次世代ロボット知能化技術開発プロジェクト
統合検証作業

RTC 開発ガイドライン

0.1 版

2011 年 2 月 28 日

RTC 再利用技術研究センター

目次

1. はじめに

1.1. 目的

本書は、「次世代ロボット知能化技術開発プロジェクト」の「ロボット知能ソフトウェア再利用性向上技術の開発」における、
統合検証作業に関する RTC の実装規約を定めた文書である。

1.2. 本書での書式

本文書で使用している記号・書式の目的を下表に示す。

表.書式一覧

No.	記号・書式	目的
1	※	注意書き
2	赤色の文字	注記

1.3. 用語の定義、略語

表.用語の定義、略語一覧

No.	表記	意味
1	RTM	RT-Middleware
2	RTC	RT-Component
3	RTS	RT-System
4	RTCD	RTC-daemon
5	coil	Common Operating-system Infrastructure Layer 移植性向上のための OS 抽象化層
6	RT-Preempt	RealTime Preemption Patch。標準 Linux カーネルに、このパッチを当てる事で、カーネルはハードリアルタイム性能を持つ。
7	分解運動速度制御	Resolved motion rate control エンドエフェクタに定めた座標軸方向に指定の速度で動くように、関節の速度・角速度を分解し各関節に割り当てて制御すること。

1.4. 参考資料

本書を作成するにあたり参照した文書・資料を下表に示す。

表.参考資料一覧

No.	文書名	備考 / URL
1	OpenRTM-aist official website	http://www.openrtm.org/

2. ログ出力ガイドライン

本項では RTC 実装時において、ログ出力を実装するためのガイドラインについて説明する。

ログ出力について、「開発及びデバッグ時にコードに手を加えず、ログ出力レベルの切り替えが可能であること」を
条件とし、原則として RTM に実装されているロガークラス(RTC::Logger)を使用する。

2.1. 実装ガイドライン

● ログレベル利用規約
ログレベルは以下の用途で利用することとする。

ログレベル	確認者	説明
FATAL	―	使用しない。
ERROR	運用者/開発者	継続不可能な問題が起きた場合の手掛かりとなる情報出力に使用する。
WARN	運用者/開発者	継続可能な問題が起きた場合の手掛かりとなる情報出力に使用する。
INFO	開発者	デバッグ時に必要な情報の内、毎周期出力する必要のない情報出力に使用する。 但し、DEBUG では RTM 本体のログも大量に出力されるため、毎周期出力する場合でも、混在するのを避けたい場合は INFO を使用する。
DEBUG	開発者	デバッグ時に必要な情報の内、毎周期出力する必要のある情報出力に使用する。(ExecutionContext の実行周期計測等)
TRACE	開発者	デバッグ時に必要な情報の内、ディスク容量やオーバーヘッドを省みない膨大な情報出力に使用する。(通信データのダンプ等)
VERBOSE	―	使用しない
PARANOID	―	使用しない

● ログの埋め込み方法
RTC のヘッダファイルに以下の通り SystemLogger.h の追加を行う。

```
#include <rtm/SystemLogger.h>
```

ログはログレベル利用規約に沿って、以下の通り埋め込みを行う。

```
RTC_INFO(("EC Rate:[%f]" d_tm ));
```

埋め込んだログは実行時、以下のように出力される。

```
Nov 17 17:25:52 INFO: LogTest0: EC Rate:[0.001289]
```

- 書式は、[時間][ログレベル]: [サフィックス]: [メッセージ]
- [時間][ログレベル][サフィックス]は自動付加。
- RTC 内部で出力した場合、[サフィックス]はインスタンス名。

2.2. 設定ガイドライン

● ログファイル名の指定

ログファイル名の指定は以下の通り、変更を行う。

ログファイルがバラバラに出力されないよう、予め格納パスを設定し、管理を行う。

＜デフォルト＞

logger.file_name: ./rtc%p.log

＜変更後＞

logger.file_name: /[ログ格納パス]/[RTC 名].log

● 日付フォーマット

ログに記載する日付・時刻のフォーマット指定は下表の通り。

原則的には、デフォルト値（[%b %d %H:%M:%S]）を使用することとする。

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	the standard date and time string
%d	day of the month, as a number (1-31)
%H	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12). Note: some versions of Microsoft Visual C++ may use values that range from 0-11.
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, sunday as the first day
%w	weekday as a decimal (0-6, sunday=0)
%W	week of the year, monday as the first day
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

● 時間計測(coil)

RTM に実装されているロガークラスの時間の出力最小単位は sec であり、msec や usec までの時刻を取得したい場合は、OS 依存のライブラリを用いないよう、OS 抽象化層である coil(coil)を極力利用すること。

実行コンテキスト周期計測(msec)処理のサンプルコードは以下の通り。

● ヘッダファイル

＜追加ヘッダ＞

```
#include <coil/Time.h>
```

＜メンバ変数＞

```
RTC::Time now_tm last_tm;
float d_tm
```

● ソースファイル

```
float calc_tm_dif(RTC::Time tm1 RTC::Time tm2) {
   return (float)(tm1.sec - tm2.sec)
      + (float)(1000000000 + tm1.nsec - tm2.nsec)/ 1000000000.0 - 1.0;
}

RTC::ReturnCode_t Sample::onActivated(RTC::UniqueId ec_id)
{
   coil::TimeValue tv;

   tv = coil::gettimeofday();
   last_tm.sec  = tv.sec(); now_tm.sec;
   last_tm.nsec = tv.usec() * 1000;

   return RTC::RTC_OK;
}

RTC::ReturnCode_t Sample::onExecute(RTC::UniqueId ec_id)
{
   coil::TimeValue tv;

   // 今回時間測定
   tv = coil::gettimeofday();
   now_tm.sec  = tv.sec();
   now_tm.nsec = tv.usec() * 1000;

   // 前回の周期との差分計算
   d_tm = calc_tm_dif(now_tm, last_tm);   // delta time

   last_tm.sec  = now_tm.sec;
   last_tm.nsec = now_tm.nsec;

   // 実行コンテキスト周期計測結果のログ出力
   RTC_DEBUG(("EC Rate:[%f]",d_tm ));
   return RTC::RTC_OK;
}
```

2.3. 注意事項

- 複数 RTC からのログ出力
  単一 RTC ではなく、複数 RTC から同一ログファイルへの書き込みを行いたい場合、
  マネージャ機能によるマルチスレッド動作時は、シリアライズしてバッファリングされる対応がされているが、
  マルチプロセスについては特に対応は行われておらず、ロガーが使用している std::basic_ostream の挙動に準ずる。
- rtclog のスコープに関する問題
  RTC 本体（DataFlowComponentBase）以外のクラス内（サービスポート、コールバック）でログ出力を行う場合、rtclog のスコープに関するコンパイルエラーが発生するため、その場合は以下の対応を行う。
  クラス内に以下の宣言を追加。

```
RTC::Logger rtclog;
```

またコンストラクタに「rtclog("xxxxxx")」を追加。

```
SampleSVC_impl::SampleSVC_impl()
:rtclog("SampleSVC_impl")
{
  // Please add extra constructor code here.
}
```

3. モジュール動作の最適化ガイドライン

本項でハードウェアデバイス制御等のハードリアルタイム性を要求される条件下において、また
リソース(CPU、メモリ、ネットワーク、ディスク)が限られた条件下において、モジュールをより効率的に
動作させるための最適化ガイドラインについて説明する。

想定する適用パターンは以下の通り。

- ロボットアームの手先位置フィードバック制御
- 移動ロボットの軌道追従フィードバック制御

3.1. データポートコールバック

- 適用指針
  粒度の細かい複数の RTC を用いたフィートバック制御を行う場合、onExecute()の周期実行の中でデータの送受信を
  行うと、余計なデータ送信によるオーバーヘッドや実行周期のズレによるデータ遅延等の問題が懸念される。

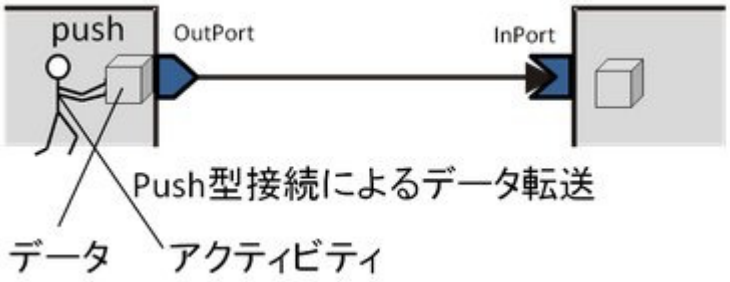
  この対策として RTC 間データ通信についてはデータポートコールバック(addConnectorDataListener)を用いた実装を推奨する。
  また RTC 間のコネクタのデータフロー型は「push」、サブスクリプション型は同期型送信方式である「flush」を推奨する。
  その他コネクタの属性については、OpenRTM-aist デベロッパーズガイド(データポート (基礎編))を確認の上、設定を行うこと。

- コールバックイベント説明
  現状バッファ書き込みまたは読み出しイベントに関連するリスナのタイプとコールバックイベントは以下の通りであり、目的に応じて適宜使用する。

  A. 共通系

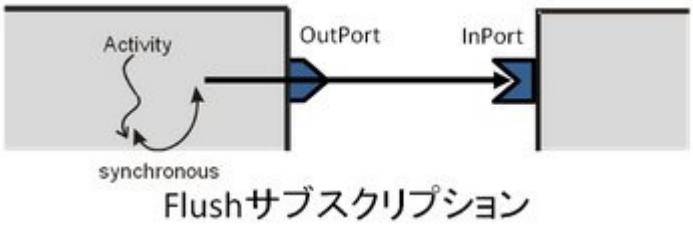
コールバック	呼び出しタイミング
ON_CONNECT	接続確立時
ON_DISCONNECT	接続切断時

  B. push 型
  OutPort が InPort にデータを送るデータフロー型。
  サブスクリプション型として同期型送信方式の「Flush」、及び非同期型送信方式の「New」、「Periodic」の 3 種類が選択できる。



B-1. push-Flush 型

OutPort から InPort へデータを push するとき、OutPort の write 関数内で直接データの送信を行う。

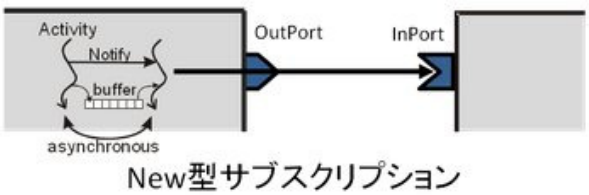


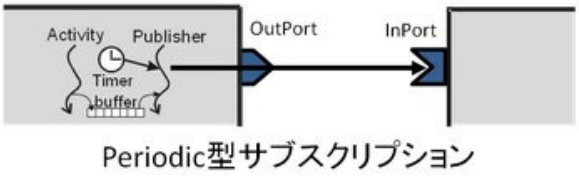
送信(OutPort)		受信(InPort)	
コールバック	呼び出しタイミング	コールバック	呼び出しタイミング
ON_SEND	InPort への送信時	ON_RECEIVED	InPort への送信完了時
ON_RECEIVED	InPort への送信完了時	ON_BUFFER_WRITE	バッファ書き込み時

B-2. push-New 型・push-Periodic 型

new 型と periodic 型には、publisher という送信のためのスレッドが接続毎に用意され、OutPort の write() 関数を呼ぶと、

データは一旦バッファに書きこまれ、送信は publisher の別スレッドが行う。

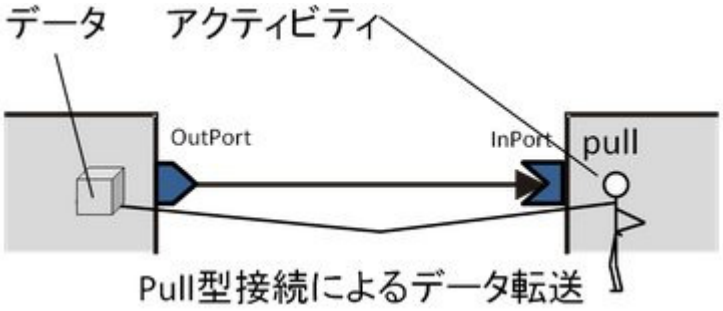




送信(OutPort)		受信(InPort)	
コールバック	呼び出しタイミング	コールバック	呼び出しタイミング
ON_BUFFER_WRITE	バッファ書き込み時	ON_RECEIVED	InPort への送信完了時
ON_BUFFER_READ	バッファ読み出し時	ON_BUFFER_WRITE	バッファ書き込み時
ON_SEND	InPort への送信時		
ON_RECEIVED	InPort への送信完了時		

C. pull 型

InPort から OutPort に問い合わせてデータを取ってくるデータフロー型。



送信(OutPort)		受信(InPort)	
コールバック	呼び出しタイミング	コールバック	呼び出しタイミング
ON_BUFFER_READ	バッファ読み出し時	ON_RECEIVED	InPort への送信完了時
ON_SEND	InPort への送信時	ON_BUFFER_WRITE	バッファ書き込み時

※ pull 型では受信(InPort)側のタイミングでデータが送信され、送信(OutPort)側では
ON_RECEIVED を受けることが出来ない。
ON_RECEIVED を使用する場合は push 型を使用すること。

D. 準異常系/異常系

各種エラーイベントについては以下の通り。発生タイミングについてはデータポート間接続時の
[Buffer length]／[Buffer full policy]／[Buffer empty policy]の設定内容に依存する。
OpenRTM-aist デベロッパーズガイド(システムエディタ（ポート間の接続 編）)を確認の上、設定を行うこと。

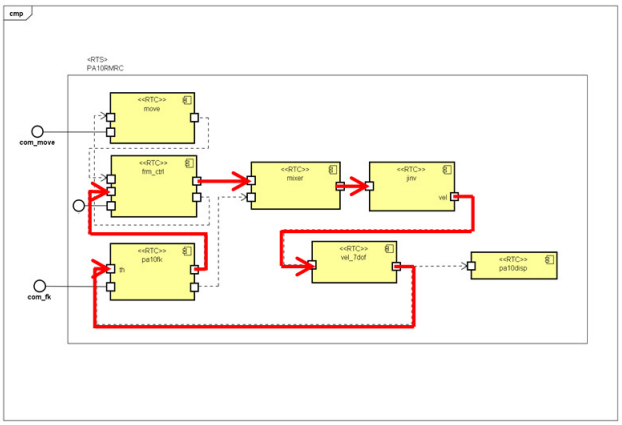
コールバック	呼び出しタイミング
ON_BUFFER_FULL	バッファフル時
ON_BUFFER_WRITE_TIMEOUT	バッファ書き込みタイムアウト時
ON_BUFFER_OVERWRITE	バッファ上書き時
ON_RECEIVER_FULL	InPort 側バッファフル時
ON_RECEIVER_TIMEOUT	InPort 側バッファタイムアウト時
ON_RECEIVER_ERROR	InPort 側エラー時
ON_BUFFER_EMPTY	バッファが空の場合
ON_BUFFER_READ_TIMEOUT	バッファが空でタイムアウトした場合
ON_SENDER_EMPTY	OutPort 側バッファが空の場合
ON_SENDER_TIMEOUT	OutPort 側タイムアウト時
ON_SENDER_ERROR	OutPort 側エラー時

● 性能評価

調査には「アームの分解運動速度制御 RTC 群」のシミュレータ動作環境を使用し、コールバック時と非コールバック時におけるフィードバック周期（下図の赤矢印のデータフロー参照）の比較計測を行った。

調査実施環境は以下の通りとする。

OS	Ubuntu10.04LTS(kernel 2.6.31-11-rt)
CPU	Intel Core i5-520M vPro
メモリ	2 GB DDR3 SDRAM
使用 RTC	アームの分解運動速度制御 RTC 群
ExecutionContext の実行周期	1000



上記 RTC 群のポート間の Connector Profile は全て以下の通り。

属性	設定内容
Interface Type	corba_cdr
Dataflow Type	push
Subscription Type	flush

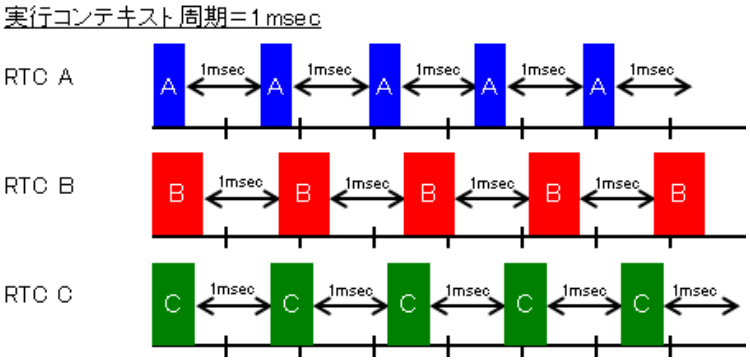
結果は、コールバック時が 4.34msec、非コールバック時が 5.38msec となり、性能向上の確認が出来た。

3.2. RTCD

- 適用指針

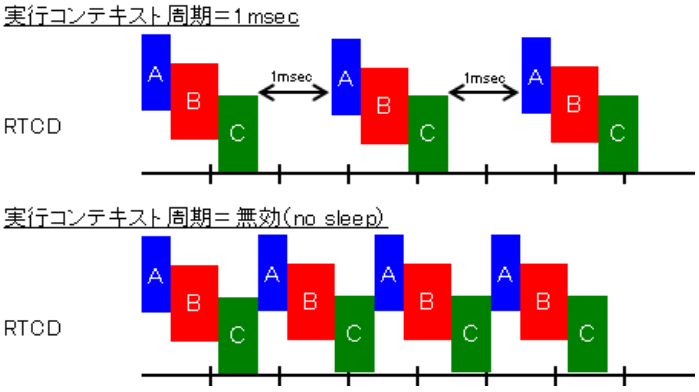
  複数の RTC を用いて一つの機能を構成する場合、各種制御を効率化・統一化するため、RTCD を用いることもできる。
  RTCD を用いたフィードバック制御の周期については 5msec が限度であり、要求性能が 5msec 以下の場合は
  モジュールの単一 RTC 化を推奨する。

  もし使用するのであれば、現状、ExecutionContext の実装上、「コアロジックの実処理時間」＋「設定した実行コンテキスト周期分の休止時間」
  が実行周期となっており、下図のように利用者が意図した周期より間延びするという問題がある。



この問題は ExecutionContext を共有する RTCD においては、さらに顕著に間延びする問題が懸念されるため、RTCD を使用する場合は実行コンテキストの

周期の設定を無効（no sleep）または非常に短い周期（1000000Hz 等）とすることを推奨する。



＜デフォルト＞

```
exec_cxt.periodic.rate: 1000        # 1000(Hz)
```

＜変更後＞

```
exec_cxt.periodic.rate: 0           #  無効(no sleep)
```

または

```
exec_cxt.periodic.rate: 1000000 # 1000000(Hz)
```

本対応は実行周期を早める副作用として CPU の占有率が高まりシステムへの負荷が大きくなることがが懸念されるため、モジュールの配置等も含め動作環境における状態により適宜調整を行う。

なお本内容は以下の調査結果に基づくものとする。

- 性能評価
  調査には「アームの分解運動速度制御 RTC 群」のシミュレータ動作環境を使用し、rtcd を用いたマルチスレッドでの RTC 動作と

  通常のマルチプロセスでの RTC 動作時におけるフィードバック周期の比較計測を行った。

  調査実施環境は以下の通りとする。

OS	Ubuntu10.04LTS(kernel 2.6.31-11-rt)
CPU	AMD Phenom II X4 905e 2.5GHz
メモリ	Memory：PC2-6400 4GB
使用 RTC	アームの分解運動速度制御 RTC 群
ExecutionContext の実行周期	1000/1000000

  結果は、デフォルトの実行周期で rtcd を動作させた場合、著しく間延びが発生し、rtcd を使用しない場合の約 2 倍の周期がかかることが判明した。また実行周期を非常に短く設定し、コアロジックの実処理後の休止時間をなくした場合は、よい結果が得られることが出来た。

実行コンテキストの周期	フィードバック周期(非 rtcd)	フィードバック周期(rtcd)
1000Hz	4.94msec	8.82msec
1000000Hz	5.69msec	4.76msec

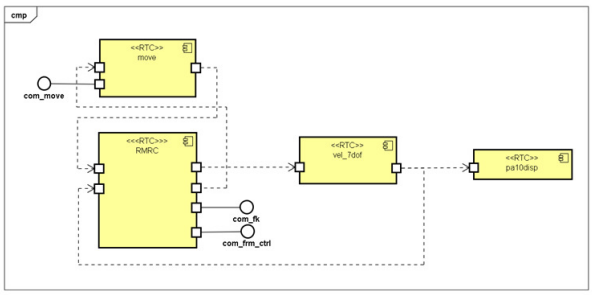
3.3. 単一 RTC 化

- 適用指針
  ハードウェアデバイス制御等のハードリアルタイム性を要求される条件下において、要求する制御周期が 5msec 以下となる場合は、複数の RTC を組み合わせて行うことは不可能であり、コアロジックを切り出して一つの RTC へまとめること

- 性能評価
  調査には「アームの分解運動速度制御 RTC 群」のシミュレータ動作環境を使用し、複数の RTC を用いた場合と単一の RTC にまとめた場合におけるフィードバック周期の比較計測を行った。

  調査実施環境は以下の通りとする。

OS	Ubuntu10.04LTS(kernel 2.6.31-11-rt)
CPU	Intel Core i5-520M vPro
メモリ	2 GB DDR3 SDRAM
使用 RTC	アームの分解運動速度制御 RTC 群
ExecutionContext の実行周期	1000



結果は、単一の RTC にまとめた場合は早く、RTC の実行周期 1 回分に相当する時間まで、短縮することが出来た。

実行コンテキストの周期	フィードバック周期(単一 RTC)	フィードバック周期(複数 RTC)
1000Hz	1.21msec	6.96msec

4. 環境の最適化ガイドライン

本項では全てのモジュールを対象に、より効率的に動作するための、動作環境の最適化ガイドラインについて説明する。

4.1. RT-Preempt の適用（Ubuntu）

● 適用指針

リアルタイム性能が要求されるデバイス制御モジュールを扱う端末については、
ハードリアルタイム処理を可能とするための RT-Preempt パッチを適用することを推奨する。

● 導入手順

インストールするには以下のコマンドを実行する。

```
$ sudo apt get install linux rt
```

インストール後、再起動を行うと、起動時に GRUB ブートローダーでリアルタイムカーネル(rt)が選択可能となる。

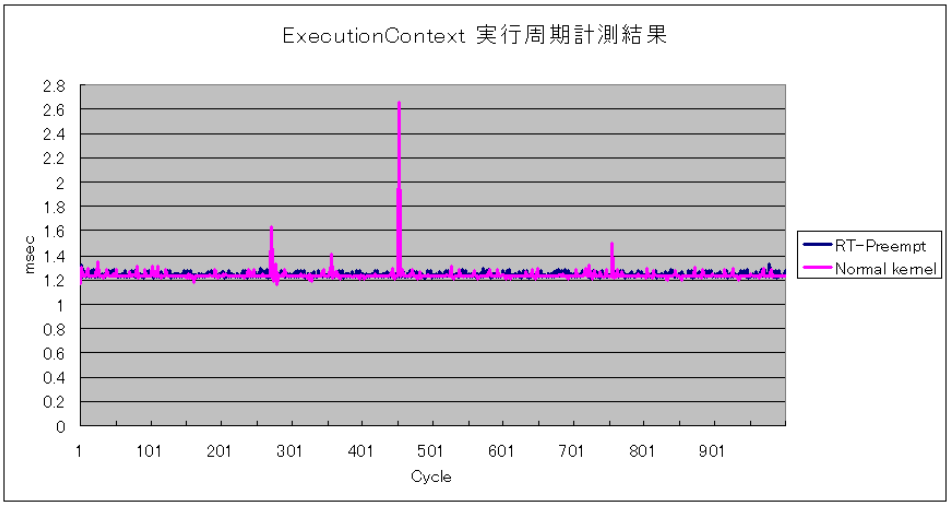
● 性能評価

調査には onExecute()内に ExecutionContext の実行周期の計測処理のみ実装した RTC を使用し、リアルタイムカーネルとノーマルカーネルの比較計測を行った。

調査実施環境は以下の通りとする。

OS	Ubuntu10.04LTS(kernel 2.6.31-11-rt)
CPU	AMD Phenom II X4 905e 2.5GHz
メモリ	Memory：PC2-6400 4GB
使用 RTC	LogTest
ExecutionContext の実行周期	1000

結果は、実行周期の平均はノーマルカーネル=1.238(msec)、リアルタイムカーネル=1.240(msec)と数値に変化は見られないが、ノーマルカーネルと比較して、リアルタイムカーネルはバラつきが少なく、安定した実行周期の計測結果が得られた。

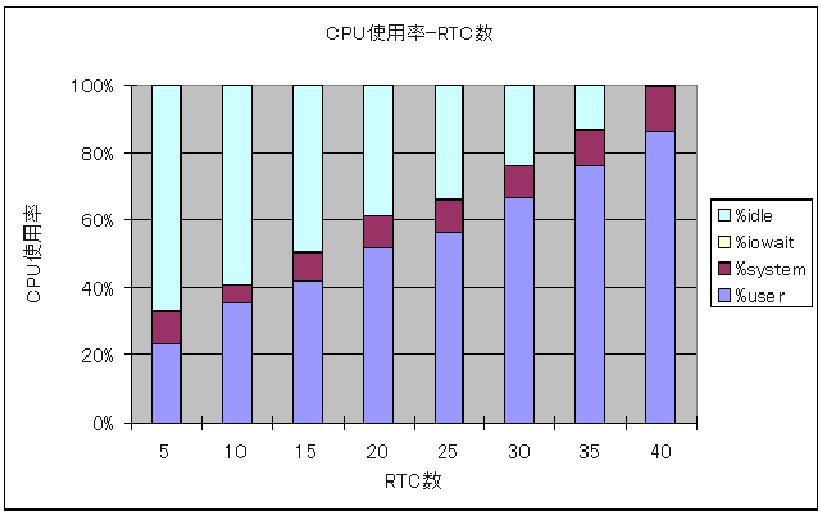


4.2. システムリソースの確保

- 適用指針
  同一 PC 上で、複数の RTC を起動させる場合に適用を推奨するものとし、その限度基準を 15 個とする。
  なお本基準は以下の調査結果に基づくものとし、動作環境における状態により適宜調整を行う。

- 性能評価
  調査には onExecute()内に ExecutionContext の実行周期の計測処理のみ実装した RTC を使用し、一定時間 sysstat による CPU 使用率及びメモリの監視を行った。
  調査実施環境は 1 と同様の環境とする。

  結果は RTC 数=40 個程度で CPU 使用率が飽和状態となり、メモリには特に大きな変動が見られなかった。



  以上の結果から、動作環境や RTC の内部実装によって変動することを考慮した上で、多くても 10～15 個が妥当と判断する。

5. 特記事項

本書をご利用される場合には、 以下の記載事項・条件にご同意いただいたものとします。

- 本書は独立行政法人 新エネルギー・産業技術総合開発機構の「次世代ロボット知能化技術開発プロジェクト」内実施者向けに評価を目的として提供するものであり、商用利用など他の目的で使用することを禁じます。
- 本書に情報を掲載する際には万全を期していますが、それらの情報の正確性またはお客様にとっての有用性等については一切保証いたしません。
- 利用者が本書を利用することにより生じたいかなる損害についても一切責任を負いません。
- 本書の変更、削除等は、原則として利用者への予告なしに行います。また、止むを得ない事由により公開を中断あるいは中止させていただくことがあります。
- 本書の情報の変更、削除、公開の中断、中止により、利用者に生じたいかなる損害についても一切責任を負いません。

【連絡先】

RTC 再利用技術研究センター
〒101-0021　東京都千代田区外神田 1-18-13　秋葉原ダイビル 1303 号室
Tel/Fax：03-3256-6353　E-Mail：contact@rtc-center.jp

以上