Tasklet 方式による Linux のリアルタイム性向上

Tasklet Mechanism to Improve Real Time Nature for Linux

○齋藤 真輝 [*1]　曽我 正信 [*2]　中出 実 [*2]　山中 禎詠 [*2]
杉岡 利信 [*3]　阿部 昌裕 [*1]　片桐 敦 [*1]　小島 一元 [*4]
比屋根 一雄[*5]　飯尾 淳 [*5]　谷田部 智之[*5]

概要　我々はプラント制御等の Mission Critical システムにおいて Linux を組込み用途で利用するために、(1) 制御演算が一定周期で確実に動作するリアルタイム性の不足、(2) 数年に及ぶ連続稼働を実現するための信頼性の不足、(3) トラブル発生時に原因究明するための記録解析機能の不足、の 3 課題を解決する開発プロジェクト R^{2}Linux を実施した。本稿では R^{2}Linux の中で主にリアルタイム性の向上に関する改良を述べる。通常のカーネル Linux-2.4.20 は、ファイル入出力・通信・メモリ解放によりリアルタイム性が大きく妨害される。この対策として Low Latency Patch と Preemption Patch が提案された。我々がターゲットとしたハードウェアでは、定周期タスク起動までの最大遅れ時間を通常のカーネルの 42msec から 6msec まで短縮できるが、制御用途ではまだ充分ではない。そこで我々は更に改良し、最大遅れ時間を 200μsec 未満にまで短縮することができた。割込みそのものがリアルタイム性を阻害すると判明したため、ハード割込みとソフト割込みの割込み処理全体を tasklet 化によってソフト割込み化し、高い優先度のタスクが動作している場合はその処理を保留し後回しにすることで解決した。この方法は多くのデバイスドライバにも適用可能であり、劇的な効果が得られると予想される。なお、本開発の成果は R^{2}Linux のホームページ `http://r2linux.sourceforge.jp/` にて公開されている。

1　はじめに

近年、組込み機器の開発プラットフォームにおいて Linux が注目されている。特に携帯情報端末 (PDA) やホームサーバ等において、組込み Linux を採用する事例が増えつつある。しかし、同じ組込み機器でも FA、プラント、航空、医療機器等の Mission Critical システムにおいて利用するには、現在利用されている市販のリアルタイム OS と比べて以下の 3 つの課題が存在する。

(1) 制御演算が一定周期で確実に動作するリアルタイム性の不足
(2) 数年に及ぶ連続稼働を実現するための信頼性の不足
(3) トラブル発生時に原因究明するためのシステム状態記録解析機能の不足

我々は、組込みプロセッサ SH4 を対象に、以上の 3 点の課題を解決する改良開発を行った。すなわち、Linux のリアルタイム性の向上、信頼性の向上、トラブル解析機能の開発である。この成果を用いることによって、Mission Critical システムに要求される長期的総合的な高信頼性を実現することができ、組込み Linux の適用分野が大幅に拡大することが期待できる。

本稿では、上記の 3 つの課題の内 (1) リアルタイム性の不足について詳しく解説する。Linux ではファイル入出力、通信処理などにより、定周期のタスク起動が妨害される。この解決策として RT-Linux や MontaVista などのリアルタイム Linux が開発された。しかし実際には、RT-Linux ではリアルタイムプログラムはカーネルモードで動作するため、システムコールは利用できない。Ethernet 通信のシステムコールを RT-Linux で呼ぶには、リアルタイムタスクから RT-FIFO を介し、通常の Linux プロセスでシステムコールを呼び出し、その結果を RT-FIFO を介して戻す必要がある。このとき通常プロセスは最も低い優先度で実行されるため、これが律速段階となりかなり待つ必要が生じてしまう。つまり、通信の優先度を考慮した複数の通信処理タスクの優先度設定ができないという問題がある。また、Monta

[*1] 株式会社エー・アンド・デイ
[*2] 三菱重工業株式会社 長崎研究所
[*3] 株式会社アイ・ティ・オー
[*4] フリープログラマ
[*5] 株式会社三菱総合研究所

Vista は、通常の Linux 同様にファイル入出力等において、タスク切替えの遅れが発生する。これを解決するために 1024 レベルの優先度を有するスケジューラを提供しているが、デバイスドライバ内部で長時間ロックをかけて処理が続く場合には遅れの発生を防止できない [1]。

この対策として Low Latency Patch [2] や Preemption Patch [3] が提案されている。Low Latency Patch は、カーネルやデバイスドライバ中に存在するタスク切替え可能箇所を試行錯誤的に見つけ、その処理を追加するパッチである。このパッチは性能限界がはっきりせず、デバイスによっては全くパッチをあてていない。また、Preemption Patch は応答速度向上のためにタスクの切替（プリエンプション）を可能にするパッチであるが、現時点では性能が十分ではない。

我々はこれまで重視されてこなかったデバイスドライバを含め、リアルタイム性への阻害程度を調査し、Low Latency Patch と Preemption Patch をベースに周期タスク起動精度向上機能を実現した。

2 リアルタイム性改良の概要

今回改良対象としたのは、Linux カーネルおよびデバイスドライバである。Low Latency Patch や Preemption Patch を出発点として、割り込み禁止処理を行っている部分を明確にした上で、絶対に必要な部分とそうでない部分を切り分け、必要でない部分の手続きを見直し、割り込み応答性を向上させる改良を実施する。同様に、スピンロックなどの OS が提供するサービスを使った排他制御の部分を見直し、できるだけ細かい単位でタスクの切替（プリエンプション）を可能にする改良を実施する。以上の方法により定周期で起動するタスクの起動精度を向上させる。

■改良方針

Linux カーネル自身および各種ドライバは、各種処理中に他の処理が割り込むのを防ぐために、一定期間割り込みをマスク（禁止）するという手続きを踏んでいる。しかしこの手続きは、適切に行なわないとシステムの応答速度に悪影響を与える。場合によっては、必要のない割り込み禁止処理により、優先度の高い処理の実行時期が遅れてしまう（レイテンシが長くなる）ことにつながるからである。そこで、Low Latency Patch を出発点として、割り込み禁止処理を行っている部分を明確にした上で、絶対に必要な部分とそうでない部分を切り分け、必要でない部分の手続きを見直し、割り込み応答性を向上させる。

システムのリアルタイム性を改善するには、割り込み禁止期間の見直しだけでは十分でない。セマフォ・ミューテックス・スピンロックなどの OS が提供するサービスを使い、特定の処理を行っている最中に同様の処理が行われないようにしている部分が多数存在する。これもシステムの応答速度に悪影響を与える大きな原因の一つである。タスクの切替（プリエンプション）を可能にする対策が求められる。割り込み応答性を向上と同様に、問題箇所の洗い出し・改善可能部分の切り出し・対策と効果の確認を行なう。

周期タスクの起動精度の悪化要因としては、上記 2 つの要素が個別に働くだけではなく、それらの組み合わせによる相乗効果や、ハードウエアデバイスへの不適切なアクセス方法に起因するものなど、現時点で明らかに認識されていないものも想定される。これら阻害要因の主要なものを特定し、分析し、改善・対策を行うことにより、最終的に周期タスクの起動精度の向上を実現する。

■ドライバの tasklet 化による起動遅れの改善

今回の改良の大きな特長として、ドライバの tasklet 化が挙げられる。tasklet はボトムハーフに代わる機能としてカーネル 2.4 から採用された。

ドライバを tasklet 化する効果として、ISR の短縮による割り込み応答性の改善があるが、起動遅れ自体は必ずしも改善しない。これは tasklet はソフト割り込みの中で実行されるため、結局 tasklet の処理が終了するまでタスクスイッチできないからである。

今回の改良では、

(1) tasklet と `disable_irq()`、`enable_irq()` を組み合わせることにより、リアルタイムタスク実行中はハードウエア割り込みを止めてしまう。
(2) ソフト割り込みを制御することで、リアルタイムタスク実行中は tasklet の実行も禁止してしまう。

という 2 つの方式を組み合わせることにより、起動遅れを短縮した所が特長である。このような目的で tasklet を使用したのは、我々が最初であろう。

■改良手順

SH 向け正式版 2.4.20 カーネル [4] を使用し、これに Low Latency Patch、Preemption Patch を当てたものをベースとし、以下の処理をさせた場合の周期タスクの起動遅れを今回改良したイベントログ機

能 (A.3 節に後述) を使用して計測する。

- プログラムの起動
- ファイル入出力
- ネットワークアクセス
- メモリの確保と解放

イベントログを分析することにより、周期タスクの起動精度の悪化要因を特定し、改善・対策を行う。対策したカーネルにおける周期タスクの起動遅れの計測データを解析し、阻害要因を特定し、主要なものから順に対策の検討・対策の適用・効果の確認を行う。目標性能が確保できまで上記手順を繰り返す。

3 リアルタイム性改良の詳細

3.1 ロックの分割

spin_lock() や local_bh_disable() などがタスクスイッチを妨害している場所があった。このような場合は状況により以下のような対策を行ない、ロック時間を短縮し起動遅れを低減した。

長い spin_lock 期間がタスクの起動を阻害している部分がある。この様な場合は Low Latency Patch の手法を取り入れ、プリエンプションポイントを挿入することによりロックを分割し、タスクスイッチ可能とすることで速やかに周期タスクが起動するようにした。

また、ソフト割り込みを実行する do_softirq() の中で処理に時間がかかる部分があった。do_softirq() の中は local_bh_disable() が呼ばれているためタスクスイッチできない。このため net_rx_action()、net_tx_action() など時間のかかるソフト割り込みは再スケジューリングしてすぐに終了するように対策した。

上記の手法により以下の関数を修正した。

- net/core/sock.c __release_sock()
- net/tcpv4/tcp_output.c tcp_write_xmit()
- mm/memory.c zap_page_range()
- net/core/dev.c net_rx_action()
- net/core/dev.c net_tx_action()
- net/core/dev.c process_backlog
- mm/vmscan.c refill_inactive()
- drivers/char/n_tty.c write_chan()
- fs/ext2/balloc.c ext2_free_blocks()
- mm/vmscan.c swap_out_pmd()
- net/ipv4/route.c rt_check_expire()
- fs/buffer.c write_some_buffers()
- net/sched/sch_generic.c qdisc_restart()
- mm/memory.c zap_pte_range()
- net/ipv4/tcp.c tcp_close()
- kernel/drivers/char/sh-sci.c put_char()

3.2 IDE ドライバの tasklet 化

アプリケーション起動・終了、ファイル I/O、FTP 転送などディスクアクセスが伴う処理によって生じる起動遅れの原因として、IDE の割り込み処理がドライバやアプリケーションの起動をブロックする現象があった。この問題を解決するために IDE ドライバを tasklet 化し、リアルタイムタスク実行中は tasklet の処理及び IDE デバイスからの割り込みを禁止する対策をした。

IDE 割り込みの中で CF からデータを読み出していたが、この処理に時間がかかっていた。割込み処理中はタスクスイッチできないためデータ転送部分を tasklet 化し、do_softirq() の中で処理するように変更した。

リアルタイムタスク動作中はデータ転送をせずに tasklet を再スケジュールして終了する。この対策によりデータ転送が終了して割り込みを許可するまで IDE 割り込みが禁止されるので起動遅れが少なくなった。

```
+++drivers/ide/ide.c

-void ide_intr (int irq, void *dev_id,
-                struct pt_regs *regs)
+static void ide_handle_intr(int irq,
+                            void  *dev_id)
 {
  unsigned long flags;
  ide_hwgroup_t *hwgroup
             = (ide_hwgroup_t *)dev_id;
  ide_hwif_t *hwif;
  ide_drive_t *drive;
  ide_handler_t *handler;
  ide_startstop_t startstop;
   ( ••• 省略 ••• )
 }

+#ifdef CONFIG_IDE_TASKLET
+static void ide_do_tasklet(unsigned long
+                           dev_id)
+{
+  ide_hwgroup_t *hwgroup
+              = (ide_hwgroup_t *)dev_id;
+  ide_hwif_t *hwif = hwgroup->hwif;
+  int irq = hwif->irq;
+
+  if (need_hi_tasklet_sched_ide()) {
+    ide_tasklet[hwif->index].data = dev_id;
+    tasklet_hi_schedule(&ide_tasklet
+                        [hwif->index]);
+    return;
+  }
+
+  ide_handle_intr(irq, hwgroup);
+  enable_irq(irq);
+}

+void ide_intr (int irq, void *dev_id,
+               struct pt_regs *regs)
+{
+  ide_hwgroup_t *hwgroup
+               = (ide_hwgroup_t *)dev_id;
```

```
+  ide_hwif_t *hwif = hwgroup->hwif;
+
+  disable_irq(irq);
+  ide_tasklet[hwif->index].data
+                    = (unsigned long)dev_id;
+  tasklet_hi_schedule(&ide_tasklet
+                              [hwif->index]);
+}
+#else
+void ide_intr (int irq, void *dev_id,
                       struct pt_regs *regs)
+{
+  ide_handle_intr(irq, dev_id);
+}
+#endif
```

3.3 Ethernet ドライバの tasklet 化

ネットワーク通信時に起動遅れが生じる原因として、Ethernet の割り込み処理がドライバやアプリケーションの起動をブロックする現象があった。この問題を解決するために IDE ドライバの tasklet 化と同様に Ethernet ドライバを tasklet 化し、リアルタイムタスク実行中は tasklet の処理及び IDE デバイスからの割り込みを禁止する対策をした。

ether の割り込みは 1 回 100μsec 以下であるが、何回か連続して入ることがある。そこで IDE ドライバと同様にデータ処理部分を tasklet 化し、`do_softirq()` の中で処理するように変更した。

IDE tasklet と同様にリアルタイムタスク動作中はパケット処理をせずに tasklet を再スケジュールして終了する。パケット処理が終了して割り込みを許可するまで Ethernet 割り込みが禁止されるので起動遅れが少なくなった。

上記対策をしても net tasklet の内部はループ構造となっていて実行に時間がかかる。そこでリアルタイムタスク動作中は処理を中止して終了するように対策した。

```
+++drivers/net/smc91111.c

-static void smc_interrupt
-                    (int irq, void * dev_id,
-                      struct pt_regs * regs)
+static void smc_handle_interrupt
+                    (int irq, void * dev_id)
 {
  struct net_device *dev   = dev_id;
  int ioaddr     = dev->base_addr;
  struct smc_local *lp
             = (struct smc_local *)dev->priv;
   ( ••• 省略 ••• )

  /* set a timeout value,
     so I don't stay here forever */
  timeout = 4;

  PRINTK2(KERN_WARNING
      "%s: MASK IS %x \n", dev->name, mask);
  do {
+#ifdef CONFIG_NET_TASKLET
+    if (need_hi_tasklet_sched_net())
+      break;
+#endif
    /* read the status flag, and mask it */
    status = inb( ioaddr + INT_REG ) & mask;
    if (!status )
      break;
   ( ••• 省略 ••• )
 }

+#ifdef CONFIG_NET_TASKLET
+static void smc_do_tasklet
+                    (unsigned long dev_id)
+{
+  struct net_device *dev
+        = (struct net_device *)dev_id;
+  struct smc_local *lp
+        = (struct smc_local *)dev->priv;
+  int irq = dev->irq;
+
+  if (need_hi_tasklet_sched_net()) {
+    tasklet_hi_schedule(&lp->smc_tasklet);
+    return;
+  }
+
+  smc_handle_interrupt(irq, dev);
+
+  if (need_hi_tasklet_sched_net())
+    tasklet_hi_schedule(&lp->smc_tasklet);
+  else
+    enable_irq(irq);
+}

+static void smc_interrupt(int irq,
+ void * dev_id,  struct pt_regs * regs)
+{
+  struct net_device *dev = dev_id;
+  struct smc_local *lp
+           = (struct smc_local *)dev->priv;
+
+  disable_irq(irq);
+  tasklet_hi_schedule(&lp->smc_tasklet);
+}
+#else
+static void smc_interrupt
+                    (int irq, void * dev_id,
+                      struct pt_regs * regs)
+{
+  smc_handle_interrupt(irq, dev_id);
+}
+#endif
```

3.4 デバイスドライバの tasklet 化の方法

他の CPU や他のデバイスドライバに対し、今回実施した tasklet 方式によるリアルタイム性向上の行う方法を述べる。

まず、デバイス構造体に tasklet 構造体を追加する。割り込みハンドラ内で `irq` と `regs` を使用している場合は、同様に `int regs` と `struct pt_regs *regs` を追加しておくと良い。デバイス構造体初期化時に `tasklet_init()` を呼び出し、tasklet 構造体も同時に初期化する。この時、`tasklet_init()` の第 3 引数にデバイス構造体のアドレスを渡してやる。これにより、デバイス毎に tasklet 構造体を持つことが出来るようになる。デバ

イス構造体は request_irq() の引数として渡されるため、この近くを探せばデバイス構造体の初期化部分が見つかるはずである。

次に、従来の割り込みハンドラはリネームして、新しい割り込みハンドラを作成する。新しい割り込みハンドラでは最初に disable_irq() を呼び出し、デバイスに対する割り込みを禁止しておく。必要ならばここで irq と regs をデバイス構造体にコピーする。次にデバイス構造体から tasklet 構造体を取り出し、tasklet 実行のため tasklet_hi_schedule() を呼び出し、スケジュールする。

最後に、tasklet が呼び出されたら引数としてデバイス構造体のアドレスが渡ってくるので、これを引数として先程リネームした従来の割り込みハンドラを呼びだす。従来の割り込みハンドラの中で時間がかかりそうな場所には、リアルタイムタスク実行中のフラグが立った場合すぐに処理を中止するためのチェックポイントをあらかじめ挿入しておくとよい。tasklet の最後ではリアルタイムタスク実行中のフラグをチェックして、フラグが立っている場合は tasklet を再スケジュールして終了する。それ以外の場合には enable_irq() を呼び出し、割り込み禁止を解除してから終了する。

リアルタイムタスク実行中フラグは以下のようにして制御する。try_to_wake_up() の中でウェークアップさせられるタスクの rt_priority が閾値以上の場合はフラグを立てる。schedule() の中で next の rt_priority が閾値以下であればフラグを落とす。

この方式は非常に単純でありオーバーヘッドもほとんど発生しないが起動遅れの改善には大きな効果を発揮する。また、特にアーキテクチャに依存している部分も無いため、他の CPU でも同様の効果を発揮することが期待できる。

3.5 リアルタイム優先度に応じたソフト割込のマスク

tasklet 方式は、ハード割り込み処理をソフト割り込み化し、リアルタイムタスク動作時にはこれらの割り込み処理をマスクする方式である。

リアルタイムタスクが動作中か否かでマスク設定を行う。

* リアルタイムタスク
  → ソフト割り込みをマスクする
* 非リアルタイムタスク
  → ソフト割り込みをマスクしない

実際の制御装置においてはリアルタイムタスクがほとんどであり、さらに IDE 割込処理（ファイル入出力）は低い優先度の処理であり、NET 割込処理（通信）は優先度が高い処理となる。このため 2 つを分離し、それぞれに対してその動作をマスクするリアルタイム優先度を設定することにした。

リアルタイム優先度は 1〜99（99 は優先度が高い）の範囲である。カーネル生成時の menu config において設定できるようにした。IDE 処理は優先度 10 がデフォルトであり、10 以上を設定すると IDE 処理がマスクされる。NET 処理は優先度 90 がデフォルトであり、90 以上を設定すると NET 処理がマスクされる。

この対策により、リアルタイムタスクか否かによらずタスク切り替え可能な場合に常にソフト割り込みが中断させられることがなくなり、リアルタイムタスクの実行とネットワーク通信、ファイル入出力を両立できるようになった。

3.6 ソフト割り込みの制御

ソフト割り込みがドライバやアプリの起動をブロックしていた。この問題を解決するためにリアルタイムタスクの実行中はソフト割り込みの実行を禁止するように対策した。

ハードウエアからの割り込みが入るとその度にソフト割り込みが実行され、これがドライバやアプリの起動をブロックしていた。そこで、ソフト割り込みを実行する do_softirq() でリアルタイムタスクの実行中はソフト割り込みを起動しないように対策した。

3.7 その他のリアルタイム性改良

■ins/outs の高速化

アプリケーション起動・終了、ファイル I/O、FTP 転送などディスクアクセスが伴う処理によって生じる起動遅れの原因として、IDE の tasklet がスケジューラの呼び出しをブロックする現象があった。具体的には、ポートから連続してデータを転送する ins/outs の処理が非効率的だった。高速に処理が実行されている評価ボード QT-PIDS のコードを参考に、ループを展開する手法でブロック期間を短くする対策をした。

この改良は評価ボード FXCPU01 と SolutionEngine MS7751RE01 については効果があった。SolutionEngine MS7751RE01 では insw でかかる時間が 200μsec から 170μsec に減少した。Solution

表 1: ボード別周期タスク起動遅れ (単位: μsec)

	QT-PIDS 通常のカーネル	QT-PIDS Preemption + Low Latency Patch	改良カーネル				
			QT-PIDS	Solition Engine MS7751RE01	Solution Platform SH4RPCI	SH-4PCI with Linux	FXCPU01
アプリ起動終了	2,235	532	147	172	237	289	337
ファイル I/O	41,994	411	150	195	250	245	274
ftp get	9,094	6,432	168	331	240	369	303
ftp put	42,129	1,514	173	319	214	281	495
メモリ確保解放	1,440	1,418	142	172	156	254	247

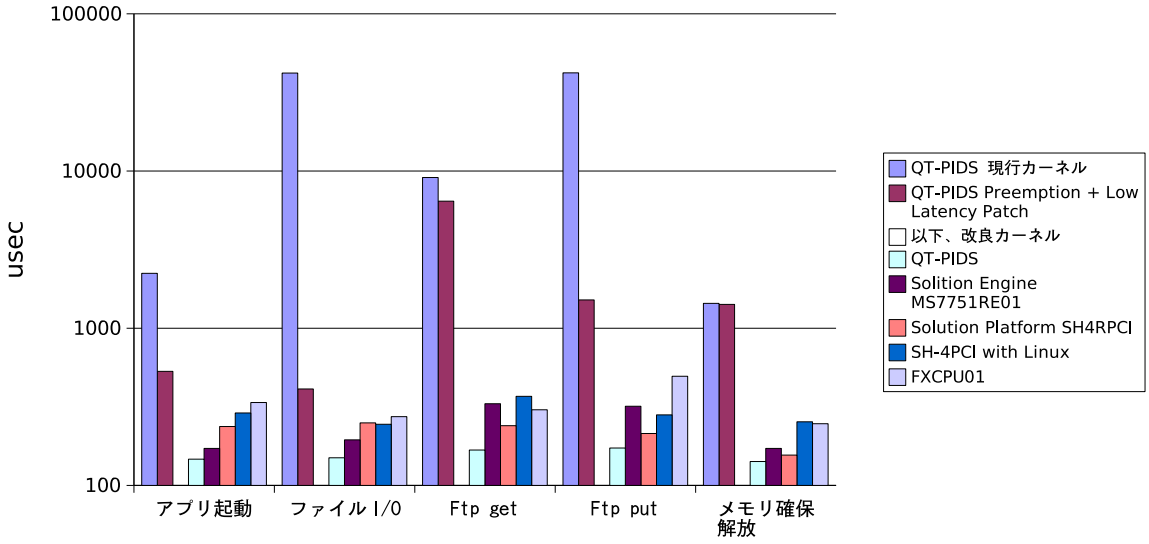

図 1: ボード別周期タスク起動遅れ

Platform SH4RPCI/SH-4PCI with Linux に関しては効果がなかったため変更しなかった。FXCPU01 で実測したところ ftp get の起動遅れが 4.1msec から 1.6msec に短縮された。

■printk() のデーモン化

割り込み処理から `printk()` でエラーメッセージを表示すると、ポーリングで文字出力するためリアルタイム性を阻害する。このため `printk()` を kprintkd デーモンで処理するように改造した。

■OS 基本周期の設定

カーネルのコンフィギュレーション時に OS の基本周期 (100Hz = 10msec) を変更できるような機能を開発した。また、CPU の基本周期の値は、現状では起動時に自動的にクロックが検出され設定されるが、クロック検出精度が十分でない場合があり、起動のたびに微妙に変動することもあるので手動で固定値に設定できるような機能を提供する。さらに、基本周期の値を機種毎に固定値に設定できる機能を提供する。

4 リアルタイム性改良結果

リアルタイム性向上は表 2 の評価ボードを対象に実施した。各評価ボードの改良後の最大起動遅れ時間を表 1 および図 1 に示す。このうちオムロン（株）製 QT-PIDS（SH7750R 240MHz、メモリ 128MB）では、通常のカーネルでは一定周期タスク起動までの遅れ時間は最大 42msec にもなった。LowLatency + Preemption パッチを適用すると、6msec まで短縮された。我々が今回実施した改良によると最大でも 200μsec 未満まで短縮することができた。他の 4 種類の評価ボードでも、最大 500μsec に収まっている。

表 2: 対象とした SH4 プロセッサ評価ボードとデバイス

製造元	製品名
オムロン (株)	QT-PIDS
アドバネット (株)	FXCPU01
日立超 LSI システムズ (株)	MS7751RSE01
CQ 出版 (株)	SH4-PCI with Linux
京都マイクロコンピュータ (株)	Solution Platform for SH4R

《対象とするデバイス》

IDE	SH4 内蔵
Ethernet	am79C973, i82559, SMC91111
RS232C	SH4 内蔵 (sh-sci)
タイマ	SH4 内蔵, ds1501

5 おわりに

本稿では、デバイスドライバの tasklet 化によるリアルタイム性向上の方法について述べた。通常のカーネル kernel-2.4.20 では数十 msec におよぶ周期起動タスクの起動遅れを確実に 200μsec にまで抑え込むことに成功した。また、我々が開発した R^{2}Linux：高信頼性組込み Linux についてもその概要を示した。Mission Critical システムにおける Linux 利用の可能性が示せたと考える。今後はこれらの改良を SH-Linux カーネルに取り込み、普及を図ってゆきたい。

また、今回提案する tasklet 化は SH-Linux だけでなく、多くの Linux デバイスドライバに適用可能な方法である。是非、本手法がいろいろな場所で採用されることを期待したい。

なお、本開発の成果は R^{2}Linux のホームページ `http://r2linux.sourceforge.jp/` にて公開されている。

謝辞

本開発は独立行政法人情報処理推進機構 (IPA) の「平成 15 年度 オープンソフトウェア活用基盤整備事業」に採択された「MissionCritical システム向け組み込み型リアルタイム Linux の開発」の一部として開発された。

参考文献

[1] 仲吉一男他, “Performance Evaluation of Linux Real-Time Extensions”, JPS 第 55 回年次大会, 2000.
`http://www-online.kek.jp/~nakayosi/RT/index-rt.html`

[2] Andrew Morton, “LowLatencyPatch”, `http://www.zip.com.au/~akpm/linux/schedlat.html`

[3] “PreEmptionPatch”, `http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/v2.4/`.

[4] “SH-Linux”, `http://linuxsh.sourceforge.net/`.

[5] Daniel P. Bovet, Marco Cesati, “詳解 LINUX カーネル (第 2 版)”, オライリー・ジャパン, 2003.

[6] Alessandro Rubini, Jonathan Corbet, “LINUX デバイスドライバ 第 2 版”, オライリー・ジャパン, 2002.

[7] 日立製作所, “SH7751 シリーズ ハードウェアマニュアル 第 2 版”, 2002.

付録A　R^{2}Linux：高信頼性組込リアルタイム Linux

我々は R^{2}Linux プロジェクト (高信頼性組込みリアルタイム Linux) の中で、リアルタイム性の改良以外にもいくつかの改良・開発を行った。R^{2}Linux ではリアルタイム性向上の他に以下の機能を提供する。更に詳しい情報は R^{2}Linux のホームページ `http://r2linux.sourceforge.jp/` を参照されたい。

A.1　起動時間の短縮

デバイスドライバ、ユーザランド（起動スクリプト）の改良である。OS の起動時間を短縮するために、デバイスオートプローブを廃し、デバイス固定で起動できるように改良した。また、ユーザランドの起動スクリプトを改良し、実際の動作で必要なサービスが短時間で使用可能な状態になるようにする。改良の結果、OS のロード完了から 5 秒以内にログイン可能な状態にできた。

A.2　信頼性の向上

数年に及ぶ安定した連続稼働を実現するためには、現在の Linux が抱える以下の課題を解決した。

■メモリリークの検出

空きメモリ領域の状態を常時監視し、メモリリークを起こしているプロセス、もしくは、カーネルモジュールを特定するための情報を提供するツールを開発した。

■リング状ファイルアクセス

syslogd(システムのログを保存するデーモン) は `/var/log` にメッセージを保存するため、ディスクの空き容量を圧迫する。ディスクフルとなるのを防ぐため、`/var/log` に保存されるログデータに関してファイルサイズが一定長以上にならないようなリングファイルシステムを実現した。

■時刻カウンタオーバフロー対策

OS の基本周期 (標準 10msec) 毎にカウントアップする jiffies 変数は、497 日でラップアラウンドする (32bit がオーバフローしゼロに戻る)。このとき、jiffies 変数を時間計算用に参照しているコードの挙動が異常になる可能性がある。さらに、本開発では、起動周期設定機能によって、この周期を 1msec 程度に設定可能になるので、上記の 10 分の 1 程度の動作時間でラップアラウンドが発生する可能性がありこの問題は一層深刻なものとなる。そこで、この不具合を洗い出し、jiffies のラップアラウンドを考慮したルーチンで置き換えた。

■通信の冗長化対策

冗長化（二重化）された Ethernet の片系が故障した場合にも正常な系での通信を保証するための機能を開発した。デバイスドライバが同一となる PCI NIC(PCI Network Interface Card) デバイスを複数用いたネットワーク冗長化システムにおいて、その片系が故障した場合に CPU を再起動すると、正常な系が故障した系の IP アドレスで初期化されてしまう場合がある。この問題を回避する手段として、PCI NIC のデバイス情報を eth0, eth1 の論理デバイスへの割り付け情報として新たに付加し、この付加された情報を基に PCI NIC デバイスと eth0, eth1 論理デバイスの対応付けを行う機能を実現した。

A.3　トラブル解析

一般的に Mission Critical システムは多重系で構成される。仮に 1 台が故障してもバックアップ系が制御を引き継ぐ。故障がめったに発生しない場合においても、一度でも発生すれば原因究明できる以下のような手段を持っておくことが重要である。

■イベントログ機能

イベントログ機能とは、タスク切替、シグナル、例外、割込、システムコール、ボトムハーフ、スピンロック、割込禁止などの Linux のイベントを、時刻と共にリングバッファに保存し、トラブルの解析や、リアルタイム機能の性能の検証に適用するツールである。時刻は SH4 内蔵の未使用タイマの 1 つをフリーランさせてこの値を読みとり、時刻の代替物として記録する。クロック数がわかっているのでこのカウンタ値の差からイベント間の絶対時間を測定することができる。タイマーチップを 1 ワード読むだけであり、CPU 負荷を少なくすることができる。

採取するイベントの内容は次のとおり。

タスク切替（旧タスク→新タスク、旧タスク実行時間）
シグナル（シグナル番号）
`LowLatency` **パッチによるタスク切替可能箇所**
`Preemption` **パッチによるタスク切替可能箇所**
`BottomHalf` **開始と終了**
例外（開始と終了、例外コード）
割込（開始と終了、割込コード）
システムコール（開始と終了、システムコール番号）
ファイル読出（開始と終了）
ファイル書込（開始と終了）
スピンロック（開始と終了）
割込禁止（開始と終了）
定周期アプリケーションタスク（定周期）
`OS` **タイマー（定周期）**
`OS` **タイマー〜アプリケーションタスクの起動まで**
タイマー `1`**（定周期）**
タイマー `1`**〜アプリケーションタスクの起動まで**
ソフト割込（開始と終了）
チェックポイント機能
ユーザーアプリケーションによるイベント指定

■通信ログ

今回実現した通信ログ機能では、カーネル内部で受信データを「カーネルメモリ上のリングバッファ（カーネルリングバッファ）」へタイムスタンプを付けて保存する。保存するデータは、MAC アドレス、イサーネットタイプ、各種プロトコルヘッダ（TCP/IP 等）とデータの一部とする。尚、カーネルがクラッシュした際には、別途実現する異常時メモリダンプ機能、オンラインデバグ機能により保存内容のトレースが可能となる。

■異常時の自動メモリダンプ

起動時に予備カーネルをアプリから見えない上位番地にロードしておく。異常時に異常検知割込で異常発生時のレジスタ値をメモリ上に保存し、この予備カーネルへジャンプし、元のカーネルのメモリをファイルとしてダンプする機能。制御装置では異常発生後リセットして再起動することが多いが、この場合でもファイルを解析して異常原因を究明することができる。

■オンラインデバッガ

プログラム実行中に、カーネル内部を含むメモリの表示・変更、逆アセンブル表示、レジスタ表示などを行なうことができる機能である。