INtime アプリケーションプログラミングガイド

http://www.mnc.co.jp

TEL: +81(0)299-90-1733

FAX: +81(0)299-92-8557

REV.1 2019/11/26 RELEASE

目次

1 概要 4
2 絶対に取り入れるべき プログラミングテクニック 10 選 4
2.1 データ型メールボックスでの上限管理を行う 4
2.2 メッセージポンプ方式の利用 4
2.3 メモリ確保した際には、生成結果を確認する。 5
2.4 メモリ確保した際には領域をクリアしておく 5
2.5 オブジェクトハンドルをルックアップした時はハンドルを確認する 5
2.6 小さいサイズのメモリ確保は malloc を使う 6
2.7 Spin 例外を気にしすぎない 6
2.8 シグナル待機 Timeout による周期の生成 6
2.9 オブジェクトのカタログは積極的に行う 7
2.10 アプリケーション稼働情報のカタログ 7
3 絶対に避けるべき 「べからず」処理 10 選 8
3.1 「ポーリングによるビジーループ」はするべからず 8
3.2 「高頻度な NTX 通信要求」はするべからず 8
3.3 「オブジェクト生成削除の繰り返し」はするべからず 8
3.4 「過度なスレッド分割」はするべからず 8
3.5 「printf デバッグ」はするべからず 8
3.6 「他スレッドからのスレッド削除」はするべからず 9
3.7 「オブジェクトの不用意なルックアップ処理」はするべからず 9
3.8 「安易な排他処理」はするべからず 9
3.9 「ファイルアクセス処理」はするべからず 9
3.10 「MAP/UNMAP の繰り返し」はするべからず 9

※本ドキュメントの内容は予告なく変更される可能性があります。
また、本ドキュメントの無断転載・使用を固く禁じます。

本書で使用するマークについて

	ノート: 操作方法や手順等の補足情報や注釈を説明しています。
	情報: 製品を利用する上で有効な豆知識となる説明をしています。
	警告: 製品仕様上注意が必要な事象について説明しています。

Windows、Visual Studio は、 米国 Microsoft Corporation の米国およびその他の国における商標または登録商標です。
INtime は、米国 TenAsys Corporation の登録商標です。
TenAsys®, INtime®, eVM® and iRMX® are registered trademarks in USA of the TenAsys Corporation.
その他、本書に記載されている会社名、商品名は、各社の商標または登録商標です。
本書の内容を無断で転載することは禁止されています。
本書の内容に関しては、予告なしに変更することがあります。あらかじめご了承ください。

※本ドキュメントの内容は予告なく変更される可能性があります。
また、本ドキュメントの無断転載・使用を固く禁じます。

1　概要

INtime プログラミングを行うにあたり、行っていただきたいプログラミングテクニックや、逆に実装は避けてほしい処理（べからず処理）をまとめました。

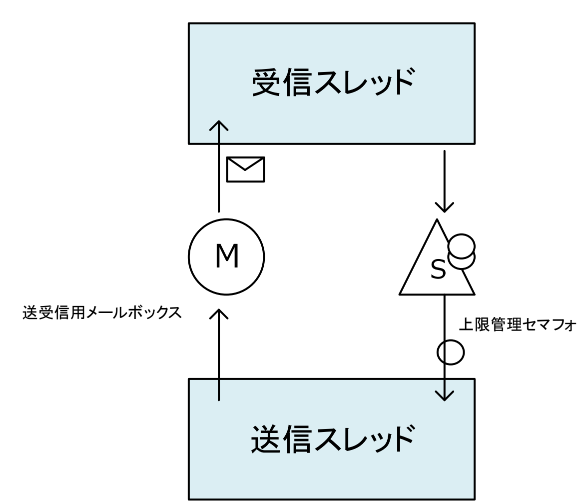
2　絶対に取り入れるべき プログラミングテクニック 10 選

2.1　データ型メールボックスでの上限管理を行う

データ型メールボックスのメッセージキューには上限がありません。そのためシステムメモリが枯渇するまでメッセージの送信処理（`SendRtData` のコール）ができてしまいます。

キューイングメッセージの上限数を意識することが大切です。キューイング数に制限を設けるためには、セマフォオブジェクトと組合せます。

送信側は、データ型メールボックスへメッセージを送信する前にセマフォよりユニットを受信し、受信ができたときのみメッセージを送信します。メッセージ受信側では受信後セマフォにユニットを解放します。こうすることで、セマフォ内に予め格納されているユニット数が、メッセージキューイングの上限数となります。

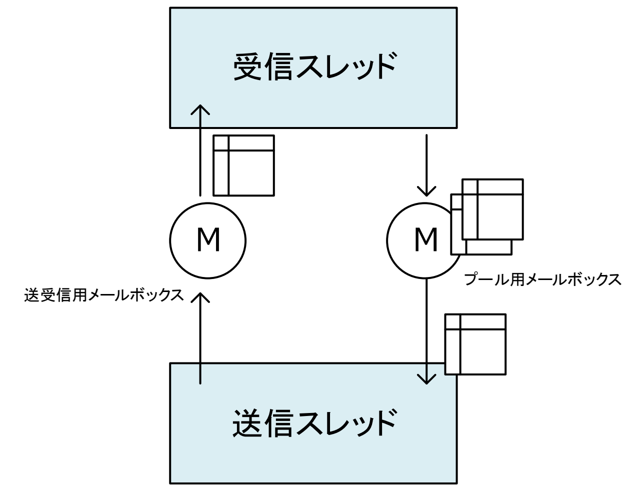


2.2　メッセージポンプ方式の利用

オブジェクト型メールボックス 2 つとメモリハンドルを組み合わせた方法で、データメールボックスの欠点（データサイズ上限 128Byte、キューイングメッセージの上限がない）を解消できる手法です。

まず、送受信メッセージ用にメモリオブジェクトを生成します。（生成数がメッセージキューイングの上限数になります）次にオブジェクト型メールボックスを、メッセージプール用と、メッセージ送受信用の 2 つ用意します。初期化処理として、メモリオブジェクトをプール用メールボックスにキューイングしておきます。

メッセージの送受信を行う際は、送信側はプール用メールボックスからメモリハンドルを受信し、送信データを記述後、送受信用メールボックスへ送信します。受信型はメッセージを受信後、受け取ったメモリハンドルを



※本ドキュメントの内容は予告なく変更される可能性があります。

プール用メールボックスへ返却します。この仕組みではメッセージデータサイズは自由に設定することができ、かつメッセージキューイングの上限数も管理できます。

2.3　メモリ確保した際には、生成結果を確認する。

AllcateRtMemory や malloc でメモリを動的に確保する際、得られるアドレスは必ず確認します。メモリ使用状況によっては、確保に失敗する場合があるためです。

```
void* p = AllocateRtMemory(8192);
if (p == NULL) {
        printf("Error 0x%0x \n", GetLastRtError());

}
```

戻り値が NULL であると、メモリ確保失敗は Page Fault として現れることとなり原因特定まで時間がかかってしまいます。しっかり NULL チェックを行い、メモリ確保失敗ケースをハンドリングします。またエラー発生時には GetLastRtError()にてエラーコードを確認することで詳しい原因が特定できます。

2.4　メモリ確保した際には領域をクリアしておく

メモリ確保に成功後、初期値にクリアします。

```
void* p = AllocateRtMemory(MEM_SIZE);
if (p == NULL) {
        printf("Error 0x%0x \n", GetLastRtError());
}
memset(p, 0x00, MEM_SIZE);
```

クリアしない場合、確保された領域の値は不定です。0 以外の場合もあり得ます。過去の事例では、問題なく動作していたシステムであるのにもかかわらず、動作環境を変えるとアプリケーションが付議愛を起こしたケースがありました。調査をしたところクリア処理を怠っていたため、メモリ確保処理で与えられた領域の値が変化し、問題となっていました。

2.5　オブジェクトハンドルをルックアップした時はハンドルを確認する

ルックアップ処理で（LookupRtHandle）得られたハンドル値は GetRtHandleType で確認します。ルックアップに成功しハンドル値が得られても、オブジェクトが削除されている場合があります。

```
RTHANDLE obj  = LookupRtHandle(h, "RtCTP", 1000);
short    type = GetRtHandleType(obj);
if (type == INVALID_TYPE){
        printf("Error 0x%0x \n", GetLastRtE
}
```

```
RefDelTsk        19c0
RslDelTsk        19a0
RslSemaphore     1980
RtCTP            1ce8 (invalid)
SYNC_THREAD      1b80
SpdrFlt          3630
```


※本ドキュメントの内容は予告なく変更される可能性があります。

この場合、得られたハンドル値を引数に GetRtHandleType をコールすると、INVALID_TYPE が得られます。

2.6 小さいサイズのメモリ確保は malloc を使う

動的なメモリ確保をする場合、または一時的に使用する処理バッファなど、比較的小さいサイズを確保するときには malloc を使用します。

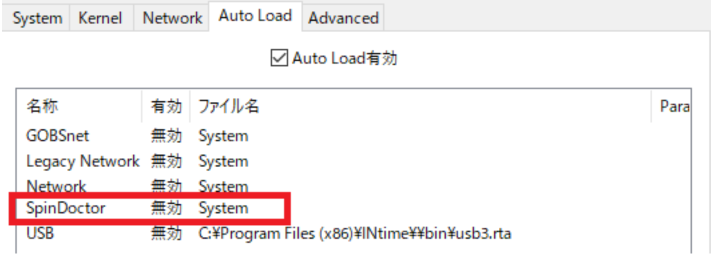
AllocateRtMemory でもメモリ確保はできますが、この場合 1 ページ（4096Byte）単位でのメモリ確保になります。そのため数十 Bytes 程度のメモリを一時的に使用するのであれば、AllocateRtMemory よりも malloc を使用する方が効率的です。

2.7 Spin 例外を気にしすぎない

プログラミング作成中に INtimeException 19（Spin 例外）が発生することがあります。これは 1 つのスレッドが一定期間 CPU を占有し続けたと判断されたときに発生するものです。このとき処理を見直す必要はありますが、意図した動作であれば問題ありません。

Spin 例外は主に開発中に使用するもので、プログラミング中のコーディングミスなどによってスレッドが動作を続けてしまうことを防ぐために使います。

意図した動作を行っているのにも関わらず、Spin 例外と判断される場合、検知機能（SpinDoctor）を無効にします。INtime Configuration から無効にできます。



2.8 シグナル待機 Timeout による周期の生成

ポーリング処理を行うにあたって、その周期の生成には、RtSleep()や knRtSleep()をよく使いますが、セマフォやメールボックスの受信待機によっても周期を作ることもできます。

こうすることで任意のタイミングでポーリング処理を終了することができます。

※本ドキュメントの内容は予告なく変更される可能性があります。

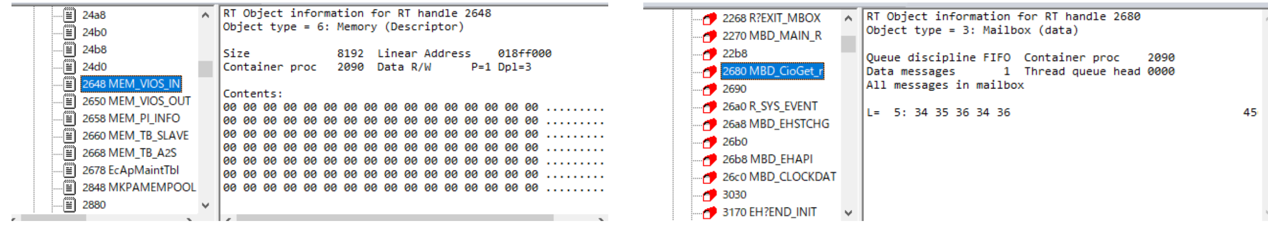
以下は、100msec 周期で外部機器からデータを取り込む（inbyte）処理ですが、StopEvent セマフォでイベント受信（ユニット受信）すると取り込み処理を中断します。ポイントはタイムアウトエラー（E_TIME）発生時に正常処理を行うことです。

```
while (TRUE)
{
        result = WaitForRtSemaphore(eveStop, 1, 100);
        if (result == WAIT_FAILED)
        {
                if (E_TIME == GetLastRtError())
                {
                        // タイムアウトの場合に正常処理を行う
                        data = inbyte(IN_PORT);
                }
                else
                {
                        // エラー処理
                }
        }
        else
        {
                // 取り込み処理中断
        }
}
```

2.9　オブジェクトのカタログは積極的に行う

作成したオブジェクトをカタログすることで、INtime Explorer から状況を確認しやすくなります。

INtime Explorer にはオブジェクトディレクトリが表示されますので、カタログしておけば、生成状態が確認できるほか、そのオブジェクトの動作状態の確認もできます。



2.10　アプリケーション稼働情報のカタログ

アプリケーション稼働状態を格納しているメモリをカタログすることで、動作中の稼働状態をモニタリングできます。INtime Explorer では、メモリオブジェクトをダンプする機能があるので、これを使ってのモニタリングも可能ですし、Windows アプリケーションを作成すれば、専用のデバッグツールを用意することも可能になります。

※本ドキュメントの内容は予告なく変更される可能性があります。

3　絶対に避けるべき 「べからず」処理 10 選

3.1　「ポーリングによるビジーループ」はするべからず

アイドル時間を持たないスレッド処理は行わないでください。この場合、そのスレッドより低いプライオリティを持つスレッドは動作できなくなります。

`RtSleep()`、`knRtSleep()`による息継ぎ時間を設けることで、低プライオリティスレッドが動作できるようになります。(低プライオリティスレッドがいない場合は問題ありません)

3.2　「高頻度な NTX 通信要求」はするべからず

Windows-INtime アプリケーション間での通信（NTX 通信）を行う際、INtime 側から高頻度に Windows 側へ処理要求を行うことはしないでください。Windows 側への高頻度入出力要求は、Windows 側の負担を累積的に増大させ、結果的に INtime 側処理が間に合わなくなります。これは機器の制御周期の乱れなどに繋がります。

高頻度な要求は行わないようにするとともに、INtime 側の処理で、NTX 通信要求処理と、リアルタイム性能が必要な処理（機器制御処理など）とをスレッド分けすることで、制御周期を保つことができます。

3.3　「オブジェクト生成削除の繰り返し」はするべからず

生成/削除処理の繰り返しは、処理時間オーバーヘッドになるだけでなく、メモリ不足等の理由から、生成処理が失敗する可能性もあるため、生成削除の繰り返しはしないでください。

オブジェクトの生成処理については、アプリケーション起動時に、実行に必要となるオブジェクトをすべて生成することで、オーバーヘッドも無くなり、また、仮にメモリ不足等で生成に失敗しても、稼働中に比べ状況を把握しやすくなります。

3.4　「過度なスレッド分割」はするべからず

適切なスレッド分割は必要ですが、過度に処理をスレッドに分割すると、逆に可視性やメンテナンス性を損なうことになります。またタスクスイッチのオーバーヘッドも無駄になります。

例えば単純なイベント駆動型のスレッドの場合、関数分けで済ませたほうが有効な場合もあります。

3.5　「printf デバッグ」はするべからず

INtime アプリケーションでも `printf` を使用できますが、内部的には、Windows 側に常駐するサービスとの同期的なやり取りが行われます。そのため、`printf` をコールするスレッドはリアルタイム性が損なわれます。

`printf` により動作状況を確認する必要がある場合、コンソール出力専用の低プライオリティスレッドを設け、各スレッドからの要求を受けてコンソール出力するようにします。

※本ドキュメントの内容は予告なく変更される可能性があります。

3.6　「他スレッドからのスレッド削除」はするべからず

DeleteRtThread をコールし、他スレッドを削除しないでください。この場合、メモリリークを引き起こす場合があります。スレッドは実行を終了するとき、リンクされている RSL の RslMain()RSL_THREAD_DETACH 終了ハンドラでリソース開放を処理します。しかし他スレッドから削除を指定された場合は終了ハンドラを処理する手段がなくなり、結果リソースリークが発生する場合があります。

スレッド削除を促すメッセージを介してスレッド自らが終了するように対策することを勧めます。

3.7　「オブジェクトの不用意なルックアップ処理」はするべからず

LookupRtHandle は他 API に比べ処理時間がかかるため、不用意にコールしないでください。

例えば、オブジェクトを生成したスレッドと同一プロセス内のスレッド処理であれば、ルックアップしなくても、生成処理で得られるハンドル値を使用することで、ルックアップ処理をする必要がなくなります。

3.8　「安易な排他処理」はするべからず

排他処理を行うと、他スレッドアクセス中では処理を待たされる場合があります。排他処理を行う場合、このような状況が起こることは当然なのですが、処理を待たされることで、問題が発生する場合あります。

排他処理を行う場合は、そもそも排他処理を使わないで済む方法をまず考える必要があります。

3.9　「ファイルアクセス処理」はするべからず

INtime アプリケーションでもファイルアクセス処理は可能ですが、内部的には、Windows 側に常駐するサービスとの同期的なやり取りが行われます。そのため、ファイルアクセス処理を行うスレッドはリアルタイム性が損なわれます。過去の事例では、ログ出力処理が各所に点在してしまい、結果リアルタイム性能が低下してしまうケースもありました。

INtime アプリケーションでファイルアクセスを行う場合は、専用の低プライオリティスレッドを設け、各スレッドからの要求を受けて処理するようにします。

3.10　「MAP/UNMAP の繰り返し」はするべからず

メモリアクセス処理において、メモリのマップ/アンマップ（MapRtSharedMemory/FreeRtMemory）を繰り返し行う実装にしないでください。メモリのマップ/アンマップ処理は比較的コストが高い処理であり、また NTX 処理でのアンマップ（ntxUnmapRtSharedMemory）は引数に渡すメモリハンドルがマップされたメモリ領域を全て解放してしまいます。このケースでは、複数スレッドから同じメモリオブジェクトにアクセスする処理があった場合、一方のアンマップ処理によって、他方のアクセス処理で Page Fault が発生してしまいます。

※本ドキュメントの内容は予告なく変更される可能性があります。

メモリアクセス処理では、一度マップ処理を行ったら、アンマップは行わず、以後はマップ済みのメモリアドレスへのアクセス処理を行うようにします。

※本ドキュメントの内容は予告なく変更される可能性があります。