御機器を危険な状態にする引数を渡してデバイスドライバを実行することを防止できる (要件 5-2)．また，機能 3 により，分散ネットワークシステムにおいて，mruby VM がメッセージを送信するデバイスドライバを短い実行間隔で呼び出すことによって，通信帯域を圧迫することを防止できる (要件 5-2)．

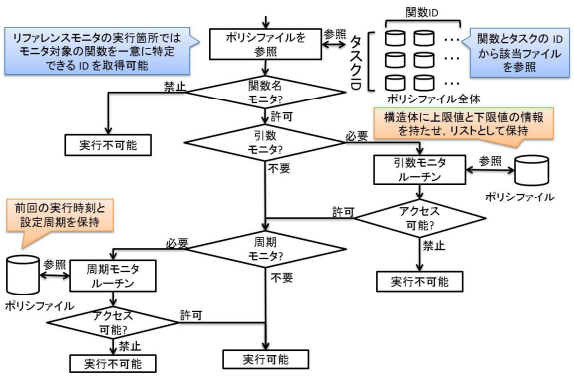

図 2 リファレンスモニタのシーケンス
Fig. 2 Sequence of the reference monitor.

4.2 アクセス制御機構の実装

アクセス制御機構の実装について述べる．まず，リファレンスモニタをフックする箇所は，mruby VM が，登録された C 言語の関数を呼び出す直前の箇所とする．リファレンスモニタの処理について，(1) 関数名モニタルーチン，(2) 引数モニタルーチン，(3) 周期モニタルーチンに分けて述べる．リファレンスモニタのシーケンスを図 2 に示す．

リファレンスモニタは，まず，VM タスクのタスク ID を取得する．また，mruby VM は，内部で，アクセス制御対象である関数を一意に特定できる ID を保持しており，リファレンスモニタは，この ID を利用する．以上の 2 つの ID からポリシファイルの中の該当する領域 (Access Control Block:ACB) を参照する．ACB は構造体となっており，各モニタルーチンが参照する情報を持つ．

関数名モニタルーチンでは，ACB から，アクセス可能か否かを示すフラグを参照し，アクセス可能か判定する．判定の結果，アクセス不可能であれば，リファレンスモニタの処理を終了し，対象の関数を実行せずに VM タスクを終了する．判定の結果，アクセスが可能であれば，次に，ACB が保持する，引数モニタが必要か否かを示すフラグに応じて，引数モニタルーチンを実行する．

引数モニタルーチンでは，関数に渡された引数が，ポリシファイルに記述された上限値と下限値の範囲内に収まっているか否か判定する．引数モニタルーチンは，モニタ対象となる引数ごとに，ACB から，各引数に関する上限値と下限値の情報を持つ構造体のリストを参照してアクセス可能か否かを判定する．ACB は，リストの先頭に存在する構造体のポインタと型を保持する．引数には符号なし整数型や，浮動小数点型などの多様な型がありうるが，引数モニタルーチンは， mruby VM の API を利用して，すべての引数を double 型として取得する．そして，取得した引数をキャスト演算により元の型に戻した後で，適正な値の範囲内かどうかを判定する．ここで，各引数の型情報は，ACB から取得した引数情報の構造体に含まれる．この構造体には，引数の上限値，下限値，モニタする引数が何番目の引数か，リストの次の構造体へのポインタとポインタの型の情報を保持している．例えば，引数が 5 つあるが，モニタ対象の引数が 3 番目の引数だけであるような場合，リストには 1 つ構造体をつなげ，この構造体の中に 3 番目の引数をモニタするように設定できる．引数を取得した後，リストの先頭から構造体を 1 つずつ取り出し，引数の型に応じて，キャストし，判定する．このとき，1 つでも，ポリシファイルの内容に違反する引数があれば，リファレンスモニタの処理を終了し，対象の関数を実行せずに VM タスクを終了する．リスト全ての構造体のモニタが終わり，違反が無ければ，次に，ACB が保持する，周期モニタが必要か否かを示すフラグに応じて，周期モニタルーチンを実行する．

周期モニタルーチンは，まず，RTOS の API を利用して，現在の実行時刻を取得する．そして， ACB に保存された，対象関数の前回の実行時刻を参照し，現在の実行時刻との差分を算出する．算出された差分が ACB が保持する実行間隔の最小値より大きければ，アクセスが可能である．そして，前回の実行時刻を現在の実行時刻で上書きする．小さい場合は，リファレンスモニタの処理を終了し，対象の関数を実行せずに VM タスクを終了する．

4.3 VM タスクの強制終了対応

VM タスクを強制的に終了させる際は，mruby VM が確保していたヒープ領域を解放する必要がある．単一のヒープ領域に対して，複数の mruby VM がそれぞれメモリ領域を確保すると，フラグメントが発生する．そこで，VM タスクごとに使用するヒープ領域を分離するために，リアルタイムメモリ割付モジュール TLSF を ASP カーネルへ導入する [2].

また，SOFT-APP の機能を更新するために，HARD-APP から VM タスクを終了させる必要があるが，VM タスクが確保したメモリ領域の解放は，そのタスクコンテキストでなければ実行できない．一方で，ASP カーネルではタスク例外という，タスクごとに例外処理を登録する機構が提供されている．タスク例外は，タスクを指定して呼び出すことができ，その例外処理は，対応するタスクコンテキストで実行される．そこで，VM タスクを強制的に終了する時には，タスク例外を用いて，その例外処理の中で，該当する VM タスクが使用していたヒープ領域の解放および，VM タスクの終了処理を実施する．実施した内容について次に述べる．

4.3.1 リアルタイムメモリ割付モジュール TLSF の導入

既存の TLSF は， API によってヒープ領域のポインタと長さを登録することで，単一の連続したヒープ領域からメモリ割付を行う．これに対し，タスクごとに異なるヒープ領域からメモリ割付を行えるように拡張する．具体的には，TLSF に登録するヒープ領域のポインタと長さの組を

多重化し，登録を行ったタスクの ID とヒープ領域を対応付けるように TLSF を拡張する．また，TLSF が，メモリ領域の確保や，解放を行う際は，呼び出したタスクの ID に対応付けられたヒープ領域を用いて，メモリ割付を行う．以上の変更により，VM タスクごとに，mruby VM が使用するヒープ領域を分離する．

4.3.2 VM タスクの管理

メモリプールとタスクプールを用いたタスク管理機能とは，VM タスクオブジェクトを複数生成し，HARD-APP のタスクから VM タスクを起動したり，強制的に終了したりするための機能である．この機能を実現するために，まず，タスクプールテーブル（TPTable）と，メモリプールテーブル（MPTable）を用意する．TPTable とは，全ての VM タスクの状態を保持するテーブルである．MPTable とは， VM タスクが mruby VM を実行するために必要なヒープ領域の状態を保持するテーブルである．VM タスクが実行中の場合， TPTable には，使用してるヒープ領域の情報が格納されている．実行中ではない場合， TPTable には 0 が格納されている．また，ヒープ領域が使用中の場合， MPTable には，使用している VM タスクのタスク ID が格納されている．使用中ではない場合， MPTable には 0 が格納されている．

VM タスクの起動処理について述べる．VM タスクを起動する場合は，まず， TPTable と， MPTable を参照し，0 が格納されているタスク ID とヒープ領域へのポインタを取得する．その後， TPTable と， MPTable を更新する．そして，起動する VM タスクに実行するバイトコードと，使用するヒープ領域の情報を VM タスクごとに割り当てられたデータ領域に書き込む．その後，RTOS の API により，該当する VM タスクを起動する．起動された VM タスクは，対応するデータ領域に書き込まれたヒープ領域とバイトコードの情報により mruby VM を起動し，バイトコードを実行する．

次に，VM タスクの終了時の処理について述べる．VM タスクは終了時にタスク例外の機構により例外処理を呼び出す．例外処理は， TPTable と， MPTable に対して，自タスクの状態と使用していたヒープ領域の状態を 0 に書き換え，自タスクを終了する RTOS の API を呼び出す．タスク例外処理は，VM タスクが正常終了する場合と，アクセス制御で違反した場合と，外部のタスクから強制的に終了させる場合に実行される．

5. 評価実験

本章では，アクセス制御機構を導入した mruby に対して行った評価実験について述べる．まず，基礎評価では，アクセス制御機構を導入したことによる実行時間オーバヘッドについて述べる．次に，リアルタイム制御アプリケーションに対して適用した評価実験について述べる．

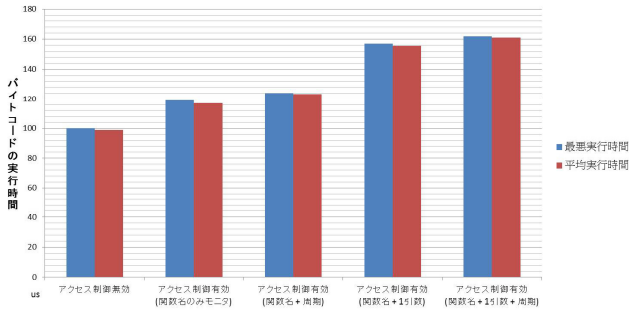

図 3 アクセス制御による実行オーバヘッド
Fig. 3 Overhead caused by the access control mechanism.

5.1 基礎評価

5.1.1 評価環境

まず実験を行った環境について述べる．評価用ボードとして，APSH2A-6A [3] を用いた．APSH2A-6A は周波数 240 MHz で動作する SH7269 プロセッサと，16 MB の SDRAM と，16 MB の Flush ROM を搭載する．

5.1.2 評価方法

実行時間オーバヘッドの評価実験は 2 種類の実験を行った．まず 1 つ目の実験では，アクセス制御機構を有効にした場合と無効にした場合で mruby VM が実行するバイトコードの実行時間を比較した．バイトコードは，mruby VM に登録された C 言語の関数を 1 度呼び出す．アクセス制御有効時は，C 言語の関数を呼び出す際に，リファレンスモニタを 1 度だけ実行する．リファレンスモニタが，(i) 関数名のみモニタ，(ii) 関数名と引数 1 つをモニタ，(iii) 関数名と周期をモニタ，(iv) 関数名と引数 1 つと周期をモニタする場合でそれぞれ 10,000 回実行を繰り返し，最悪実行時間と平均実行時間を計測した．

次に 2 つ目の実験では，mruby VM に登録する関数（リファレンスモニタがモニタ対象とする）の数を 3，10，20 と増加させて，アクセス制御を有効にした場合と，無効にした場合の実行時間の差分を計測した．バイトコードは，mruby VM に登録された C 言語の関数を 1 度呼び出す．リファレンスモニタはいずれも，関数名のモニタ，引数モニタ (引数 1 つをモニタ)，周期モニタ全てを実行する．試行回数は 10,000 回であり，アクセス制御を有効にすることによる最大増加時間と平均増加時間を計測した．

5.1.3 評価結果

まず 1 つ目の実験結果について述べる．最悪実行時間と平均実行時間の計測結果を図 3 に示す．

まず，アクセス制御無効時とアクセス制御で関数名のみモニタする場合の実行時間の差からアクセス制御により関数名のみモニタする場合最大で 19μs オーバヘッドが生じることがわかる．また，アクセス制御有効時で関数名のみモニタする場合と，関数名と周期をモニタする場合の実行時間の差から，周期をモニタする場合，最大でさらに 5μs オーバヘッドが生じることがわかる．そして，アクセス制

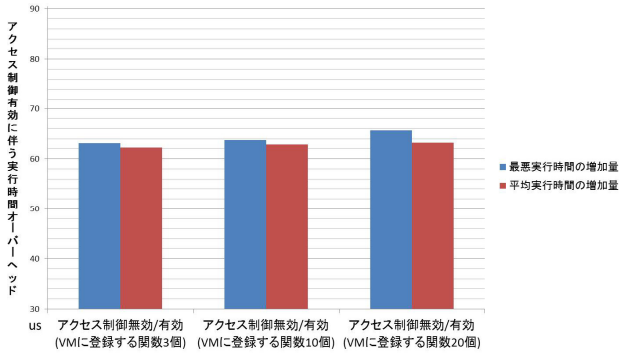

図 4 モニタする関数を増加させた場合の実行オーバヘッドの増加量
Fig. 4 Amount of increase in the overhead

御有効時で関数名のみモニタする場合と，関数名と引数 1 つをモニタする場合の実行時間の差から，引数 1 つをモニタすることで，最大でさらに 38μs オーバヘッドが生じることがわかる．最後に，平均実行時間と最悪実行時間の差は，全ての項目について 2μs 以内である．このことから，実行オーバヘッドの見積りは可能であり，SOFT-APP に求められる，ソフトリアルタイム性への影響は小さいと考えられる．

次に 2 つ目の実験結果について述べる．最悪実行時間と平均実行時間の計測結果を図 4 に示す．

上記の結果から，リファレンスモニタの実行時間オーバヘッドの平均は，62μs から 64μs の間に収まっており，モニタする関数の数を増加しても，一定であった．これは，関数の ID とタスクの ID から ACB へオーダ 1 でアクセスできるためであると考えられる．

5.2 リアルタイム制御アプリケーションへの適用性評価

アクセス制御機構を導入した mruby をリアルタイム制御アプリケーションに適用して行った評価実験について述べる．リアルタイム制御アプリケーションは，PUPPY という倒立二輪ロボットを制御するアプリケーションである．

5.2.1 評価環境

評価環境は，組込みボードとして APSH2A-6A を使用した．また，制御対象として，PUPPY を使用した．構成は，APSH2A-6A と PUPPY を CAN232 というアダプタを介して接続し，APSH2A-6A はシリアルで通信し，PUPPY は CAN で通信する．PUPPY はジャイロセンサとロータリエンコーダセンサの値を送信し，mruby + RTOS を搭載した，APSH2A-6A が倒立制御に必要なモータの出力値を算出，送信する．このアプリケーションのデッドラインは 10ms であり，それ以内に APSH2A-6A が応答できないと PUPPY が転倒する．また，APSH2A-6A がセンサ値を取得してから PUPPY へ制御値を送信するまでの応答時間を計測するために，CAN のパケットをモニタリングする MLT 3rd Series(2006)[4] を使用した．

表 1 アクセス制御機構によるオーバヘッド
Table 1 Overhead caused by the access control mechanism.

	最悪実行時間	平均実行時間
アクセス制御有効	4.22ms	4.10ms
アクセス制御無効	4.12ms	4.00ms

5.2.2 評価方法

アクセス制御を無効にした場合と有効にした場合で，応答時間を比較した．試行回数はそれぞれ 500 回である．mruby VM は PUPPY からセンサ値を取得してから，制御値を送信するまでに 5 回 C 言語の関数を呼び出す．アクセス制御有効時は呼び出す C 言語の関数に対し関数名のみモニタするように設定した．

5.2.3 評価結果

実験の結果を表 1 に示す．この結果から，アクセス制御機構により，100μs 応答時間が増加していることが分かる．関数名のモニタは 1 回につき最大 19μs オーバヘッドがあったため，5 回モニタすることで 100μs 実行時間が増加するのは妥当な結果と考えられる．応答時間は全体でおよそ 2.5%増加しており，このアプリケーションのデッドラインから，アクセス制御機構を導入することによる実行時間のオーバヘッドは小さいと考える．

6. おわりに

本研究では，安全性の担保を要件とする組込みシステムを対象とし，mruby を用いて要件を満足させる実行環境を提案した．まず，研究で対象とするシステムを明確化し，性質の異なる機能が共存する組込みシステムを対象とした．次に，対象システムに求められる要件を定義した．そして，定義した要件に対し，mruby で満足できる要件とできない要件を整理した．その結果，満足できないと分類された要件を満足させる機構として，アクセス制御機構を検討し，mruby へ実装して，実行時間のオーバヘッドを評価した．評価の結果から，アクセス制御機構を mruby に導入する影響は小さいと考える．

今後の課題としては，ポリシファイルを動的に書き換える機能の実装および評価を行うことが挙げられる．これは， SOFT-APP の機能を動的に変更することに伴い，ポリシファイルを動的に変更する必要性があると考えられるためである．

参考文献

[1] NPO 法人軽量 Ruby フォーラム , “http://forum.mruby.org/,”.
[2] TOPPERS 新世代カーネルへのマイグレーションガイド , “http://www.toppers.jp/docs/tech/migration-120.pdf,”.
[3] APSH2A-6A , “http://www.apnet.co.jp/support/man/ap_sh2a_6a.pdf,”.
[4] MLT 3rd Series , “http://www.prism-arts.co.jp/mlt/,”.