システムソフトウェアと
オペレーティング・システム 84－9
(2000. 5. 25)

PC 上における組み込みシステムのエミュレーション

高橋 雅彦[†] 高野陽介[‡]

[†]NEC 情報通信メディア研究本部 [‡]NEC インキュベーションセンター

要旨

組み込みシステムは、その機器に特化した OS を用いることによりオーバヘッドを小さくしてリアルタイム性を実現している。このような構造は、完成したシステムにとってはオーバヘッドのない動作が可能となるため便利であるが、アプリケーションの誤動作がカーネルに影響を及ぼすこともあるため、開発段階では困難をともなう。
汎用 OS を用いればこの問題は回避できるが、汎用 OS はこれまでオーバヘッドをともなっていたため、リアルタイム処理の実現性が低かった。本稿では、最近の PC の高速性を活用して組込システムをエミュレーションするシステムを汎用 OS 上に構築し、そのリアルタイム性を検証したところ、366MHz の PC での割り込み応答遅延は平均で 20μs 程度であるとの性能結果を得た。

An Emulation Method of Embedded Systems on PC

Masahiko TAKAHASHI[†] Yosuke TAKANO[‡]

[†]NEC Computer & Communication Media Research [‡]NEC Incubation Center

Abstract

It is difficult to develop applications with an embedded OS due to its structure. An embedded OS is specified to minimize overhead to realize realtime operations, and, then, may be harmed by applications because there is no protection between the OS and applications.
In the case of an OS on PC, by using the virtual address space and the non-privilege mode of a CPU, it accomplishes the safety from buggy applications. It, however, has contained overhead through itself when devices interrupt.
This paper describes a system that emulates an embedded system on PC with a recent faster CPU. We evaluate its realtime performance, and its interrupt latency is about 20μs on the average with 366MHz CPU.

1 はじめに

組み込み OS は、OS の機構を簡潔にして余計なオーバヘッドを取り除くことでリアルタイム性を確保している。例えば、割り込み応答性に優れた組み込み OS では、カーネルとアプリケーションの動作の制約には違いがない。つまり、アドレス空間は 1 つしかなく、常に CPU の特権モードで動作している。このような構造は、完成したシステムにとってはオーバヘッドなしに動作出来るため便利である。しかし、アプリケーションの誤動作によってカーネルを破壊することも出来るため、特に開発段階においては、組み込み OS 上で動作するアプリケーションを開発するのは容易ではない。

一方、汎用 OS では、CPU モードの切替えや仮想アドレス空間を利用して、カーネルとアプリケーションの区別やアプリケーション毎の独立性を実現している。つまり、組み込み OS がオーバヘッドの小さいリアルタイム性を重視しているのに対し、汎用 OS では安全性・独立性を重視して設計されているといえる。

しかし、近年の CPU の高速化により、汎用 OS 上でのリアルタイム処理が現実的になってきている。というのは、安全性を確保するために複雑な機構を持つ汎用 OS であっても、高速な CPU を利用すれば OS 内でのそれらの処理が短時間ですむようになり、相対的にオーバヘッドが小さくなるからである[1]。

本稿では、高速な CPU を活用してリアルタイム処理を PC 上で行うエミュレーション環境を構築した。PC 上でエミュレーションすることのメリットには、汎用 OS のユーザーレベルで実行されるため、仮想アドレス空間の保護を受けて開発がしやすくなることがあげられる。従来の汎用 OS の課題であったリアルタイム性は、高速な CPU を利用することで確保している。

今回我々はこれを Linux 上で実装し、Pentium II 366MHz を用いた PC 上でエミュレーションした際のリアルタイム性能を測定したところ、デバイスからの割り込み応答遅延は平均で 20μs 程度、デバイスへの I/O 命令発行の遅延が 1.7μs という性能結果を得た。

[1] ただし、高速な CPU はまだまだ高価であるので、組み込み機器の量産品に組み込むのは困難な場合が多い。本研究ではそうではない機器か組み込み機器の開発支援をターゲットとする。

2 組込み OS の割み込みモデル

組み込みシステムとは、機器に組み込まれるハードウェアとソフトウェアの全てを含んだ呼び方であるが、本稿では特にソフトウェア (組み込みソフト) に関して言及する。組み込みソフトには、組み込み OS、ハードウェア (デバイス) を制御するためのデバイスドライバ、及び、アプリケーションなどが含まれる。

組み込み OS の割り込みモデルを、広く使われている μITRON を例に説明する (図 1)。デバイスが割り込みをかけてきたとき、割り込みは最初 ISR(interrupt service routine) に伝わる。一般に、ISR は割り込み禁止の状態で実行されるため、割り込み処理が長くかかるハンドラは、必要最低限の処理を行う ISR と、割り込み許可状態で実行される DSR(deferred service routine) に分ける。ISR での処理が終わったあと、他の割り込みがペンディングされていなければ DSR に実行が移る。割り込み処理が終わると、アプリケーションの動作に戻る。一般に、デバイスが割り込みをかけてから ISR で受け取るまでの遅延を割り込み応答遅延といい、組み込み OS では割り込み応答遅延の最大値を保証している。

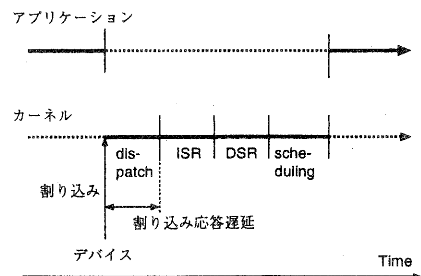

図 1: 組み込み OS の割り込みモデル

3 PC 上における組み込みシステムのエミュレーション

本研究では、PC 上において組み込みシステムをエミュレーションする環境を構築した。この章では、PC 上でエミュレーションする際の課題を述べた後、今回実装したプロトタイプのメカニズムを述べる。

3.1 エミュレーションでの課題

最近、PC のコストパフォーマンス性と利便性の向上、及び、GUI インターフェースの普及により、PC をミリオーダーが必要とされる制御用に使用しようという傾向がある。従来、このような用途に PC を使用するためには、制御ソフトをカーネルレベルで動作させる必要があった。というのは、ユーザーレベルのアプリケーションからでは、OS におけるオーバヘッドが大きく、制御に必要とされるミリ秒オーダーの制御が困難だったからである。例えば [1] では、カーネルでの割り込みから DSR までの遅延は 1ms 以下、また、デバイスドライバのスレッドまでの遅延でも 1.125ms 程度 (Pentium II 300MHz, Windows NT) と報告されている。

しかしながら、カーネルレベルで制御を行うソフトの開発は、特に開発段階においては困難がともなう。というのは、カーネルレベルで動作すると、誤動作の際にカーネルのデータにも影響を及ぼすことがあり、ときには再起動を必要とされるからである。

今回、我々は PC のユーザーレベルで組み込みシステムをエミュレートするシステムの試作及び性能評価を行った。この狙いは、最近の高速な PC を使用すればカーネルレベルではなくとも一定のリアルタイム性が実現出来ることを示すことである。ユーザーレベルでの動作であれば、アプリケーションがカーネルのデータを破壊することは出来ず、安全な開発が可能である。このユーザーレベルでエミュレーションする機構は、組み込みシステムのプロトタイプの作成やデバイスドライバの作成などの支援の他にも、PC 用のデバイスをユーザーレベルから制御するアプリケーションを開発する際にも利用出来る。

しかし、PC 上で組み込みシステムをエミュレーションする際には

(a) PC 上の OS のスケジューラの制限を受ける
(b) ユーザーレベルからは直接 I/O 命令が発行できない
(c) デバイスの割り込みをユーザーレベルで受け取る機構がない

といったことが問題になる。

(a) に関しては、具体的には

- 他のプロセスの実行を禁止することは出来ない
- 他のプロセスに比べて、最優先の実行を保証することは出来ない

ことなどである。このため、他プロセスの存在によってリアルタイム実行に影響があることがある。これは、汎用 OS 上の 1 つのプロセスとして実行されている以上避けられない問題であり、これを回避するためには、汎用 OS に変更を加えるしかない。例えば RTLinux のようなリアルタイムスケジューラを実現した OS によって解決できると考えている。

(b) に関して、組み込み OS は特権モードのみを使用しているため、カーネルもアプリケーションも I/O 命令が自由に発行できる。一方、PC 用の OS では、カーネルが実行中は特権モード、アプリケーションが実行中は一般モードを使用するといったモードの切替えを行っている。PC 上でのエミュレーション環境はユーザーレベルなので I/O 命令は発行できない。

ただし、PC の IA アーキテクチャでは、I/O ポートをアクセスするレベル (IOPL) を変更することによってユーザーレベルからでも直接 I/O 命令を発行できる。IA アーキテクチャ用の Linux はこの機能を利用するためのシステムコール (iopl) を実装しており、例えば、X Window System の X server も、iopl を使用して直接 I/O 命令を発行している。しかし、iopl を用いるとユーザーレベルからでも割り込み禁止命令が発行できるので、カーネルのデータは破壊できないが、割り込み禁止状態のままにしてしまうことが出来る。

(c) に関しては、一般に OS はデバイスを直接見せずに抽象化するため、デバイスの割り込みをユーザーレベルに通知する仕組みを提供しない。前述の iopl を利用してもデバイスの割り込みをユーザーレベルで受け取る手法はないため、この場合でも、デバイスに変化が起きているかどうかは周期的にポーリングしなければ分からない。

これらの問題に対処するため、Windows 用のものには [8, 5] などの手法が提案・製品化されている。これは、PC 上の OS に汎用のデバイスドライバを組み込み、ユーザーレベルからの I/O を代わりに発行したり、デバイスからの割り込みをユーザーレベルへ伝達する仕組みである。我々は同様な仕組みを Linux 上で実装し、そのリアルタイム性を検証した。

3.2 エミュレーションのメカニズム

PC 上で組み込みシステムをエミュレーションするためのメカニズムは、2 つのコンポーネントからなる。一つは汎用デバイスドライバであり、もう一つは API ライブラリである。

汎用デバイスドライバはカーネルに組み込まれて、ユーザーレベルからの I/O 命令を受け取ってデバイスに発行したり、デバイスからの割り込みをユーザーレベルに伝える役割を担っている (図 2)。

ユーザーレベルから直接 I/O 命令を発行することは出来ないため、ユーザーレベルでは I/O の API を用意し、そのシンタックスは in/out 命令と同じである。この API はユーザーレベルのライブラリの形態で提供され、アプリケーションの作成時にリンクすることで使用できる。また、割り込みハンドラを定義する API なども同様にライブラリで提供している。これらライブラリ関数は全て ioctl システムコールを用いて汎用デバイスドライバに制御内容を伝えている。I/O ポートへのアクセスの際には、対象となるデバイス毎に I/O ポートのアドレスのチェックを行っているため、制御するデバイス以外にはアクセス出来ない。

ユーザーレベルでの実行となって安全性は向上したが、一方で、割り込み応答遅延などの問題が危惧される。今回想定している PC の CPU 速度は、通常の組み込み用途で使われる CPU と比べて 10 ~ 15 倍ほど高速であるため、この高速性を活用すればリアルタイムに近い精度での制御が可能であると考えられる。4 章で、このリアルタイム性能を実際の PC 上で検証する。

3.3 プロトタイプ実装

今回のプロトタイプの実装には Linux カーネル (version 2.2.12) を使用した。Linux カーネル自体には変更を加えず、汎用デバイスドライバはカーネルモジュールとして動的にインストール・アンインストールが出来るようにしている。

カーネルからユーザーレベルへデバイスの割り込みを通知するためには FIFO を使用する。この FIFO はキャラクターデバイスで実装されている。デバイスからの割り込みは非同期的に発生するので、この FIFO にも非同期 I/O の設定がしてあり、割り込みをユーザーレベルに伝えるときは SIGIO のシグナルが送られる (図 2)。このため、ユーザーレベルでの割り込みハンドラは実際にはシグナルハンドラから呼び出される。

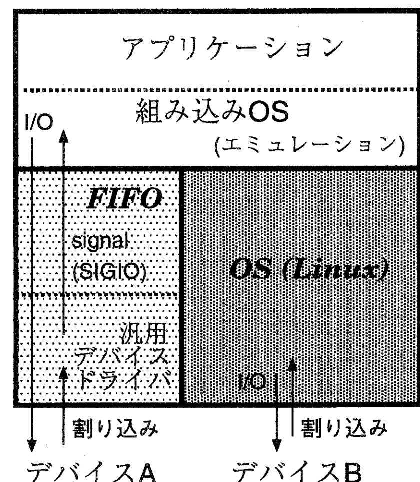

図 2: エミュレーション環境を実現するメカニズム

Linux での実装における課題は、主に以下の 2 点である。

- デバイスとの DMA 転送の際などに物理アドレスが必要
- デバイスからの割り込みをエミュレーションする方法

通常、汎用 OS のユーザーレベルは仮想アドレス空間を使用しており、物理アドレスを必要とすることがない。しかし、カーネルレベルでのデバイスドライバと同様に、ユーザーレベルのエミュレーション環境で動作しているデバイスドライバでも、DMA 転送などを設定する際に物理アドレスが必要となる。このため、カーネルが持っているページテーブルの情報をもとに、「物理アドレス ↔ 論理アドレス」相互変換のための API を用意した。

デバイスからの割り込みをシグナルを用いて伝えるメカニズムは前述したが、更に、これをエミュレーションする際に以下のような課題が生じる。

1. ユーザーレベルで割り込み禁止を実現する方法
2. デバイスからの多重割り込みと割り込み優先順位をシグナルを用いて実現する方法

以下、これらを順に検討していく。

1. デバイスドライバで発行する CPU の割り込みに関する命令 (cli, sti) の目的がデータの排他制御ならば、これらはシグナル (SIGIO) をブロックさせることで実現できる。
2. 組み込み OS では、割り込みハンドラに優先順位をつけることが出来る。今回のプロトタイプでは、複数のデバイスを制御するためにそれぞれのデバイス毎に FIFO を割り当てている。割り込みをかけたデバイスを特定するために、ユーザーレベルのライブラリで select システムコールを用いて SIGIO を送ってきたキャラクターデバイスを特定し、それに対応するデバイスドライバを呼び出している。
   多重割り込みをエミュレーションする際に、割り込み禁止のためにシグナルをブロックしてしまうと、現在実行中の割り込みより優先度の高い割り込みがブロックされて受け取れなくなる。このため、多重割り込みを使用する場合には、割り込み禁止をシグナルブロックで実現せずに、シグナルハンドラ内で優先度を考慮した実装をする必要がある。具体的には、多重割り込みがかかった場合、シグナルハンドラではより優先順位の高い割り込みがかかってきたときにはそれを先に実行し、そうでなければその割り込みに関するペンディングのフラグを立ててシグナルハンドラから復帰するようにする。割り込みハンドラが終了した際には、ペンディングされているフラグがあるかどうかを確認して、あればその割り込みを実行する。

4 性能測定

実際に PC 上で組み込みシステムをエミュレーションする際のリアルタイム性を検証するために、割り込み応答遅延と I/O 命令の遅延を測定した。測定環境は Pentium II 366MHz を搭載した PC で、遅延の測定には Pentium II の TSR (time stamp register) を使用した。

4.1 割り込み応答遅延の測定

図 3 は PC 上のエミュレーション環境での割り込みモデルを示したものである。エミュレーションの場合、ユーザーレベルで 1 つのプロセスとして動く組み込み OS の ISR までが割り込み応答遅延 (③) となる。割り込み処理の高速化には、カーネルの割り込み禁止区間と、カーネルが割り込みを検知してから処理されるまでの遅延の 2 つの課題がある。前者は 図 3 における ① であり、後者は ② である。今回の測定では、デバイスが割り込みをかけた正確な時刻を知る手段を用意しなかったので、正確な割り込み応答遅延 (③) を測定するのではなく、① と ② をそれぞれ測定して、それらの和 (つまり、① + ② ≤ ③) を示して割り込み応答遅延の最悪値を示す。

割り込み禁止区間の時間を測定する目的は、カーネルが割り込みを検知するまでの時間を測定するためである。CPU が割り込み禁止状態にあるときにはデバイスからの割り込みをカーネルが受け取

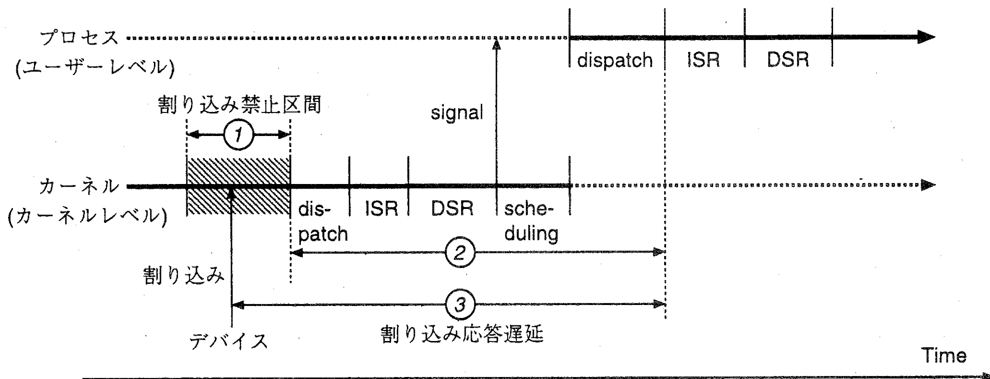

図 3: 割り込み伝達の遷移図

表 1: Linux カーネルの割り込み禁止区間の計測

	1 回目	2 回目	3 回目	4 回目	5 回目
最大値 (μs)	40.6	39.2	35.5	40.6	40.5

ることは出来ず、これは、カーネルの割り込み禁止区間の時間に左右されるからである。

測定した区間は、Linux カーネルが発行する割り込み禁止命令 (cli) から許可命令 (sti) の間、および、割り込みが発生してから復帰 (iret) するか割り込み許可 (sti) が発行されるまで間、の 2 種類である。前者は Linux カーネルの cli() 関数と sti() 関数をそれぞれフックして時間計測の命令を追加し、後者は割り込みハンドラの最初と最後に時間計測の命令を追加して計測した。

今回は、Linux カーネルを再コンパイルする作業中にカーネルが発行した割り込み禁止の時間を測定した。測定は 5 回行った。それぞれの最大値を表 1 に示す。また、3 回目の測定の分布を累計比率で示したのが図 4 である。測定結果より、割り込み禁止区間の最大区間は 40.6μs なので、図 3 における ① の時間は 0〜41μs となる。また図 4 より、これらのうちの 95% は 4μs 以下であることが分かる。

次に割り込み応答遅延だが、本稿のエミュレーション環境では、ユーザーレベルにある組み込み OS の ISR までが割り込み応答遅延となるため (図 3)、Linux カーネル内でのオーバヘッドが大きいとユーザーレベルでのリアルタイム性の実現が難

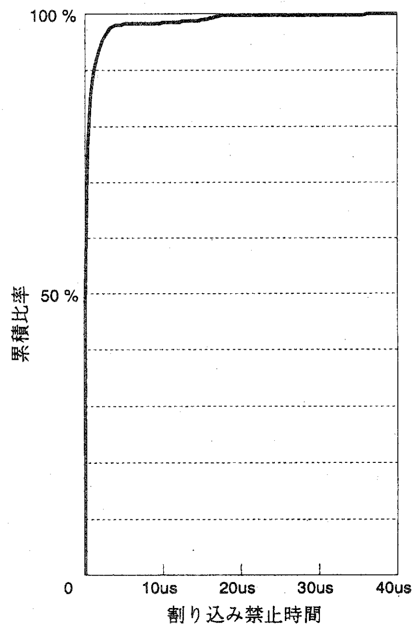

図 4: 割り込み禁止区間の時間 (累積比率)

しくなる。しかし、高速な CPU を用いることで、この Linux カーネル内のオーバヘッドが相対的に縮小されるため、ユーザーレベルでのリアルタイム性が現実的になると期待した。

測定したのは、PS/2 マウスの割り込みをカーネルが受け取ってから、SIGIO を経由してユーザーレベルの組み込み OS の ISR で受け取るまでの遅延である。これは、割り込みが 1 万回起こるまで測定した。測定の結果、最小遅延が 15.2μs、最大

遅延が 57.9μs、平均遅延は 16.1μs であった。従って、② の時間は 15〜58μs となる。

これらの計測の結果をまとめると、① の時間は 0〜41μs、② の時間は 15〜58μs であるため、③ の時間は 15〜99μs となる。このうち、平均では 20μs 程度であるため、高速な CPU を用いることにより、割り込み応答遅延での一定のリアルタイム性を実現できることが示せた。

4.2 I/O 命令の遅延

ユーザーレベルからデバイスに対して発行する I/O 命令の遅延を測定した結果が表 2 である。実験は次の 3 種類を行った。

1. カーネルレベル: カーネルの中で outb 命令を 1 命令実行する時間と 1024 命令繰り返し実行する時間
2. ユーザーレベル: 3.3 節で説明した汎用デバイスドライバを経由して、ユーザーレベルから outb 命令を 1 命令実行する時間と 1024 命令繰り返し実行する時間
3. ユーザーレベル (配列使用): 2 と同様に汎用デバイスドライバを経由して発行しているが、outb 命令を 1024 回発行する命令列を配列に入れておき、カーネルレベルでまとめて発行する時間

なお、outb 命令の出力先は全てパラレルポートである。

表 2: I/O 命令発行の遅延

	1 回	連続 (1024 命令)
カーネルレベル	0.6	585
ユーザーレベル	1.7	1770
ユーザーレベル (配列使用)	—	1106

(μs)

測定の結果、カーネルレベルから out 命令を 1 命令発行する場合が 0.6μs であったのに対して、ユーザーレベルから汎用デバイスドライバを経由して発行する場合は 1.7μs と 3 倍の時間がかかっている。I/O 命令を汎用デバイスドライバに伝えるために使用している ioctl システムコールだけでも 1.0μs 以上かかっているため、より遅延を小さくするにはシステムコールを用いず、カーネルに手を加えて専用のソフトウェアトラップで実装するなどの手法が考えられる。

また、連続して I/O 命令を発行する場合などで遅延が問題になる場合は、ユーザーレベルで配列に命令を蓄えておいて、汎用デバイスドライバで一括してその命令列を発行する方法も有効である。この手法では、ユーザーレベルとカーネルレベルを切替える回数を減らすことで更に遅延を小さくすることが可能である。実際、1024 命令の発行で 1106μs (1 命令あたり 1.1μs) になり、カーネルレベルで発行する場合と比べて約 1.9 倍にまで抑えられた。ただしこの手法は、例えば in 命令で受け取ったデータに応じて挙動を変えることは出来ないため、out 命令を連続して発行する際に特に有用である。

5 関連研究

Linux をリアルタイム拡張する研究がいくつか行われており、代表例の RTLinux[7] はハードリアルタイムを実現している。RTLinux では Linux カーネルの最下層の割り込みハンドラ部分に大きな変更が施されており、リアルタイムモジュールを組み込んだ際には、Linux カーネルよりリアルタイムモジュールを優先させたスケジューリングも可能である。ただし、ハードリアルタイムで動作できるリアルタイムモジュールはカーネルレベルで実行されるため、誤動作の際にカーネルを破壊することもある。

DARMA[6] は PC から機器を制御するために、リアルタイム OS と汎用 OS の 2 つの OS を載せる手法である。リアルタイム OS にはリアルタイム処理を任せ、汎用 OS にその他のユーザーインターフェースに関する処理を任せ、必要に応じて OS を切替える。

SPIN[3] はカーネルモジュールを安全に実行するための機構を持つマイクロカーネルの研究であ

る。カーネルモジュールはModula-3という型安全な言語を用いて作成され、そのモジュールを動的にコンパイルしてカーネル内で実行する。安全性を確保した半面、使用できる言語に制約がかかっている。

また、拡張可能OSのデバイスドライバを安全に実装する手法として、多段階保護機構[4]が提案されている。これは、デバイスドライバの実装フェーズに応じて保護の強さを開発者が変えられる機構である。

更に、カーネルの実行にトランザクションを用いたVINO[2]もある。

6 まとめ

組み込みシステムではOSの構造を簡潔にすることでリアルタイム性を実現していた。このような構造は、完成したシステムにとってはオーバヘッドなく動作出来るため便利であるが、アプリケーションの誤動作によりカーネルを破壊できてしまうため、特に開発段階では困難がともなう。

本研究では、汎用PC用のOSのユーザーレベルでリアルタイムシステムをエミュレーションする環境を構築すれば、仮想アドレス空間の利用やCPUモードを切替えて動作することにより安全に組み込みシステムが開発できることに着目した。汎用OSの問題であったリアルタイム性は、最近の高速なCPUを用いることで確保できる。本稿では、高速なCPUを活用したPCを用いてそのリアルタイム性能を検証した。実際にLinuxカーネルを用いて366MHzのPentium IIを使用したPC上で測定した結果より、デバイスが割り込みをかけてからユーザーレベルで割り込みを受け取るまでの遅延は0～99μs程度、平均では20 μsであると予想される。また、I/O命令の遅延は1.7μs程度であった。割り込み応答遅延もI/O命令の遅延も、より高速なプロセッサを使用することによって更に短縮できると考えられる。

参考文献

[1] Erik Cota–Robles, A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98, In *Proceedings of the USENIX 3rd Symposium on Operating Systems Design and Implementation*, 1999

[2] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp 213–227, 1996

[3] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System, In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, 1995

[4] 光来健一、千葉滋、益田隆司, 多段階保護機構：拡張可能OSの新しいFail-safe機構, 情報処理学会論文誌, 39巻11号, pp. 3054–3064, 1998

[5] 片山吉章、竹並春佳、川上武、二村祐地, 汎用PCにおけるリアルタイム制御機構エミュレータの実現, 情報処理学会第57回全国大会, pp. 1-54–55, 1998

[6] 齋藤雅彦、加藤直、大野洋、中村智明、上脇正、井上太郎, 組込み向けデュアルOS実行システムDARMAの開発, 情報処理学会第59回全国大会, 1999

[7] Victor Yodaiken, The RTLinux Manifesto, 1999

[8] Blue Water Systems Inc, WinRT version 3.5 Users Manual