継続と決定的スケジューリングを用いたリアルタイム・システム

Real-time System using Continuation and Deterministic Scheduling

河野 真治　　揚 挺
Shinji Kono E-Mail: kono@ie.u-ryukyu.ac.jp You Tei

Information Engineering, University of the Ryukyus,
PRESTO, Japan Science and Technology Corporation

Abstract

新しいリアルタイム・システムの構築法として、継続を持つ言語 CbC によるシステム構築手法について考察する。従来の手法での割り込みのオーバヘッドの原因を考察し、従来の RTOS + アプリケーションという枠組、OS + トラップという方式ではなく、割り込みを状態遷移ととしてとらえ、システムの応答性を上げることを考える。本手法で重要となる CbC で割り込みを実現する状態遷移記述について考察する。

1　従来のリアルタイム･システムを越えて

近年のリアルタイム・システムへの要求は、従来のものと、大きく異なってきている。ネットワークや CPU の速度の向上とともに、リアルタイムの動画再生や、ユーザにリアルタイムで応答する 3D グラフィックスハードウェアとの連係がなければ、これらの要求に答えることは出来なくなりつつある。

本論文では、継続中心とする従来とは異なるプログラミング単位を導入することにより、

ハードウェアとソフトウェア

仕様記述と実装記述

などの分離を乗り越えて、新しいリアルタイム・プログラミングの手法を提案する。

これまでのリアルタイム・システムは、RTOS (リアルタイム・カーネル) を中心として、タスクを定義し、システムコールにより、CPU 資源などをカーネルが資源を管理する手法をとっている。RTOS の中心は、スケジューラであり、メモリや CPU の資源がもっとも重要であった時代の設計となっている。

リアルタイム・システムで重要な要素は外部のイベントへの対応である。これらは、

定期的に外部デバイスを見に行くポーリング

外部からの信号により処理を強制的に変える割り込み

の二種類がある。これらの外部デバイスは、RTOS の中で標準的な API によりアクセスする必要があり、特別な資源として RTOS 固有なものとして定義されることが多い。複数のプロセッサを持つシステムでは、同期を行なうセマフォなども RTOS のデバイスとして定義される。

一方で、ハードウェアは、Verilog や VHDL のようなハードウェア言語で別に記述され、RTOS からの資源管理とは別に記述されている。ハードウェアの集積度は劇的に上がって来ており、資源の有効利用よりは、重要なのは消費電力や速度に変わって来ている。ハードウェアレベルでの複数資源の競合は、通常は、バスの占有処理として表れる。これらは、アービタと呼ばれる専用の回路で実現されるのが普通である。

1.1 従来のリアルタイム・システムの問題点

RTOS とハードウェアの間は、ハードウェア側が定義する API にしたがって行なわれるのが普通であり、これが、ハードウェア側の処理の複雑さの原因になるとともに、ソフトウェアを含むシステムのデバッグなどを難しくしている。これが実機のない状態でのシステム検証や、ICE (In-Circuit Emulator) 抜きでのデバッグを不可能にしている。

リアルタイム・システムを構成するのは、さまざまな実際の処理を行なうタスクである。タスクは、ハードウェア・ソフトウェア上で、逐次あるいは同時に処理される。複数のタスクが特定の資源を共有する場合は、同期をとることが必須である。しかし、ハードウェア上の同期方式と、ソフトウェア上の同期方式は、別な方法で実現されている。これがシステム記述を複雑にしている。

RTOS では、割り込みとタスク切替えによるオーバヘッドを減らすことが重要である。しかし、RTOS の介入により、むしろ、そのオーバヘッドは増える傾向にある。

組み込みシステムでは、メモリは限られた資源である。しかし、構造化プログラミングやオブジェクト指向プログラミングは、スタックに依存している。原理的に使用量を予測しづらいスタックの使用は組み込みシステムに向かない。したがって、構造化プログラムあるいは、オブジェクト指向プログラムを全面的に採用することができない。

2 継続を中心とした C 言語 CbC

CbC (Continuation based C) [2] は、本研究室で開発した、

ループとサブルーチンコールを持たない

C 言語である。通常のサブルーチンの代りに、すべては、code と呼ばれる単位を用いてプログラミングされる。以下の例では、state1 は、入力状態 input は、interface と呼ばれる構造体型 state で記述されている。state1 は、パラメタ付き goto 文で分岐されるまでの処理を記述する。

```
code state1(state input,state arg) {
    if (input_cond1(input)) {
        goto state2(arg);
    } else if (input_cond2(input)) {
        goto state2(arg);
    } else if (input_cond3(input)) {
        goto state3(arg);
    }
}
```

Stack を明示的に取り扱うことにより、通常の C 言語を、CbC へコンパイルすることも可能である。

code は goto 文の行き先であり、行き先を間接的に指定したものは、継続と呼ばれる。すなわち、CbC は、継続のみを持つ C 言語だということができる。

この言語は、状態遷移を記述するための言語であり、C で記述したソフトウェア部分と、VHDL などで記述したハードウェア部分の両方を同時に記述することが一つの目標となっている。また、状態遷移を記述することにより、状態遷移記述による仕様記述も、この言語を用いて記述することができる。

構造化プログラミングあるいは、オブジェクト指向プログラミングで記述されたものを、CbC に変換することにより、リアルタイム応答性や全体の資源の使用量などをはっきり予測することが出来るようにすることが本研究の目的である。本論文では、特に状態遷移を使って RTOS のオーバヘッドなしに割り込みを処理する方法を考察する。(fig.1)

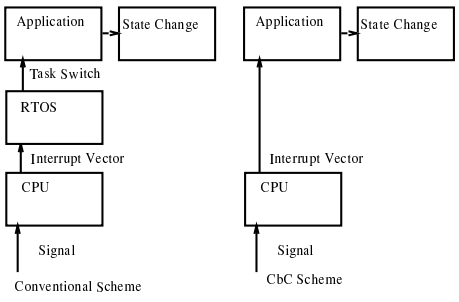

Fig1 New Real-time Scheme

3 CbC のタスクとスケジューラ

CbC では、code の実行のシーケンスが自然にタスクを構成する。code は、ループを含まないので、実際の実行により実行時間を測定することが可能である。タスクの内部状態は、大域変

数と interfaceにより決定される。つまり、従来のスタック型言語と異なり、呼び出された code がすべての状態を明示的に持っている。

したがって、CbC のタスク切替えは、決まった interface を持つ一つの code であるスケジューラを呼び出すことによって実現できる。スケジューラでは、interface に格納された状態を、適当な大域変数タスクキューに格納し、別なタスクに対応した interface 状態を取り出して、適切な code へ goto すれば良い。

従来のスタック型のプログラミング言語では、このような操作は、本質的に記述することができない。それは、スタックに格納された環境へ直接アクセスすることができないからである。したがって setjump などように特別な API を用意する必要がある。

3.1 リアルタイム・システムでの割り込み処理

割り込みは、リアルタイムシステムでは、重要な要素である。割り込みは、通常は、CPU の特別な機能として実現されている。割り込みコントローラなどの補助的な機構を用いることもあるが、基本的には

CPU には、割り込みを発生させる特別な信号線が用意されており、CPU の回路がその信号線を常にチェックしている。信号が来ると、CPU は、現在実行中の命令パイプラインを中断し、内部状態をセーブする。次に (あるいは、それと並行しながら) 信号線に対応した割り込みベクタを読み込む。ベクタにしたがって、新しい処理を進める。これが CPU にとっての割り込みである。割り込みベクタは、現在では、割り込みコントローラから与えられることもあり、仮想記憶などの処理もからんで複雑である。割り込みは、CPU にとっては特別な状態であり、これを解除する仕組みが必要である。通常は、割り込み状態が解除されるまで新規の割り込みは、受け付けることはあっても、処理されない。これを処理することにすると、CPU のセーブした状態や、割り込み要求先の状態が失われる可能性がある。

割り込み信号検知
現実行状態の格納
割り込みベクタ決定
割り込み処理ルーチンの呼び出し
割り込み状態からの復帰

RTOS から見た割り込み処理は、その後に始まる。通常、RTOS は、汎用の割り込み処理ルーチンを持っており、それが呼び出される。しかし、状況は、さらに複雑である。

[CPU 型] CPU からの割り込みを直接処理する RTOS に属さないルーチン

[Call 型] CPU の割り込み状態は解除するが、タスク切替えを伴わない RTOS の割り込み処理ルーチン

[Task 型] タスク切替えを伴う処理

の三種類の処理を持っている場合が多い。(これらの名前は勝手なものである) それぞれ、特徴があり、

[CPU 型] 高速応答が可能であるが、RTOS の資源を使うことは一切出来ない。他の割り込みを受け付けることが出来ない。

[Call 型] ある程度の RTOS の資源 (スタックやレジスタ) を使うことができるが、長時間の処理を行なったり、RTOS の他のデバイスにアクセスするようなことはできない。CPU 型以外の割り込み処理を行なうことができない。

[Task 型] 応答時間がもっともかかるが、RTOS のすべての資源にアクセスできる。

というようになっている。デバイスドライバ中などの処理は、アクセスする資源は、デバイスのみに限るために、Call 型で十分なことが多い。Unix などの高度な OS では、CPU 型の割り込みを使用することはほとんどない。RTOS は、実際のアプリケーションの内容には、関わらない。生じた割り込みをアプリケーションに反映することは、タスク切替えが伴うので、必ず Task 型の割り込み処理が必要である。

つまり、実際の割り込み処理は、

CPU 型 → Call 型 → Task 型

という手順を踏んでアプリケーションに反映されることになる。例えば、Unix では、割り込みの結果は、アプリケーション側では、以下のように反映される。

read/write などのシステム・コールからの復帰

特に select システム・コールからの復帰

シグナルハンドラの呼び出し

シグナルハンドラを経由しないプロセスの状態の変更

これらは、すべて、なんらかのイベント待ちループを要求するのが普通である。つまり、外部と相互作用するアプリケーションは、すべて、

```
for(;;) {
```

```
        fds = select(...);
        do_service();
    }
```

のような形を取ることになる。Unix の daemon は、このようなものの典型である。これをここでは、メインループ手法と呼ぶ。

アプリケーション間の通信も、このような形で実現されていることが多く、実際の外部イベントがアプリケーションの挙動に反映されるまでには、

割り込み → メインループ → カーネル要求

の繰り返しを多数えることになる。

これらの繰り返しが、特に Unix での、割り込み応答の速度低下を引き起こしている。実際には、タスク切替えには、仮想記憶関連の処理も入るために、さらにオーバヘッドが増えてしまう。これが、仮想記憶サポートがリアルタイムシステムでは禁止される原因でもある。

しかし、今は、仮想記憶を持つような巨大なシステムにこそリアルタイム応答性が要求されているのであり、ここで述べて来たような割り込み応答のオーバヘッドを原理的に含まないような外部イベントの処理の仕組みが必要である。

大きなシステム、Unix や Windows の応答性が CPU 速度の向上に見合う程、高速になっていないは、体感上も事実であり、なんらかの原理的な解決が必要である。テレビゲームや、ビデオ処理システムなどの応答性は決して悪くないであり、これは、ハードウェアやソフトウェアの限界と言うよりは、現在のソフトウェアを含むアーキテクチャの限界であると考えられる。

4 ハードウェア上での割り込み

電子回路そのものには、割り込みと言う概念は存在しない。割り込みを検出する信号線は、特に特別なものではない。現在のハードウェアは、有限状態遷移で記述されているのが普通である。つまり、割り込みは、その状態遷移制御する信号線の一部でしかない。

しかし、ハードウェアが複雑になるにつれて、そのような単純なイメージでは、割り込みを正しく理解したことにはならない。現在のハードウェアは、単純な 1 レベルの状態遷移系ではなく、複数の状態遷移系の合成となっている。それぞれの回路は、低電力化などのために、電源が供給されていなかったり、遅いクロックで駆動されていたりする。特に、パイプライン上の処理をハードウェアで行なうことは、もはや普通であり、任意の時点で、任意の状態に移行することは、ハードウェアでも簡単なことではなくなっている。

パイプライン・アーキテクチャの割り込みは、外部から見えるべきハードウェアの状態と、内部での状態が一致していないということを考慮する必要がある。割り込みを受けた段階で、ハードウェアは、割り込み状態に移行するが、それは、ハードウェアの受け付けた入力履歴の特定の段階で状態を移行した形をとる必要がある。パイプライン実行や投機的実行では、内部の状態を、その特定の状態に収束させる必要がある。そのためには、パイプラインを構成する複数の状態遷移機械を同期させる必要があり、必然的に時間がかかることになる。

これを避ける方法の一つは、割り込みを受け付けるハードウェアを別に用意することである。例えば、割り込みを受け付けない CPU などを独立に用意することにより、割り込みのオーバヘッドを下げることができる。しかし、これは、割り込みの応答速度を上げることには結び付いていない。

5 コールバック

ソフトウェアから見た割り込みは、メインループ以外の方法としては、コールバックという手法で受け付けることができる。コールバックは、イベントハンドラに、あるタスク内にあるサブルーチンを引渡す。イベントハンドラは、そのルーチンを外部イベントにしたがって呼び出す。

この方法は、もともとイベント駆動型に書いてあるアプリケーションには有効である。例えば、GUI アプリケーションのボタン毎にコールバックを割り当てることにより、ボタン状態を大域変数に反映することは簡単である。大域変数は、メインループの振舞を変えるために使われるのが普通である。つまり、大域変数を変更する手法では、メインループ手法と応答性と言う点では差がない。

Unix の signal もコールバック・ルーチンの一種である。しかし、signal は、外部状態を直接渡

すことは出来ない。signal を受けたタスクは、別途、signal の原因を調べるために、なんらかのデバイスにアクセスする必要がある。その調べるルーチンは、メインループから呼ばれないとすると、必然的に、プログラム全体をマルチスレッドで構成する必要がある。そのためには、複数のスレッドがセマフォなどの同期を必要とし、それが、また、別なオーバヘッドになる。本来、余計な同期を避けるべく導入されたコールバックが、余計な同期を導入していることになる。(fig.2)

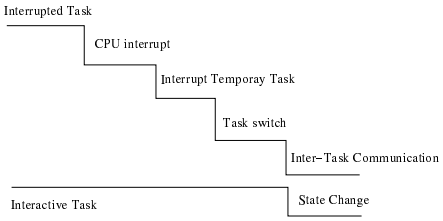

Fig2　Interrupt Overhead

6　割り込みと状態遷移

これまで、見て来た割り込みの手法は、同じことが繰り返し表れているのが分かる。それは、割り込みの本質は、状態遷移であり、最終的なタスクの状態を変えるという目的を果たすためには、さまざまな状態遷移を中断させる必要があるということである。

しかし、外部イベントによって動作を変えるアプリケーションの状態遷移がもっとも重要なものであり、ハードウェアレベルでの割り込みが、直接アプリケーションの状態を変えることが望ましいことは言うまでもない。

問題は、アプリケーションそのものが状態遷移を受け付けることを考慮してないことが問題なのだと思われる。CPU の持つパイプライン構成や、タスクを構成する資源自体が、外部の状態によって状態遷移することを中心に作られていないことが、割り込み処理のオーバヘッドそのものになっている。

アプリケーションがメインループで行なっていることは、基本的にはポーリングである。ポーリングとは、イベントが起きているかどうかを判断する機構を定期的に呼び出すことである。このポーリングが自動的にアプリケーションに組み込まれるようにすることが割り込みのオーバヘッド減らすことにつながると思われる。

ハードウェアでも、保存すべき状態がクロック単位あるいは、命令単位であることが、問題を難しくしている。割り込みが起きた時に、現在のパイプライン状態よりも大きな単位での状態に戻ることができれば、よりハードウェア構成が簡単になる可能性がある。

7　CbC での割り込み

サブルーチンを持たない CbC では、割り込みは、単純な状態遷移である。それは、スケジューラを用いて間接的に記述することも出来るし、goto 文の間に、別な code を実行するような形で、暗黙的に記述しても良い。また、code の中に、直接、割り込みを示す状態変数へのアクセスという形で明示的に記述することも出来る。(fig.3)

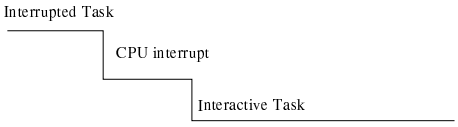

Fig3　Interrupt in CbC

したがって、CbC での割り込み処理は、状態遷移を CbC の中でどのように記述するかと言うことに帰着される。

割り込み信号は、CPU 型の割り込みにより大域変数の変更に帰着される。変更は、単純な値の変更あるいは、大域変数上に構成されたキューの変更となる。キューの変更の場合のコンシステンシ (整合的なキューの変更の保証) は、なんからの方法 (例えば読み出し時の割り込み禁止) で保証される必要がある。

7.1　CbC での状態遷移表現

CbC では、C 言語と同様な状態遷移の表現をとることができる。しかし、C 言語では、サブルーチンを使ったスタックを移動させる表現が使われるが、CbC の場合は、goto 文を使った直接的な表現が可能である。基本的には、直接表現、間接表現、スケジューラ表現の三種類が考えられる。

7.2 直接表現

直接表現は、code 中の条件文として状態遷移を記述する方法である。高速でコンパクトだが、コードの変更などはできない。

```
code state1(state input,state arg) {
    if (input_cond1(input)) {
        goto state2(arg);
    } else if (input_cond2(input)) {
        goto state2(arg);
    } else if (input_cond3(input)) {
        goto state3(arg);
    }
}
```

7.3 間接表現

間接表現は、テーブル中に goto 文の行き先を格納することによって状態遷移を記述する。テーブルの書き換えにより状態遷移を動的に変更することができる。しかし、状態遷移先は、すべて同じ interface を持つ必要がある。

```
code state1(state input,state arg) {
    goto (state1[input_cond(input)])(arg);
}
```

7.4 スケジューラ表現

スケジューラ表現は、より複雑な分岐を可能にするスケジューラを呼び出す手法である。C では、スタック環境にアクセスする API を導入することによって、この表現が可能になる。この手法でも、呼び出される側は、共通の interface を持つ必要がある。

```
code state1(process_queue q, state arg) {
    goto scheduler(q,arg);
}
```

7.5 メモリ空間切替えを含む場合

以上の 3 手法は、すべて同じメモリ空間上で実現するための手法である。より複雑なシステムでは、複数の code シーケンスが別々のメモリ空間上で動作している状況が要求される。(fig.4)

メモリ空間切替え自体は、tagged TLB などをサポートしている現在のシステムでは、それほど大きなコストではない。メモリ空間は、code の interface 中に暗黙に指定された属性と考えられる。code 自体も、タスクが異なれば、異なるメモリ空間上に配置される可能性がある。

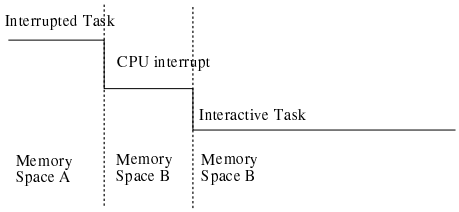

Fig4 Memory Space in CbC

割り込みによって、次にどのタスク (code シーケンス) が実行されるかは、決まっていない。したがって、従来のシステムとは異なり、複数のメモリ空間上に属する code と interface を goto 文で行き来するようなシステムとなる。従来のシステムでは、なんらかのシステムコールがなければ、メモリ空間の切替えは起こらなかった。しかし、CbC では、複数のメモリ空間を同時に扱う必要がある。

これまで、そのような複数のメモリ空間を同時に扱うことが出来なかったのは、一部のプログラムが、与えられたタスクの権限を越えて、他のメモリ空間にアクセスすることを防ぐためである。したがって、mmap などの共有メモリを用いることにより、許可されたメモリ空間を自分のメモリ空間に読み変えてアクセスすることは許されていた。

したがって、このような複数のメモリ空間での干渉を防ぐためには、「特定のメモリ領域しかアクセスできない」という性質を検証する必要がある。検証できない code の場合は、複数のメモリ空間が共存する形での実行を認めることは危険すぎる。

7.6 ポーリング

通常のアプリケーションのもっとも良くある状況は、idle ループであり、idle ループ上で複数のメモリ空間を共存させることには問題はない。なんらかのセキュアでないタスクが走っている状況では、なんらかの方法でセキュアな code を定期的に呼び出す必要がある。

リアルタイム応答は、現在では CPU の速度に比べて遅いのが普通である。例えば、CPU が GHz で動いている時に、ビデオの一水平線に要

求される応答速度は、MHz 以下である。したがって、このポーリングは、1/1000 程度のオーバヘッドで良い。一方で、タスク切替えを含む割り込み処理に要求される命令数は、本システムでは、CPU 型割り込みと 1 ジャンプ命令である。これは、タスク切替えに要求される数百命令に比べれば極めて小さい。これは、つまり、0.1 より、割り込みの応答性を、1000 倍にする技術である。(fig.5)

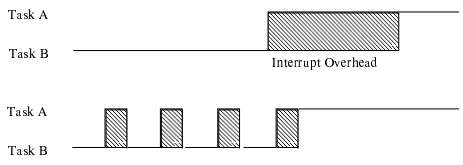

Fig5　Polling

すべてのアプリケーションが CbC で書かれている状況ならば、goto 文の途中にそのような code を挟み込むようにすれば良い。これは、一種のプログラム変換である。このようなプログラム変換を OS の中心として採用したものとしては、Synthesis OS [1] が有名である。本システムでは、システム全体を継続を中心とすることにより、システム全体を巨大な状態遷移系として取り扱うことになる。

8　CbC によるリアルタイム・システム設計

CbC を使ったリアルタイム・システム設計は、ポーリングを使うために、特定の時点で生じる可能性のあるイベントをすべて知っている必要がある。つまり、この意味で、このシステムは、動作が入力によって完全に決定されるシステムとなっている。

そのイベントの種類自体は非常に多いと考えられる。しかし、イベントの元は、アプリケーションの持つ入力デバイス数で抑えられる。一つのコンピュータに数百のデバイスが付くことはありえない。そのような場合は、単一 (あるいは少数の) のネットワークを接続し、そこから入力を得るのが普通である。つまり、多数の入力の処理は、木構造のような方法で少数の競合する入力に置き換えられる。

このような状況で、特定の時点で生じる可能性のあるイベントをすべて知っていることは、不可能ではないと思われる。

しかし、入力デバイスの追加、あるいは、ソフトウェアの切替えなどで、外部イベントの集合が変更されることはありえる。その様な場合には、間接法、あるいは、スケジューラ法などで、変更可能な状態遷移系を実現することで対応可能である。また、新しい状態遷移を表現した code を生成することにより、より高速な実現が可能となる。

8.1　決定的スケジューリングの問題点

本手法を使ったリアルタイム・システムは、従来のメインループ型のアプリケーションとは、まったく異なる。本質的にイベント駆動型のプログラムを行ない、あらゆる時点で、他のイベントが起こることを考慮したプログラミングを要求される。

出来上がった最終的なプログラムは、巨大な状態遷移となる。これは、本来、人間か書けるようなものではない。また、「止めて調べて再実行」のような手法ではデバッグすることができない。

状態数 (code) 自体は、それほど大きくないと期待される。本システムでの状態数は、アプリケーションの総ステートメント数程度となる。状態遷移数は、入力状態に比例し、状態数をかけたものとなるが、入力状態数は、入力デバイスの数程度となると考えられる。結果的に、code 自体がある程度の大きさを持つ場合は、従来の手法よりも多少コード数が大きくなる程度ですむと思われる。code が数命令程度のものばかりであると、従来の手法の数倍のコードを要求する場合もある。

一方で、より細かい単位での記述を行なうために、より細かい code 共有を可能にすることが期待される。その場合は、従来の手法よりも、小さなコードで十分である。

一方、入力デバイスが変更される、あるいは、アプリケーションが入れ替わるような場合は、code を実行時に生成する必要がある。これは、より多くの資源を要求することになる。

8.2 決定的スケジューラの大きさ

決定的スケジューリングは、スケジューラを使うことにより、非決定的スケジューリングを模倣することができる。本来、コンピュータ・システムは決定的なので、非決定的スケジューリングは、もともと模倣手法でしかなかった。

決定的スケジューリングは、決定性オートマトンと非決定性オートマトンと同じく、決定的な方が、より大きくなる傾向がある。例えば、a と b という二つのイベントがある場合、非決定的なスケジューラの場合は、状態遷移は 3 つで良い。

しかし、決定的なスケジューラの場合は、以下のような 4 つの状態を記述する必要がある。一般的に、入力イベントの指数乗の状態遷移を記述する必要がある。

これを解決するためには、二つの方法がある。一つは、変数を見る順序を固定して、見掛け上、線形の数の状態遷移に帰着させる方法である。この方法では、状態数は増えることになる。

もう一つは、重要でないイベントの組合せを無視する方法である。これは、ハードウェア合成での don't care の利用と同じことになる。

どちらの手法をとるにせよ、決定的スケジューラの大きさを抑えるためには、なんらかの最適化手法が必要となると思われる。通常、外部イベントは、極少数なので、この問題は、普通の場合は問題とはならない。

9 まとめ

本論文では、新しいリアルタイム・システムの構築法として、継続を持つ言語 CbC によるシステム構築手法について考察した。

従来の手法での割り込みのオーバヘッドの原因を考えることにより、オーバヘッドの中心は、状態遷移を考慮しないアーキテクチャであるあることを指摘した。

従来の RTOS + アプリケーションという枠組、OS + トラップという方式ではなく、割り込みを状態遷移ととしてとらえ、システムの応答性を上げることを目指している。

従来のものと大幅に異なるために、さまざまな問題があるが、ここでは、CbC で割り込みを実現する状態遷移記述について考察した。

References

[1] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. In *Computing Systems*, Vol. 1. The MIT Press, 1988.

[2] 河野 真治 . 継続を持つ C の下位言語によるシステム記述 . 日本ソフトウェア科学会第 17 回大会論文集, September 2000.