現所属　九州工業大学大学院生命体工学研究科
ishii@brain.kyutech.ac.jp

リアルタイム Linux “ RTLinux ”の導入とロボットへの適用

九州工業大学情報工学部　石井和男

Realtime Linux “RTLinux” and Its Application to Robot System

Kazuo ISHII
Kyushu Institute of Technology
email: ishii@ces.kyutech.ac.jp

1. はじめに

ロボット等のハードウェアを動作させるには、センサからのデータ取得、制御アルゴリズムの計算処理、アクチュエータへの指令等を決められた時間内に確実に処理する必要があります。そのためには、専用のプロセッサ、専用のリアルタイムOSを用いる必要があり、筆者もINMOS社のトランスピュータやMS-DOS上でプログラム開発を行いロボットの制御を行っていました。[1] トランスピュータは、ハードウェア及びソフトウェアでの並列処理が行えるため面白いプロセッサなのですが、高価である上に生産中止となりました。

リアルタイムOSとして有名なものには、VxWorksやOS-9、ITRON等があげられます。しかしながら、専用のプロセッサや周辺機器を用いてロボットのシステムを構築するには、(筆者にとって) 高価であり簡単には手が届きません。リアルタイム処理が可能で、安価な汎用プロセッサでロボットのシステムが組めないかと検討していたところ、黒田先生*からRTLinux[2]を紹介され現在に至っています。

本原稿では、Linux と MS-DOS でのプログラミングの比較、RTLinux のインストール法とその API を用いたプログラム開発、ロボットへの応用例について紹介します。

2. Linux と MS-DOS

2.1 Linux

Linux は GPL に則って無償で手にいれることができ、動作が安定していることなどからwebやメール、ファイルサーバとして利用されています。更にソースコードが公開されており、必要な機能が自由に拡張できるというロボット開発者にとって素晴らしい環境が整っています。Linux は PC-UNIX としての特徴の他に、以下のような特徴を備えています。

・複数の仮想コンソール
・多くのファイルシステムに対応
・直接 I/O ポートを扱える（ioperm 関数、iopl 関数）
・DOSemu（DOS のエミュレーション）
・SVGALIB（画面の表示）
・コミュニティーが活発
・アマチュア無線との親和性
・多くの機種でサポートされている
　（x86, Alpha, 68000, Sparc, PowerPC, etc)

Linux に関する詳しい情報は、

　http://www.linux.or.jp

に掲載されており、How To が JF というドキュメントにまとめられています。現在リリースされている安定版カーネル Linux 2.2 は、AMD、Cyrix、Celeron などCPU に最適化され、SMP に本格的に対応しているようです[3]。

2.2 Linux と MS-DOS における I/O プログラミング

ロボットシステムを開発しようとする者にとって、直接 I/O ポートを扱えるという特徴は非常に魅力的です。

入出力関数のLinxu及びMS-DOSにおける関係は、表１に示すようになります。Linuxにおける各関数の定義は、

```
#include <asm/io.h>
 unsigned char inb (unsigned short port)
 unsinged short inw (unsigned short port)
 void outb (unsgined char value, unsiged short port)
 void outw (unsgined short value, unsiged short port)
```

となっています。出力関数では、引数である出力先のアドレス番号（port）と出力値（value）の順序が逆

表１　入出力関数の比較

	Linux	MS-DOS
byte入力	inb (port)	inport (port)
word入力	inw (port)	inportw (port)
byte出力	outb (value, port)	outport (port, value)
word出力	outw (value, port)	outportw (port,walue)

*明治大学理工学部

になっているので注意が必要です。また、これらの関数の末尾に_p をつけると（例えば、outb_p)、読み込み或いは、書き出した後に遅れ（delay）をつくることができます。これらの関数を動作させるには、コンパイル時に

```
$ gcc  -O2 filename.c
```

というように、-O2 オプションが必要となります。($ はプロンプト）

上記のように直接アドレスを操作するには、他のプロセスが同じアドレスを使用していないか注意する必要があります。DOS では <dos.h> をインクルードしさえすれば入出力関数を使用できましたが、Linuxでは許可を得る必要があり、下記の関数が用意されています。

```
#include <unistd.h> /* for libc5 */
#include <sys/io.h> /* for glibc */
 int  ioperm(unsigned  long  from,  unsigned  long num,
              int turn_on);
 int iopl(int level);
```

```
#include <asm/io.h>

void outport(unsigned short int port,unsigned short
int value){
        outw(value,port);
                /* vallue と port の順序を変える */
}

short int CtlISA( int arm )
{
        int intreq;
        int access=0;

        _Base = 0x200 + arm*0x10;
        access=ioperm(_Base,16,1);
          /* _Base から 16 bytes のアクセス許可 */
//      access=iopl(3); /* iopl でも可 */
        fprintf(stderr,"I/O access=%d\n",access);

        intreq = _Base + itr;
        outport( _Base + mdr    , 0x00 );
        return(intreq);
}
```

List 1 DOS プログラムの Linux への移植例

これらの関数をinb/outb関数等の前に呼び出します。ioperm 関数は、0x3FF より小さいアドレスにアクセスするための関数であり、アドレス from から num バイトのメモリ空間に対してアクセスの許可を申請します。turn_on は１とします。成功すれば０、失敗すれば -1 が返されます。

iopl 関数は、0x3FF よりも大きいアドレス、アドレス空間全体にアクセス許可を得たいときに使用します。引数 level = 3（デフォルト 0）を入力します。返り値は ioperm と同様です。

MS-DOS プログラムを Linux 用に移植するためには、もう一つ、int 型のバイト数に注意する必要があります。int 型の大きさはシステムに依存しますが、MS-DOS では２バイト、（使用した）Linux では４バイトであり、下記のようなポインタの演算

```
int* ptr;
ptr ++;
```

では処理内容が違いますので、(short int)として宣言した方が無難です。筆者の学科では三菱重工の汎用知能アーム PA-10 と MS-DOS 用のライブラリを購入しており、上記の点に注意し一部分修正することによりDOS用のライブライリをLinux 用にコンパイルすることに成功しています。その一部分をList 1に示します。（しかしながら、PA-10 の Linux 用ライブラリはライブラリ購入者に対して無償で提供されているとのことです。）

また、アセンブラのコードに対しては、

```
__asm__( );
```

という関数が用意されており、使用例は、

```
/usr/include/asm/*.h
```

にありますのでご参照下さい。

割り込みに関しては、割り込み番号irqとそのハンドラを関数に登録するための関数として、

```
#include <linux/sched.h>
#include <linux/signal.h>
int request_irq(
        unsigned int irq, /* ＩＲＱ番号 */
        void (*handler)(int, struct pt_regs *),
                /* 割り込みハンドラ */
        unsigned long flags, /* フラグ */
       const char *device); /* デバイス名 */
```

及び、割り込みを解放するための関数として

```
void free_irq(unsigned int irq);
```

が用意されています。これらは通常のLinuxカーネル

への割り込みの登録であり、後述するRTLinuxには、rtl_request_irq()及びrtl_free_irq()等の割り込み用のAPIが用意されており、RTLinuxでロボットの行動を制御する場合は、RTLinux用のAPI関数を使う方が簡単です。

3. RTLinuxの導入

3.1 RTLinuxとは？

これまでの説明でDOSプログラマーにとって、Linuxはプログラムの移植など移行しやすい環境であることがお分かり頂けたと思います。次に、ロボットシステム或いはハードウェアシステムの開発者にとって一番の問題は時間管理です。決められた時間おいて、対象のプロセスに十分なCPU時間が割り当てられ、決められた時間内に処理が終了する事が保証されるかが重要となります。通常のLinuxのスケジューリングはTSS（タイムシェアリング）であり、プロセス全体の効率を上げるようにスケジュールされます。ミリ秒オーダーの精度であれば、POSIXに準拠したOSであれば可能です。1秒以内の応答であれば通常のスケジューラに頼っても十分であり、10秒単位の応答であればプロセスのプライオリティを上げればよく、2-100ミリ秒の範囲であればプロセスをリアルタイミングスケジューリングクラスにすれば可能です。[4] しかしながら、サブミリ秒のスケジューリングは困難です。[5]

サブミリ秒のスケジューリングをLinuxに付加したものが、Victor YodaikenやMichael Barabanovらによって開発されたRTLinuxです。RTLinuxの最新動向は、下記のホームページに掲載されています。また、書籍[6]や雑誌の特集号[7]としても出版されていますので是非ご一読ください。

http://www.rtlinux.org/rtlinux.new/index.html

RTlLinuxは、Linuxカーネル2.0.27対応のVersion 0.5が公開されてから、現在では表2に示すようにLinux 2.2系及び2.3系に対応したカーネルパッチが公開されています。バージョンアップの際に、用意されているAPIの仕様も変化しており、特に、RTLinuxのV.1系統とV.2系統ではAPIの名前自体が変わっています。これは、RTLinuxの仕様がPOSIXのリアルタイム拡張であるPOSIX 1003.1に沿った形で書き直され、POSIX threadモデルが使用されたためです。また、Version 2.0はLinux 2.2対応、つまりSMPに対応しています。次章以降では、RTLinuxのVersion 2.0を対象として使用法を述べていきます。

Table 1 Linuxと対応するRTカーネルパッチ

Linux 2.0.37	Version 1.3
Linux 2.2.13	Version 2.0
Linux 2.3	Version 2.3

図1にRTLinuxの構造を示します。RTLinuxでは、Linuxはリアルタイムプロセスがないときに動作する1つのプロセスとして取り扱われ（優先度が最も低いプロセス）、リアルタイムプロセスがCPUを必要としたときは、横取りするように設計されています。I/Oや割り込み等は、RTカーネルが取り扱い、通常のLinuxのプロセスとの通信は、RT-FIFOやShared Memory、Soft Interruptsを介して行います。

通常のLinuxとRTLinuxにおいて、周期的なプロセスがどの程度の精度で実行処理されるかを比較調査した結果が下記のホームページに掲載されています。Fig.2はその結果から引用したものです。

http://www.zentropix.com/products/support/testdata.html

上から、

- 負荷無し
- init 3レベルのデーモン（httpd、lpd等）がバックグラウンドジョブとして存在
- ハードディスクとネットワークの負荷
- バードディスクの負荷
- ネットワークの負荷
- 浮動小数点を含む計算処理を並行して処理

という条件のもとで処理時間の精度を比較したものです。負荷がない場合は、両者とも分散において同じオーダーの精度ですが、高負荷の場合は約200倍

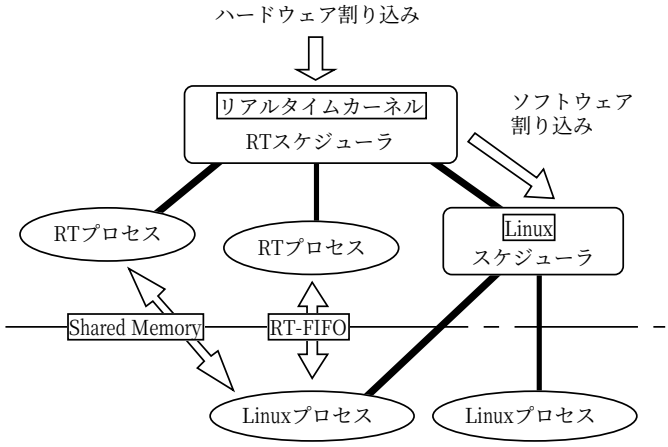


Fig. 1 RTLinuxの構造

もの違いが生じており、RTLinux の方がリアルタイム処理に適していることを示しています。上記のどの負荷条件に対しても、RTLinux では応答時間の分散がほぼ同様な値を示しています。割り込みの応答を調査した結果も先のホームページに掲載されていますので興味のある方はご覧下さい。

ハードウェア割り込みに関しては、最悪のケースで割り込み発生から15マイクロ秒の遅れで実行、周期的なプロセスに対してはスケジューリングされた時刻から25マイクロ秒以内の遅れで実行される仕様になってます。[5]

3.2 RTLinux のインストール

RT-Linuxをインストールするには、linuxのカーネルのソースコードにRT-Linuxのパッチをあてる必要があります。ここでは、Redhat 6.0やSlackware 7.0などの2.2系列のディストリビューションを用いて既にLinux がインストールされているものとします。Linux カーネル及び RTLinux のホームページからカーネルパッチをダウンロードします。

・linux-2.2.13.tar.gz
・rtlinux-2.0.tgz

既にカーネルパッチをあてたファイルもダウンロード可能です。まず、Linux カーネルを解凍します。*

```
$ tar   xvfz    linux-2.2.13.tar.gz
```

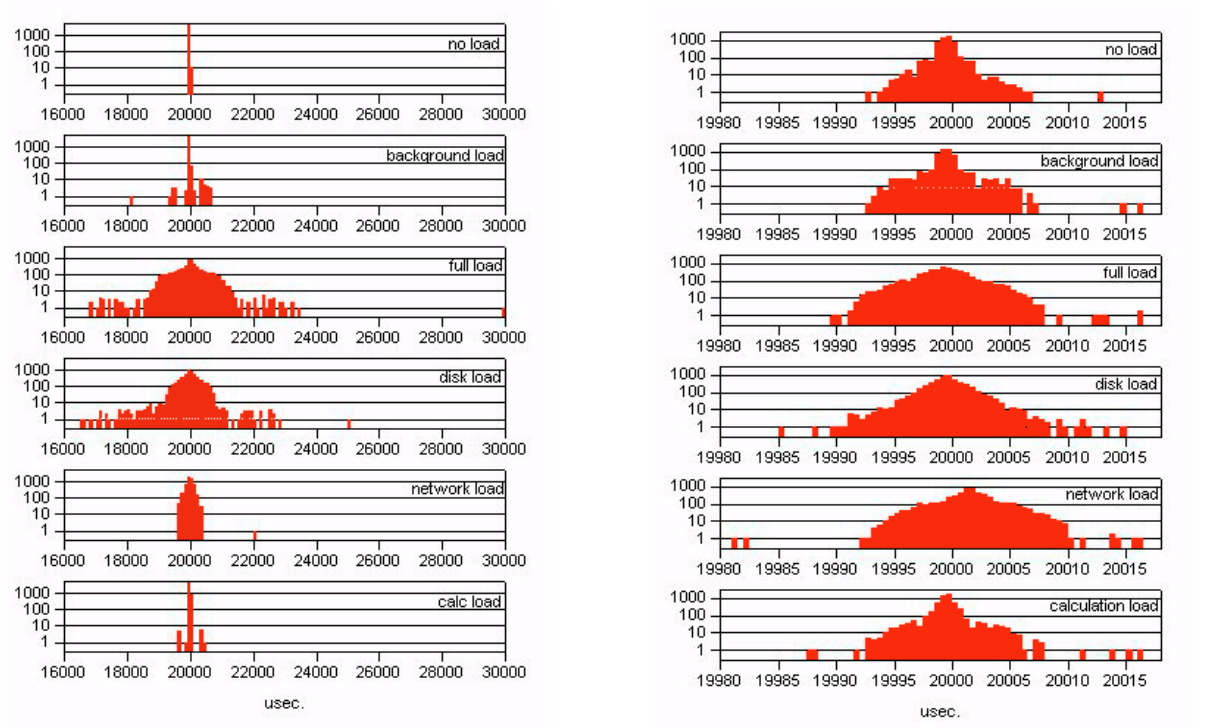
ディレクトリ /usr/src において解凍すると、
/usr/src/linux
が作成されます。同様に rtlinux-2.0.tgz を解凍すると以下に示すファイルが作成されます。

```
$ tar xvfz rtlinux-2.0.tgz
$ ls -F rtlinux-2.0
COPYRIGHT  GettingStarted.txt  README  changelog  rtl/
CREDITS    INSTALL             TODO    kernel_patch
FAQ        INSTALL.phil        UPGRADING
```

ここで展開されたINSTALLにカーネルパッチのあて方についての説明があります。他のテキストファイルについても参考になるのでご一読下さい。カーネルのパッチは、

```
$ cd linux
$ patch -p1 < ../rtlinux-2.0/kernel_patch
```

とします。次に、カーネルのコンパイルオプションを設定し、カーネルをコンパイルします。



(a) *(b)*

Fi.g. 2 通常の Linux (a)と RTLinux (b)の処理時間の比較（参考文献(7)より引用）

* この時に、以前のカーネルを前もって他の名前に変更するのを忘れないようにしましょう。

```
$ make xconfig *
$ make dep; make clean; make zlilo
$ make modules; make modules_install
```

カーネルのコンパイルオプションは以下の設定に注意します。

```
 Processor type and features  --->
       (PPro/6x86MX) Processor family
#CPU の種類の設定
       [*] Symmetric multi-processing support
#SMP の場合は設定
       [*] Hard realtime support
# 必須のオプション

 Loadable module support  --->
       [*] Enable loadable module support
# カーネルモジュールをロード可能にする
       [ ] Set version information on all symbols for
         modules
       [ ] Kernel module loader
```

新しく構築したカーネルを適当なディレクトリにコピーし、再起動することを確認する。次に、RTLinux関係のモジュールを構築する。その際、ディレクトリ rtlinux-2.0 の下に linux カーネルのディレクトリが必要なのでシンボリックリンクを作っておく。

```
$ cd /usr/src/rtlinux-2.0
$ ln -s /usr/src/linux   . /linux
$ cd rtl
$ make; make install
```

正常に終了すると、

```
$ cd /lib/modules/2.2.13-rtl2.0/misc
$ ls -F
        mbuff.o     rt_com.o    rtl_posixio.o  rtl_time.o
        mbuff.o.061  rtl_fifo.o  rtl_sched.o
```

上記のモジュールがインストールされます。起動時にモジュールが自動的に読み込まれるようにするには、ディレクトリ /etc/rc.d/rc.modules というファイルを作成しておくと便利です。

```
# rc.modules
/sbin/modprobe rtl_fifo.o
/sbin/modprobe rtl_sched.o
/sbin/modprobe rtl_posixio.o
/* 続き */
/sbin/modprobe rtl_time.o
/sbin/modprobe rt_com.o
/sbin/modprobe mbuff.o
```

List 2 /etc/rc.d/rc.modules

次に RTLinux の動作確認を行います。カーネルモジュールのロードや確認、削除は以下のコマンドで行います。

- lsmod　現在ロードされているモジュールの一覧を表示する。
- insmod　モジュールをロードする。
- rmmod　モジュールを削除する。

lsmodと入力し、必要なモジュールがロードされているか確認します。

```
$ lsmod
Module      Size        Used by
mbuff       5676        0
rt_com      4120        0  (unused)   **
rtl_fifo    5676        0  (unused)   ***
rtl_posixio 6536        0  [rtl_fifo]
rtl_sched   4280        0  [rtl_posixio]
rtl_time    4344        0  [rtl_sched]
```

RTLinux のサンプルモジュールがあるディレクトリに移動し、動作確認を行う。サンプルプログラムとして、下記のプログラムが用意されています。

```
$ cd /usr/src/rtlinux-2.0/rtl/examples
$ ls -F
README  compat/  fp/  frank/  hello/  measurements/
mmap/  regression/  sound/
```

ここではfrankというサンプルプログラムをロードする。****

```
$ cd /usr/src/rtlinux-2.0/rtl/examples/frank
$ insmod frank_module.o
$ frank_app
FIFO 1: Frank Frank
FIFO 2: Zappa Zappa
FIFO 2: Zappa
FIFO 2: Zappa
```

* make config 或いは make menuconfig でも OK。
** rt_com は v.0.5.3 を使用している。
** * mbuff は v.0.7.1 を使用している。
**** 早期のバージョンから存在したサンプル。

先のように表示されれば、RTLinuxのインストールは無事終了です。モジュールが必要なくなれば、削除しておきましょう。

```
$ rmmod frank_module
```

4. RTLinux の API

ここではカーネルモジュール及びRTLinuxの主なAPIについて説明します。RTLinuxのモジュールも通常のLinuxモジュールの作成と同様に行います。

4.1 カーネルモジュール

モジュールはC 言語で記述し、init_moduleとcleanp_moduleという関数を含む必要があります。

int init_module()

この関数は、カーネルにモジュールがロードされる時にcallされます。オブジェクトの初期化や生成等を記述します。成功した場合は0、失敗した場合は負の値を返します。

void cleanup_module()

モジュールを削除するときにcallされます。使用したメモリの解放など終了処理を記述します。

4.2 スレッドの生成/削除のための API

RTLinux V1.0系では、モジュールにおける各処理をRT_TASK構造体として記述していましたが、V2.0では、POSIXスレッドとして記述します。スレッドとは、共通のアドレス空間を共有するプロセス（群）であり、簡易プロセス（Light-weight process）とも呼ばれています。(4)

pthread_t tid;

スレッドIDのデータ型であり、実装に依存したサイズを持つ構造体です。

pthread_attr_t attr;

pthread属性オブジェクト型です。

Real-Timeスレッドを作成するには、pthread_create関数を使用します。

```
#include <pthread.h>
int  pthread_create(pthread_t  *  thread, pthread_attr_t *
     attr, void * (*start_routine)(void *), void * arg);
```

引数として、pthread_t構造体、スレッドの属性attr、実際に実行される関数のポインタ*start_routine、及び、start_routineの引数であるargを指定します。attrにNULLが指定された場合、デフォルトのスレッド属性が使用されます。また、RTLinuxのスレッドとして使用するには、init_module()においてcallする必要があります。

```
#include <pthread.h>
pthread_t pthread_self(void);
```

スレッド自身のスレッドIDを返す関数です。
スレッドを削除するには、次の関数を使用します。

```
#include <rtl_sched.h>
int pthread_delete_np (pthread_t thread);
```

引数として削除したいスレッドIDをとります。また、関数の末尾の_npはnon-portable又はnon_POSIXを意味しており、他のPOSIX環境においては使用できないことを表している。

4.3 スレッド属性のための API

スレッド生成するには、スレッドに対して属性を与える必要があります。

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

引数としてスレッド属性オブジェクト型をとり、デフォルト値を用いて初期化します。デフォルト値はシステム毎に異なります。

4.4 スケジューリングのための API

RTLinuxのスケジューラは、スレッドの優先順位のみからCPUを割り当てます。優先度が高く、ready状態のスレッドが常に選択されます。スレッドに優先順位等を割り当て、スケジューリングするための関数として以下のAPIが用意されています。

sched_param param;

スケジューリングパラメータ構造体であり、スレッドに与える優先度を指定する。

```
#include <pthread.h>
int  pthread_setschedparam(pthread_t  target_thread,  int
     policy, const struct sched_param *param);
```

引数として、pthread_t構造体、スケジューリングポリシー、sched_paramのポインタをとります。policyには、SCHED_FIFO、SCHED_RRを指定します。SCHED_FIFOの場合は、キューに先に入ったスレッドが先に実行されます。SCHED_RRは、スレッドが

キューの中で循環することを除くとSCHED_FIFOと同じです。リアルタイムスレッドにおいては、sched_param構造体の優先度のみを記述すればよいことになっています。従って、プログラム中では以下のように指定します。

```
struct sched_param p;
p . sched_priority = 1;
pthread_setschedparam (pthread_self(), SCHED_FIFO,
 &p);
```

List 3 優先度の設定

現状ではスケジューリングポリシーの値に意味がありませんが、将来との互換性のために設定しておく方が無難です。優先順位が同じ場合、どのスレッドが選択されるかは定義されていません。

スケジューリングポリシーとパラメータを取得するには次の API を使用します。

```
int   pthread_getschedparam(pthread_t  target_thread,  int
      *policy, struct sched_param *param);
```

次に、スレッドの呼び出し、周期的な呼び出し、一時停止、中止等の API を以下に示します。

```
#include <rtl_sched.h>
int pthread_wakeup_np (pthread_t thread);
```

待ち状態にあるスレッド（pthread_wait_np()により）を起動するための関数。引数はスレッド ID。

```
#include <rtl_sched.h>
int  pthread_make_periodic_np(pthread_t  thread,  hrtime_t
     start_time, hrtime_t period);
```

スレッドを周期的に呼び出すための API です。スレッドの開始時間start_timeと呼び出す周期periodをhrtime_t 型変数として取ります。hrtime_t 型変数は long long 型として定義されており、１秒間はrtl_time.h において以下のようになっています。

```
#define NSECS_PER_SEC 1000000000
```

```
#include <rtl_sched.h>
int pthread_wait_np(void);
```

周期的に呼び出すスレッドにおいて、一回の処理が終了し、CPU を解放する時に使用します。

```
#include <rtl_sched.h>
int pthread_suspend_np (pthread_t thread);
```

pthread_wakeup_np()により起動されるまで、実行を停止します。

4.4 RT-FIFO のための API

リアルタイムスレッドと Linux プロセスの通信には Real-Time FIFO 及び shared memory が使用できます。Real-TimeFIFO はリアルタイムスレッドと Linux プロセスの両者から読み書きできるキューです。通信は一方通行なので、双方向の通信が必要な場合は２つの RF-FIFO を使用する必要があります。

RTLinux のモジュールを make した時に、/dev/rtf0 から /dev/rtf7 迄の８つのデバイスファイルが作成されます。足りない場合は以下のようなシェルファイルを実行するとよいでしょう。

```
/* rtf.sh */
#sh
for i in 8 9 10 11; do mknod /dev/rtf$i c 150 $i; done
```

```
$ sh rtf.sh
```

上記のコマンドにより、major 番号 150、minor 番号が 8-11、キャラクタ型のデバイスファイルが作成されます。

RT-FIFO 関係の API には以下の関数が用意されています。

```
#include <rtl_fifo.h>
int rtf_create(unsigned int fifo, int size);
```

引数の fifo は、RT-FIFO の番号であり minor 番号に対応します。size にはバッファの大きさを入力します。成功した場合は０、失敗した場合はエラーに対応した負の値が返されます。

```
#include <rtl_fifo.h>
int rtf_destroy(unsigned int fifo);
```

fifo で指定された RT-FIFO を削除します。成功した場合は０、失敗した場合は -1 が返されます。通常、cleanup_module において call します。

RT-FIFO に対してデータを読み書きするには、以下の関数を使用します。

```
#include <rtl_fifo.h>
int rtf_get(unsigned int fifo, char * buf, int count);
int rtf_put(unsigned int fifo, char * buf, int count);
```

引数のcountには送受するデータのバイト数を指定します。

更に、RT-FIFO に対してデータが書き込まれた或いは読み込まれたときに起動するhandler関数を設定することができます。

```
#include <rtl_fifo.h>
int rtf_create_handler(unsigned int fifo, int (* handler)(unsigned int fifo));
```

fifo で指定された RT-FIFO に読み書きが成された場合に handler で指定された関数が call されます。引数は FIFO 番号と handler 関数のポインタです。成功した場合は 0、失敗した場合は負の数が返されます。

ユーザ側のプログラムでは、open、write、read、close 等のシステムコールを用いてデバイスファイルのオープン、クローズ、データの送受を行います。

4.5 Clock 関係の API

時間を取得するための API として以下の関数が用意されています。

```
#include <rtl_time.h>
hrtime_t gethrtime(void);
```

システムがbootしてからの経過時間がナノ秒単位で取得できます。スレッドの開始時間の設定などに用います。

```
#include <rtl_time.h>
hrtime_t clock_gethrtime(clockid_t clock);
```

引数として、CLOCK_REALTIME（POSIX のリアルタイムクロック）、CLOCK_8254（通常の x86）、CLOCK_APIC（SMP 用）が指定できます。

4.6 浮動小数点の演算のための API

リアルタイムスレッドでは、デフォルトで浮動小数点の演算を禁止していますが、次の API を用いることにより、スレッド内において数学関数等の浮動小数点演算を行うことが可能です。

```
#include <rtl_sched.h>
int pthread_setfp_np (pthread_t thread, int flag);
```

引数として、スレッドの ID、flag に 1 を指定すると浮動小数点演算を行うことが可能になります。禁止するには flag に 0 を指定します。

4.7 SMP 関係の API

SMP 関係の API として、スレッド属性に対して実行するCPUを指定できる関数が用意されています。*

```
#include <rtl_sched.h>
int pthread_sttr_setcpu_np(pthread_attr_t *attr, int cpu);
```

引数cpu で指定された CPU においてスレッドが実行されるようにスレッド属性を設定します。

```
int pthread_sttr_getcpu_np(pthread_attr_t *attr, int *cpu);
```

スレッドが実行されている CPU の id を取得します。

4.8 割り込みのための API

ロボットシステム開発者にとって、最も興味ある API の１つが割り込み関係の関数だと思います。割り込みには２系統の API が用意されています。１つはハードウェア割り込みのための API であり、もう１つはソフトウェア割り込み、つまり通常の Linux カーネルの割り込みのための API です。ハードウェア割り込みに対しては、割り込み関数の起動時間が保証されていますが、ソフトウェア割り込みは保証されていません。

```
#include <rtl_core.h>
int rtl_request_irq(unsigned int irq, unsigned int (*handler)(unsigned int irq, struct pt_regs *regs));
```

割り込み番号irq に対して、割り込み関数のポインタ *handler を設定します。irq 番号のハードウェア割り込みが発生した場合、関数 hanlder が実行されます。成功した場合は０、失敗した場合は負の数が返されます。

```
#include <rtl_core.h>
int rtl_free_irq(unsigned int irq);
```

割り込み番号 irq を解放します。返り値は、上記 API と同様です。

ハードウェア割り込みを有効にするには、下記の API を実行する必要があります。

```
#include <rtl_core.h>
void rtl_hard_enable_irq(unsigned int irq);
```

引数 irq は許可する irq 番号です。割り込みを繰り返し行うためには、この関数を handler で指定された割り込み関数が実行される毎に call する必要があります。

---

* これらの API は、筆者の手元に SMP のマシンが無いため確認していません。

ソフトウェア割り込みを使用するには次の API を使用します。

```
#include <rtl_core.h>
int rtl_get_soft_irq (void (*handler)(int, void *,  struct
    pt_regs *), const char * devname);
```

デバイスファイル*devnameに対して、割り込み関数 handler を割り当てます。返り値は割り込みベクタが返されます。

```
#include <rtl_core.h>
void rtl_free_soft_irq(unsigned int irq);
```

ソフトウェア割り込みを解放します。
ソフトウェア割り込みを有効にするには、次の関数を呼び出す必要があります。

```
#include <rtl_core.h>
extern void rtl_global_pend_irq(int irq);
```

引数として、rtl_get_soft_irq()で得られた割り込みベクタを指定します。この関数が call されたときにのみ割り込みが発生します。

4.9 RTLinux V1.3 と V2.0 の API の対応

V1.3 の API から V2.0 の API に移行するには、プログラムを書き換える必要があります。新旧の API の対応関係を表３に示します。

V2.0 において、V1.3 の API を使用するには、ファイルrtl_conf.hにおいて、CONFIG_RTL_USE_V1_API オプションを指定することにより可能となります。下記のように通常はコメントアウトされています。

```
/* #define CONFIG_RTL_USE_V1_API 1 */
```

このオプションを指定した場合は、RTLinux のモジュールを make しなおす必要があります。

Table 3 新旧 API の対応

Old API	New API
rt_get_time	gethrtime(3), clock_gettime(3)
rt_task_init	pthread_create(3)
rt_task_delete	pthread_delete_np (3)
rt_task_make_periodic	pthread_make_periodic_np (3)
rt_task_wait	pthread_wait_np(3)
request_RTirq	rtl_request_irq(3), rtl_hard_enable_irq(3)
free_RTirq	rtl_free_irq(3)

5. RTLinux におけるプログラミング

本章では、ディレクトリ rtlinux-2.0/rtl/examples にあるサンプルプログラムを例にとって RTLinux におけるプログラミングについて説明していきます。

5.1 “ Hello World ” プログラム

簡潔な RTLinux プログラムの例が examples/hello ディレクトリに展開されます。その hello.c に少し手を加えたものを List 4 に示します。

最初の３つのインクルードファイルは、必ず必要となります。pthread_t 型のオブジェクト thread を宣言し、thread と処理内容である start_routine が init_module の pthread_create によって関係付けられています。

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread;

void * start_routine(void* arg)
{
        struct sched_param p;
        p . sched_priority = 1;
        pthread_setschedparam (pthread_self(),
SCHED_FIFO, &p);
        pthread_make_periodic_np (pthread_self(),
gethrtime(), NSECS_PER_SEC);

        while (1) {
                pthread_wait_np ();
                rtl_printf("I'm here; my arg is %d\n", *arg);
                rt_printk("I'm here; my arg is %d\n", *arg);
        }
        return 0;
}

int init_module(void) {
        return pthread_create (&thread, NULL,
start_routine, (void*)1);
}

void cleanup_module(void) {
        pthread_delete_np (thread);
}
```

List 4 hello.c

生成された thread は cleanup_module において、rmmod 時に削除されるようになっています。start_routine では、ched_param 構造体の変数 p に対して優先度１を与え、 pthread_setschedparam関数を用いてスレッド属性を thread に設定します。次に、pthread_make_periodic_np関数により現時刻から１秒毎に thread が起動されるようにスケジューリングしています。pthread_wait_np 関数において、次のスケジューリングタイムまでブロックされ、ブロックが解除されると、rtl_printf 関数によりコンソールおいて、下記の結果が１秒毎に表示されます。

```
$ insmod hello
I'm here; my arg is 1
```

rt_printk 関数は、dmesg コマンド及び /var/log/messagesに実行結果を記録するための関数です。使用するには、

```
$ cd /usr/src/rtlinux-2.0/rtl/drivers
$ insmod rt_printk
```

と入力し rt_printk モジュールをロードします。

hello.c をコンパイルするには make と入力します。Make ファイルは、List 5 のようになっています。

```
all: hello.o

include rtl.mk

clean:
        rm -f *.o
```

List 5 Make ファイル

ここで include されている rtl.mk には、List 6 に示されるようにincludeファイルやコンパイルオプション等が記述されています。RTLinux モジュール作成時に自動的に生成されます。

```
#Automatically generated by rtl Makefile
RTL_DIR = /usr/src/rtlinux-2.0/rtl
RTLINUX_DIR = /usr/src/rtlinux-2.0/linux
INCLUDE= -I/usr/src/rtlinux-2.0/linux/include -I/usr/
src/rtlinux-2.0/rtl/include -I/usr/src/rtlinux-2.0/rtl
CFLAGS = -I/usr/src/rtlinux-2.0/linux/include -I/usr/src/
rtlinux-2.0/rtl/include -I/usr/src/rtlinux-2.0/rtl -I/usr/src/
rtlinux-2.0/rtl/include/posix -Wall -Wstrict-prototypes
-O2 -fomit-frame-pointer -D__RTL__ -D__KERNEL__
-DMODULE -pipe -fno-strength-reduce -m486 -
/* 続き */
malign-loops=2 -malign-jumps=2 -malign-functions=2
-DCPU=686
ARCH = i386
CC = gcc
RTL_MODULES=/lib/modules/2.2.13-rtl2.0/misc
```

List 6 rtl.mk ファイル

コンパイルオプションには[8]

・-O2　最適化。

・-fomit-frame-pointer　フレームポインタを必要としない関数では、フレームポインタをレジスタに保存しない。

・-D__RTL__ -D__KERNEL__ -DMODULE

RTLinux において使用するモジュールであることを宣言。例えば、#define __RTL__ とプログラム中に宣言するのと同じ。

・-pipe　コンパイルの中間ファイルの代わりににパイプを使用する。

・-Wall -Wstrict-prototypes　警告レベルの設定

等が設定されている。

5.2 RT-FIFO を用いたプログラム

RT-FIFO を用いたプログラミング例として、examples/frankディレクトリのfranプログラムについて説明します。frankプログラムの全体構造をFig. 3に、frank_module.c を List 7-1~3 に示します。（一部削除）

frankプログラムは、RTLinuxモジュールとして記述されたfrank_module.c及びLinuxの通常のプロセスである frank_app.c から構成されており、両者の通信はRT-FIFOの1~2 を介して行われます。RT-FIFO1及び2はモジュールからLinuxプロセスの送信に使用されています。文字列{“ frank ”, ” Zappa ”}が各スレッドから対応するFIFOに書き込まれます。FIFOにデータが書き込まれると、システムコールselectにより検知され、read 関数により module_app においてデータ

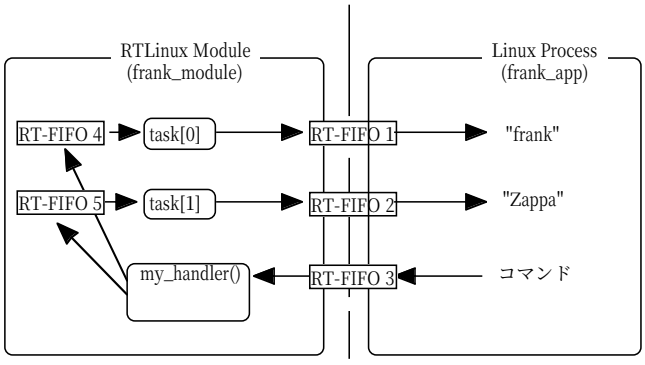


Fig. 3 frank プログラムの構造

を読み込むことが可能となります。一方、frank_app からは RT-FIFO 3 に対してスレッドの状態を変化させるためのコマンドを書き込みます。100 回文字列が送受信されると frank_app は終了します。

次に、frank_module.c を詳しく見ていきます。スレッド task[0],[1]における処理は thread_code 関数に設定されます。各 task は RT-FIFO 4 or 5 にデータが書き込まれると周期的に起動するようスケジューリングされます。RT-FIFO にデータ書き込みがある場合は、if文以下の処理を実行し、無い場合はfrank_app に対して一定周期で文字列を送信します。STOP_TASK コマンドを受信した場合、スレッドは停止状態になります。

関数 my_handler は RT-FIFO 3 に書き込みがあった場合に自動的に起動され、RT-FIFO 4或いは5を介してスレッド task にデータを送信し、スレッド task を起動します。

List 2に示すモジュールの初期化ルーチンでは、関数 rtf_create を用いて５個の RT-FIFO を作成し、リアルタイムスレッドを２個生成しています。また、関数 rtf_create_handler を用いて、RT-FIFO 3 と my_handler を関係付けています。前節のプログラム hello.c では、スレッド属性の設定を実際の処理内容を示した関数内で行いましたが、ここでは関数 init_module において初期化しています。関数 cleanup_module では、作成した RT-FIFO 及びスレッドを削除しています。以上が RTLinux のモジュールにおける RT-FIFO の使用法です。

```
void *thread_code(void *t)
{
int fifo = (int) t;
int taskno = fifo - 1;
struct my_msg_struct msg;
while (1) {
        int ret,err;
        ret = pthread_wait_np();
if ((err = rtf_get (taskno + 4, &msg, sizeof(msg))) == sizeof
(msg)) {
        switch (msg.command) {
                case START_TASK:
pthread_make_periodic_np(pthread_self(), gethrtime(),
msg.period * 1000);break;
                case STOP_TASK:
                pthread_suspend_np(pthread_self());break;
        }
        }
        rtf_put(fifo, data[fifo - 1], 6);
}
}

int my_handler(unsigned int fifo)
{
struct my_msg_struct msg;
int err;
while ((err = rtf_get(COMMAND_FIFO, &msg, sizeof
(msg))) == sizeof(msg)) {
        rtf_put (msg.task + 4, &msg, sizeof(msg));
        pthread_wakeup_np (tasks [msg.task]);
}
        return 0;
}
```

List 7-1 frank_modules.c のスレッドの処理内容

```
int init_module(void)
{
int c[5];
pthread_attr_t attr;
struct sched_param sched_param;
int ret;
c[0] = rtf_create(1, 4000);
   ・・・
c[4] = rtf_create(5, 100);/* input control channel */
pthread_attr_init (&attr);
sched_param.sched_priority = 4;
pthread_attr_setschedparam (&attr, &sched_param);
pthread_create (&tasks[0],  &attr, thread_code, (void *)1);

pthread_attr_init (&attr);
   ・・・
pthread_create (&tasks[1],  &attr, thread_code, (void *)2);
rtf_create_handler(3, &my_handler);
return 0;
}

void cleanup_module(void)
{
rtf_destroy(1);
   ・・・
rtf_destroy(5);
pthread_delete_np (tasks[0]);
pthread_delete_np (tasks[1]);
}
```

List 7-2 frank_module.c の初期化ルーチン

では、対応するユーザ側のプログラム、つまり通常の Linux プログラム frank_app.c（List 7-3、一部省略）について見てみましょう。

(A)最初にシステムコールを利用するためのヘッダファイルを include しています。

(B)fd_set 変数として rfds を宣言しています。この変数に、RT-FIFO にあたる部分のビットを設定します。timeval 構造体として tv を宣言し、tv にある時間を書き込むことにより、設定した時間毎に RT-FIFO を調べます。

(C) open 関数を使って、RT-FIFO の 1 及び 2 番、/dev/rtf1 ,rtf2 を読み込みモードでオープンしています。オープンに成功した場合、ファイルディスクリプタの値が fd0、fd1 に返ります。また、/dev/rtf3 は書き込みモードでオープンしています。

(D) RTLinuxモジュールに対してスレッドの起動コマンドを送信する。

(E) fd_set 変数の rfds を FD_ZERO 関数を用いて初期化しています。次に、FD_SET関数を用い、先ほどオープンしたファイルディスクリプタ fd0、fd1 に該当する部分のビットを立てています。

(F) tv に時間をセットしています。この例では、一秒をセットしています。次に select関数を用い、該当するファイルディスクリプタにデータが入っているかどうか調べています。tv に一秒がセットされているので、select関数により、一秒間だけデータが入るまで待ち状態になりますファイルディスクリプタにデータがある場合は、１が返ります。データがない場合は 0 が返されます。ここで、FD_SETSIZEは1024という数字が入っています。

(G) RT-FIFO に該当するディスクリプタ fd0 と fd1 にデータが入力されているかどうか、FD_ISSET関数を用いて調べます。fd0 又は fd1 にデータが入力されている場合、read 関数を用いてデータを読み込みます。read 関数の引数は、読み込み先のディスクリプタ、データを読み込むバッファ、データのサイズとなっています。その結果をprintf関数により表示します。

```
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>                                    (A)
#include <sys/ioctl.h>
#include <rtl_fifo.h>
#include <rtl_time.h>
char buf[BUFSIZE];

int main(){
fd_set rfds;                                           (B)
struct timeval tv;
int retval;
int fd0, fd1, ctl;
int n,i;
struct my_msg_struct msg;
fd0 = open("/dev/rtf1", O_RDONLY);
fd1 = open("/dev/rtf2", O_RDONLY);                     (C)
ctl = open("/dev/rtf3", O_WRONLY);
/* now start the tasks */
msg.command = START_TASK;msg.task = 0;
msg.period = 500000;
write(ctl, &msg, sizeof(msg)) ;                        (D)
msg.task = 1;msg.period = 200000;
write(ctl, &msg, sizeof(msg)) ;
```

```
for (i = 0; i < 100; i++) {
        FD_ZERO(&rfds);
        FD_SET(fd0, &rfds);                            (E)
        FD_SET(fd1, &rfds);
        tv.tv_sec = 1;
        tv.tv_usec = 0;
        retval = select(FD_SETSIZE, &rfds, NULL,
NULL, &tv);                                            (F)
        if (retval > 0) {
                if (FD_ISSET(fd0, &rfds)) {
                n = read(fd0, buf, BUFSIZE - 1);
                printf("FIFO 1: %s\n", buf);
                }                                      (G)
                if (FD_ISSET(fd1, &rfds)) {
                n = read(fd1, buf, BUFSIZE - 1);
                printf("FIFO 2: %s\n", buf);
                        }
                }
        }
/* stop the tasks */
msg.command = STOP_TASK; msg.task = 0;
write(ctl, &msg, sizeof(msg)) ;
msg.task = 1;                                          (H)
write(ctl, &msg, sizeof(msg)) ;
return 0;
}
```

List 7-3 ユーザプログラム frank_app.c

(H) RTLinuxモジュールのスレッドに対して停止コマンドを送るために、ディスクリプタ ctl に write 関数を用いて書き込みます。

frank プログラムを実行については、3 章で述べた通りです。

5.3 浮動小数点演算プログラム

関数 pthread_setfp_np を使用して、リアルタイムスレッドで浮動小数点演算を行うプログラムを List 8-1（rt_process.c）に示します。

```
#include <rtl.h>
#include <time.h>
#include <asm/io.h>
#include <pthread.h>
#include <math.h>
pthread_t mytask;

void *fun (void *t) {
        int i = 0;
        double f, f2, f3;
        double x=0;
        while (1) {
                f = 10.0 /(1.0 + exp(-x));
//              f=10.0*sin(x);
                f2=10.0*log(x+1.0);
                f3=sqrt(x);
                x+=1.0;
                if(x>10) x=0;
                pthread_wait_np();
        }
}

int init_module (void) {
struct sched_param p;
hrtime_t now = gethrtime();
pthread_create (&mytask, NULL, fun, (void *) 1);
pthread_make_periodic_np (mytask, now + 2 *
NSECS_PER_SEC, NSECS_PER_SEC);
pthread_setfp_np (mytask, 1);
p . sched_priority = 1;
pthread_setschedparam (mytask, SCHED_FIFO, &p);
return 0;
}

void cleanup_module (void) {
pthread_delete_np (mytask);
}
```

List 8-1 浮動小数点演算プログラム

数学関数を使用するために<math.h>をインクルードしています。関数 fun では、exp()や log()、sqrt()等の浮動小数点演算をしており、そのためには init_moduleにおいてpthread_setfp_np (mytask, 1)とし、スレッド mytask に浮動小数点演算を許可しています。次に、Make ファイルを List8-2 に示します。

```
rt_process.o: rt_process.c
        $(CC) ${INCLUDE} ${CFLAGS} -c -o
rt_process_tmp.o rt_process.c
        ld -r -static rt_process_tmp.o -o rt_process.o -L/usr/
lib -lm -lc
        rm -f rt_process_tmp.o
```

List 8-2 浮動小数点演算用の Make ファイル

先に rt_process.c をコンパイルし、数学ライブラリと C ライブラリをスタティックリンクしています。sin 関数や cos 関数は、-lm をリンクするだけで使用できますが、exp 関数等は -lc をリンクする必要があることにご注意下さい。

5.4 Shared memory を使用するプログラム

リアルタイムスレッドとユーザプログラム間で通信する方法として、先に RT-FIFO を使用する手法について述べましたが、ここではshared memoryを用いて通信する手法を紹介します。shared memoryを用いる手法には、デバイスファイルとして /dev/mem を用いる方法と、/dev/mbuff を用いる方法が存在します。

(1) /dev/mem を用いる方法

カーネルによって確保されないアドレス空間を用います。そのためには、カーネルの boot 時に append により mem パラメータを与える必要があります。例えば、LILO を用いて boot しており、メモリ 128M 搭載したマシンにおいて 4M の shared memory を得るには、/etc/lilo.conf は List 9-1 のようになります。

```
boot = /dev/hda3
timeout = 10
prompt
  vga = normal
  append = "mem=124M"
  read-only
map=/System.map
image = /bzImage
  label = linux
  root = /dev/hda3
```

List 9-1 lilo.conf の設定

shared memory として設定できるのは、ページングの関係から4Mバイトとなっており、4M以上を設定するとエラーを発生します。/dev/mem ファイルを用いたプログラムをList 9-2に示します。このプログラムは examples/mmap/rtp.c であり、リアルタイムスレッド及びユーザプログラムとしてコンパイルされます。

まず、/dev/mem デバイスファイルをopen関数を用いてオープンします。次に、mmap関数によりメモリ空間の確保を行います。最初の引数は、インデックスとして開始メモリを与えるか、通常は0とします。length は確保するメモリのバイト数です。

```
#include <rtl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
int *start;
int address = 0xA0000;
size_t length = 1000;

int init_module(void ){
        int i;
        int fd = open("/dev/mem", O_RDWR);
start=(int *)mmap(0,length, PROT_READ |
PROT_WRITE, MAP_SHARED, fd,address);
        for (i = 0; i < 100; i++) {
                rtl_printf ("%x ", start[i]);
        }
        rtl_printf ("\n");
        return 0;
}

void cleanup_module(void ){
      munmap(start, length);
}

#ifndef MODULE
int main(void)
{
        if (init_module() == 0) {
                cleanup_module();
        }
        return 0;
}
#endif
```

List 9-2 shared memory プログラム -1

PROT_READ と PROT_WRITE はメモリ保護の状態を表し、ここでは書き込みと読み込みに許可を与えています。MAP_SHAREDはマップの種類であり、全てのプロセスからアクセスできることを意味しています。MAP_PRIVATE を指定すると、オープンしたプロセスのみがアクセス可能となります。fd は open 関数から返り値、ファイルディスクリプタであり、addressはメモリ空間からのオフセットとなっています。

RTLinux V1.3 での使用法を List9-3 に示します。

```
/*   shmem.h : header file for shared memory       */
#define SH_MEM_BASE_ADDR (124*0x100000)
#define MAP_FAILED       (-1)

typedef struct {
        char variable1[10];
        int variable2;
        float variable3;
} RUN_DATA;

RUN_DATA* get_sh_mem_ptr(void);
-----------------------------------------------
/* モジュールプログラム rt_shmem.c */
#include "shmem.h"
RT_TASK shmemtask;
RUN_DATA *ptr;
void shmemwrite(int t){
        static int i=0;
        float f=0.0;
        sprintf(ptr->variable1,"RT_TASK\n");
        ptr->variable2 = 0;
        while(1){
                ptr->variable2 = (int)f;
                f = ptr->variable3;
                rt_task_wait();
        }
}

int init_module(void){

        RTIME now=rt_get_time();
        ptr = (RUN_DATA*) SH_MEM_BASE_ADDR;
        rt_task_init(&shmemtask,shmemwrite,0,3000,4);
        rt_task_use_fp(&shmemtask,1);

/* 続く */
```

```
        rt_task_make_periodic
        (&shmemtask,now,0.1*RT_TICKS_PER_SEC);
        return 0;
}

void cleanup_module(void){
        rt_task_delete(&shmemtask);
}
-----------------------------------------
/* ユーザプログラム */
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
#include "shmem.h"
RUN_DATA *ptr;
RUN_DATA *get_sh_mem_ptr(void){
        int fd;
        RUN_DATA *ptr;
        fd=open("/dev/mem", O_RDWR)) ;
        ptr = (RUN_DATA*) mmap(0,sizeof
(RUN_DATA),PROT_READ|PROT_WRITE,

MAP_FILE|MAP_SHARED,fd,SH_MEM_BASE_ADDR);
        close(fd);
        return(ptr);
}

int main(void)
{
        char tmp1[10];
        int tmp2;
        float tmp3, f=0.0;
        ptr = (RUN_DATA *)get_sh_mem_ptr();
        while(1){
                sscanf(ptr->variable1,"%s",tmp1);
                tmp2 = ptr->variable2;
                ptr->variable3 = f;
                printf("shared memory data= %s, %d,
%f\n",tmp1,tmp2,f);
                f+= 10.0;
                sleep(1);
        }
}
```

List 9-3 shared memory プログラム -2

(2) /dev/mbuff を用いる方法

/dev/mbuff デバイスファイルを用いて shared memory を介して通信するには、mbuff.o モジュールをロードする必要があります。mbuff は Tomasz Motylewski によって開発されたモジュールであり、/dev/mem のようなメモリ空間の制限がありません。ディレクトリ /usr/src/rtlinux-2.0/rtl/drivers にインストールされており、最新版としてwebにおいてv0.7.1のモジュールが公開されています。カーネルモジュールとして使用する場合のプログラムをList 10-1、ユーザプログラムを List 10-2 に示す。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <mbuff.h>
#define sharename "shm1"

volatile char* ptr;
volatile char* start;
int num_space = 1000;        /* 1000 bytes */

int init_module ( void ) {
  int i=0;
  ptr=mbuff_alloc(sharename,num_space);
  start = ptr;
  for(i=0;i<1000;i++){
        *ptr=(char)i;
        ptr++;
  }
  return 0;
}

void cleanup_module( void ){
  mbuff_free(sharename,ptr);
}
```

List 10-1 mbuff を用いたモジュール作成

下記の関数を用いてメモリー空間を確保する。

```
#include <mbuff.h>
volatile char* mbuff_alloc(char* name, unsignedshort num);
```

引数nameは確保するメモリー空間へのインデックスであり、num バイト数のメモリを確保する。返り値は、確保されたメモリ空間のポインタとなっている。

モジュールを削除する際には、

```
#include <mbuff.h>
void mbuff_free(char* name);
```

を用いてメモリを解放する必要がある。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <mbuff.h>
#define sharename "shm1"
volatile char* ptr;
int num_space=1000;
/* main routine */
int main (void ) {
  int i=0;
  ptr=mbuff_alloc(sharename,num_space);

  for(i=0;i<1000;i++){
        printf("ptr[%d]=%x ",i,ptr[i]);
        if(i%10==0) printf("\n");
  }
  mbuff_free(sharename,ptr);
}
```

List 10-2 mbuff を用いたユーザプログラム作成

ユーザプログラムにおいても、mbuff_alloc関数及び、mbuff_free関数を使用する。共通のshared memoryにアクセスするには、引数であるメモリー空間へのインデックスとしてモジュールと同じ名前を指定する必要がある。また、/dev/mbuffデバイスファイルのmajor番号は10、minor番号は254であり、RTLinuxモジュール構築の際に自動的に作成される。

5.5 シリアルポート通信プログラム

シリアルポート通信用のモジュールとして、Jens Michaelsen, Jochen Kらによって、rt_com.oが開発されている。最新版はv 0.5.3である。インストールし使用可能とするには、

```
$ cd rt_com-0.5.3
$ make; make install
$insmod rt_com
```

と入力し、モジュールをロードする。

用意されているAPIを以下に示す。

```
#include <rt_com.h>
void rt_com_setup( unsigned int com, unsigned baud, unsigned parity,
                unsigned stopbits, unsigned wordlength );
```

引数のcomは、rt_com.cにおいてrt_com_tableとして定義してある変数である。例えば、com=0の場合、ポートアドレス=0x3f8、irq=4と設定される。baudは通信速度、parityはパリティチェック、stopbitsはストップビット、wordlengthは通信バイト長を意味する。これらの引数の値は、rt_com.hに定義されている。

```
#include <rt_com.h>
void rt_com_write(unsigned int com, char *buf, int cnt );
```

引数bufは送信データのポインタであり、cntバイト数を書き込む。

```
#include <rt_com.h>
void rt_com_read(unsigned int com, char *buf, int cnt );
```

引数bufは受信データのポインタであり、cntバイト数を読み込む。

rt_comモジュールを展開すると、rt_com-0.5.3/testが生成される。ディレクトリtestにサンプルプログラムがあるので実行して動作を確認する。List11-1にhelloworld.cを示す。

```
#include <rt_com.h>
static int com=0;
int init_module(void){
char hello[] = "Hello World";
if( 0 > rt_com_setup( com, 9600,
                RT_COM_PARITY_NONE, 1, 8 ) )
return( -1 );
rt_com_write( com, hello, sizeof( hello ) );
printk( KERN_INFO "rt_com test: >>%s<< sent.\n", hello );
return( 0 );
}
/** free all allocated resources */
void cleanup_module(void)
{
   rt_com_setup( com, -1, 0, 0, 0 );
   printk( KERN_INFO "rt_com test: finished\n");
}
```

List 11-1 rt-com を用いたシリアル通信

参考のため、筆者らがMS-DOSプログラムからLinux用に移植したシリアル通信プログラムをList 11-2に示す。

```
/* serial.h */
#define COM1_BASE_ADR    0x3F8 /*** COM1 ***/
#define COM2_BASE_ADR    0x2F8 /*** COM2 ***/
#define COM3_BASE_ADR    0x3E8 /*** COM3 ***/
#define COM4_BASE_ADR    0x2E8 /*** COM4 ***/

#define IER              1 /*** Interrupt Enable Register ***/
#define FCR              2 /*** FIFO Control Register ***/
#define LCR              3 /*** Line Control Register ***/
#define MCR              4 /*** Modem Control Register ***/
#define LSR              5 /*** Line Status Register ***/

#define    IMR    0x21 /*** Interrupt Mask Register */
#define ICR    0x20 /*Interrupt Control Register*/
#define EOI    0x20 /*** End Of Interrupt ***/

#define BUF_MAX            256
#define BAUD_300  0x0180
#define BAUD_1200          0x0060
#define BAUD_2400          0x0030
#define BAUD_4800          0x0018
#define BAUD_9600          0x000C
#define BAUD_19200         0x0006
#define BAUD_38400         0x0003

#define FIFO_TRIG_1    0x01 /* FIFO trigger = 1 byte */
#define FIFO_TRIG_4    0x41 /* trigger = 4 byte */
#define FIFO_TRIG_8    0x81 /* trigger = 8 byte */
#define FIFO_TRIG_14    0xc1 /* trigger = 14 byte */
#define FIFO_CLEAR         0x07 /*** FIFO clear ***/
-------------------------------------------------------------
/* serial.c */
#include <asm/io.h>
#include "serial.h"
static unsigned short int port, baudrate;
void my_rs_init(int com, int baud)
{
switch(com){
        case 1:
        port = COM1_BASE_ADR;break;
        case 2:
        port = COM2_BASE_ADR;break;
        case 3:
        port = COM3_BASE_ADR;break;
        case 4:
        port = COM4_BASE_ADR;break;
        default:
        }
switch(baud){
        case 9600:
                baudrate = BAUD_9600;break;
        case 4800:
                baudrate = BAUD_4800;break;
        case 2400:
                baudrate = BAUD_2400;break;
        default:
        }
outb(0x0,port+IER); /* interrupt disable */
outb(0x80,port+LCR); /* Divisor Latch Access Bit = 1*/
outw(baudrate,port); /*** Divisor set ***/
outb(0x03,port+LCR); /* DLAB=0,data bit=8,stop
bit=1,no parity */
outb(FIFO_TRIG_8,port+FCR);
outb(0x0f,port+IER); /* Interrupt enable (recieve) */
}

int rs_send(char* sdata, int n){
int i=0;
for(i=0;i<n;i++){
        while(1){
        if((inb_p(port+LSR) & 0x60)==0x60) break;
        }
        outb(sdata[i],port);
        }
return(1);
}

int rs_receive(char* rdata){
        if((inb(port+LSR) & 0x01)==0x01) {
        *rdata = inb(port);
        return(1);
        } else {
        *rdata=0;
        return(-1);
        }
}
```

List 11-2 MS-DOSから移植したシリアル通信プログラム

5.6 割り込みプログラミング

ハードウェア割り込みを使用したプログラムをList 12-1 に示す。割り込みが発生した場合の処理関数irq_handler と割り込み番号の関係を init_module のrtl_request_handler により設定する。割り込みを可能にするには、rtl_hard_enable_irq 関数を割り込みが発生する毎に call する必要がある。

ソフトウェア割り込みを使用したプログラムをList12-2に示す。割り込み関数はList 12-1と同じである。rtl_get_soft_irq 関数により、デバイスファイル /dev/ttyS0 に割り込みが発生した場合、irq_handler が起動するように設定されている。ソフトウェア割り込みは、rtl_global_pend_irq 関数が呼ばれたときにトリガーがかかるので、周期的に呼び出して常時監視する必要がある。

```
#include <rtl.h>
#include <rtl_core.h>
#include <time.h>
#include <pthread.h>
#include <asm/io.h>
#include "serial.h"

pthread_t servotask;
pthread_t readtask;
static char      getbuf[DATA_SIZE];

unsigned int *irq_handler(unsigned int irq, struct pt_regs
*regs){
        rs_receive(getbuf);
        rtl_printf("get:%s\n",getbuf);
        rtl_hard_enable_irq(4);
        return(0);
}

int init_module(void){
        struct sched_param p;
        hrtime_t now=gethrtime();
        my_rs_init(1,9600);
        rtl_request_irq(4,irq_handler);
        rtl_hard_enable_irq(4);
        return 0;
}

void cleanup_module(void){
        rtl_free_irq(4);
}
```

List 12-1 ハードウェア割り込みプログラム

```
void * read(void* t){
        while(1){
                rtl_global_pend_irq(softirq);
                pthread_wait_np();
        }
}

int init_module(void){
        struct sched_param p;
        hrtime_t now=gethrtime();
        my_rs_init(1,9600);
        pthread_create(&readtask,NULL,read,(void*) 0);
        pthread_make_periodic_np(readtask,
                now,NSECS_PER_SEC*0.0001);
        p.sched_priority = 20;
        pthread_setschedparam(readtask,
                SCHED_FIFO,&p);
        softirq=rtl_get_soft_irq(irq_handler,"/dev/ttyS0");
        return 0;
}

void cleanup_module(void){
        pthread_delete_np(readtask);
        rtl_free_soft_irq(softirq);
}
```

List 12-2 ソフトウェア割り込み

5.7 C++ によるモジュールの記述

カーネルモジュールは、基本的に C 言語により記述するように設計されているが、C++ の new 演算子やコンストラクタ、デストラクタを記述することでC++を使用することが可能となる。プログラムをList 13-1に示す。カーネルモジュールを作成する際に、通常インクルードされるヘッダファイルは、C 言語を前提としているので、

```
extern "C"    {}
```

を用いてＣリンケージであることを明示する必要がある。更に、C++ のみで使用される変数、実行時型情報（RTTI）や CPU 例外処理の関数は、自分で与えるか、Makeファイルにおいて、次のコンパイルオプションを設定する必要がある。

```
-fno-exceptions -fno-rtti
```

new 演算子、delete 演算子の代わりとして、

```
        void* kmalloc( size_t size, priority);
        void kfree( void* ptr);
を用い、メモリ空間の確保を行う。引数の priority には通常 GFP_KERNEL を設定する。*

extern "C" {
#define NULL 0
#define new _new
#include <linux/module.h>
#include <linux/kernel.h>
#include <rtl.h>
#include <rtl_time.h>
#include <pthread.h>
#include <math.h>
#undef new

void * kmalloc(unsigned size, int prio);
//void   kfree(void * p);

pthread_t thread;

void * start_routine(void* arg);
int  init_module();
void cleanup_module();
void perror(char *wrt) { printk(KERN_ERR ": %s\n",wrt); }
//void __throw() { panic("C++ Exception thrown!"); }
//void __rtti_user(void) {}
//void __rtti_si(void) {}
};

class matrix{
private:
        int row,col;
public:
        matrix(int, int);
        ~matrix(void);
        double *ptr;
};

void terminate (void) {;}
void * operator new (unsigned size) { return kmalloc
(size,GFP_KERNEL); }
void * operator new[](unsigned size, unsigned nb)
         { return kmalloc(size*nb,GFP_KERNEL); }
void operator delete (void * p) { kfree(p); }
void operator delete[](void *p) { kfree(p); }
```

* 筆者の環境では、signal.h において数カ所 break 文を追加する必要があった。

```
matrix::matrix(int r, int c) {
        ptr=new double(r*c);
        for(int i=0;i<r;i++){
                for(int j=0;j<c;j++){
                        ptr[i*c+j]=(double)(i+j);
                        rtl_printf("%d ",(int)ptr[i*c+j]);
                }
        }
}

matrix::~matrix(){}
matrix *m;

void * start_routine(void* arg){
        int i=0,j=0;
        while (1) {
                pthread_wait_np ();
                for(i=0;i<3;i++){
                        for(j=0;j<3;j++)
                        rtl_printf("[%d,%d]= %d ¦ ",i,j,(int)
m->ptr[i*3+j]);
                        rtl_printf("\n");
                }
        }
        return 0;
}

int init_module(){
        struct sched_param p;
        p . sched_priority = 1;
        m = new matrix(3,3);
        pthread_create (&thread, NULL,
                start_routine, (void*)0);
        pthread_make_periodic_np (thread,
        2*NSECS_PER_SEC,NSECS_PER_SEC);
        pthread_setfp_np(thread,1);
        pthread_setschedparam (thread,
                 SCHED_FIFO, &p);
        return 0;
}

void cleanup_module(){
        pthread_delete_np (thread);
        kfree(m);
}
```

List 13 C++ を用いたモジュールプログラム

5.8 Java との連携

ユーザプログラムをJavaで開発し、リアルタイム処理をカーネルモジュールにおいて行うには、ネイティブメソッドを作成する必要がある。例えば、List 14-1 に示すようにJava でオブジェクト定義（SharedInterface.java）したとする。

```
public class SharedInterface {
 static {
  System.loadLibrary("share");
 }
 public SharedInterface()
 {
  getPointer();
 }
 public synchronized native void getPointer();
}
```

List 13-1 SharedInterface.java

次に、C 言語により実際の処理を記述する。ファイル名を SharedNative.c とし List 13-2 に示す。

```
#include "SharedInterface.h"
RUN_DATA *ptr;

void  SharedInterface_getPointer(struct HSharedInterface
*this)
{
 ptr = (RUN_DATA *)get_sh_mem_ptr();
}
```

List 13-2 SharedNative.c

以下のコマンドにより、ダイナミックリンクライブラリを作成することができる。

```
$ javac SharedInterface.java
        SharedInterface.class が作成される
$ javah SharedInterface
        ヘッダファイルSharedInterface.hが作成される
$ javah -stubs SharedInterface
        SharedInterface.c が作成される
$ gcc -c -I /usr/local/jdk/include -I /usr/local/jdk/include/
genunix SharedNative.c -o Sharednative.o
$ gcc SharedInterface.c -o SharedInterface.o
$ gcc -shared SharedInterface.o SharedNative.o -o
libshare.so
```

5.9 モジュールへの引数の渡し方

MODULE_PARM 関数を使用すると、カーネルモジュールをロードする際に、main(int argc, **argv)のように引数を渡すことが可能である。プログラムを List 14 に示す。

```
char num_b=0;
short int num_h=0;
int num_i=0;
long int num_l=0;
char *moji;

MODULE_PARM(num_b,"b");
MODULE_PARM(num_h,"h");
MODULE_PARM(num_i,"i");
MODULE_PARM(num_l,"l");
MODULE_PARM(moji,"s");
・・・
```

List 14 モジュールへの引数

このモジュールは、ロード時の引数として、

```
$ insmod rt_process num_b=1 num_h=2 num_i=3
num_l=4 moji= ” name ”
```

とすると各変数に値が渡される。MODULE_PARM 関数の２つめの引数は、読み込む際のデータ型を表しており、b はバイト、h は short、i は int、l は long、s は文字列のポインタを入力値としてとる。

6 ロボットへの適用

本章では、RTLinux を実際にロボットへ適用した例を紹介する。

6.1 四足歩行ロボットへの適用

使用した歩行ロボットの概観をFig.4に示す。機構部は各脚３自由度を持つ４足歩行ロボットTITAN VIII 、コンピュータシステムは DOS/V 互換の CPU ボードとDAボード、カウンタボードをバックプレーンに取り付けて構築している。前方にビジョンチップ[9]を搭載している。

視覚センサとして用いたビジョンチップは、網膜の視覚情報処理機能を模倣して開発されたアナログCMOS集積回路である。このチップはラプラシアン-ガウシアンフィルタの機能を有するため、輪郭強調された２次元画像を得ることができる。さらに、感度調節機能が内蔵されており、自然照明下において

も使用できるという移動ロボットの視覚システムに適した特徴を有している。

全体のハードウェア構成をFig.5に示す。ロボットに搭載されたアクチュエータの制御はD/Aボードとシリアルポートを使用している。D/Aボードを介し、DCモータドライバに対しロボットへの指令電圧が送信される。シリアルポートに対しては、カメラのpan、tiltを制御するためのサーボモータドライバへ角度指令値が送信される。

ビジョンチップからの視覚情報は、FPGAにおいて2値化された後、PIOを介してコンピュータに入力される。視覚情報の一例をFig.6に示す。

ソフトウェア構成をFig.7に構成とした。直進、回頭等の行動決定は、ユーザプログラムとして開発し、脚の動作など実時間で制御する必要がある処理は、Linuxモジュールとして開発している。視覚画像の取得は、Fig. 8に示すように割り込みを介して行っている。2値化作業において、エッジの部分ををHigh、その他ではLowとして出力されるので、エッジ出力をDAボードの割り込み信号として用いている。割り込み関数では、エッジのある画素の位置を記録する。

本セミナーにおいて、視覚情報に基づいた行動決定、対象物の追跡等を実施する予定である。

Fig. 4 四足歩行ロボットの外観

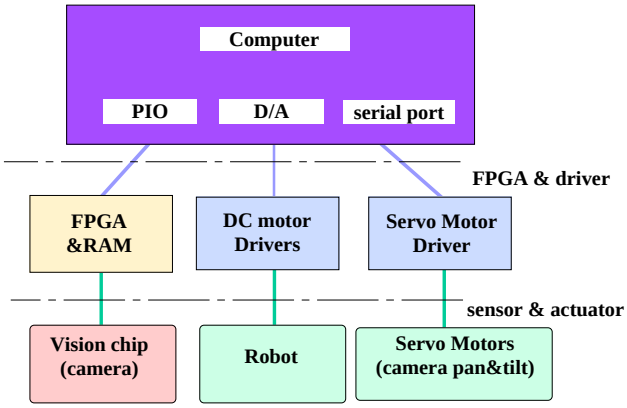


Fig. 5 ハードウェア構成

背景(黒)　球(白) を見たビジョンチップ

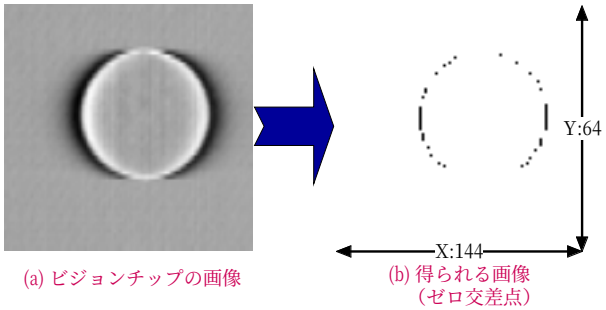


Fig. 6 視覚情報

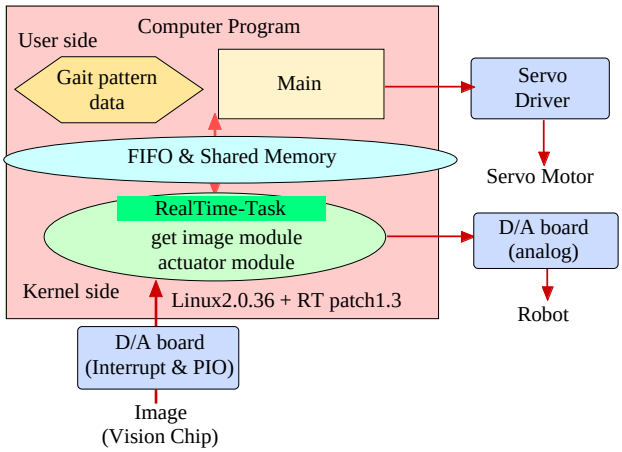


Fig. 7 ソフトウェア構成

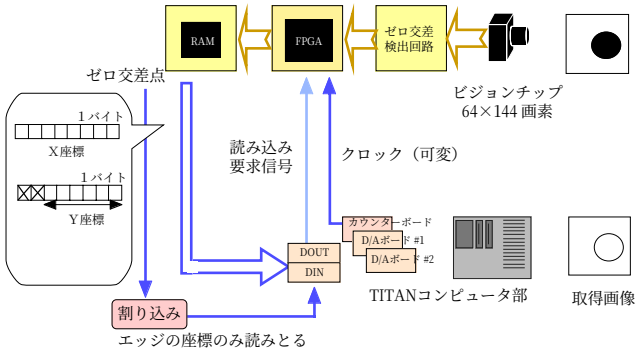


Fig. 8 割り込みによる視覚情報の取得

6.2 ノートパソコンを対象としたシステム開発例

次に、ノートパソコンを車輪型の移動ロボット（Fig.9 参照）の搭載コンピュータとしてロボットシステムを開発した例を示す。

ハードウェアの構成は、Fig.10に示すように、アクチュエータとして 2 個の DC モータ及びカメラの Pan、Tilt 用に 2 個のサーボモータ、センサとして姿勢センサ（TCM2:方位角、ピッチ角、ロール角等）、GPS、2 台の CCD カメラとなっている。

モータドライバはシリアルポート接続、指示値に応じたPWM信号をモータへ送信する。姿勢センサ及び GPS は PC カードを介してシリアルポートに接続されている。パラレルポートには前後、左右、前斜め45度前方の６つの超音波センサが取り付けられており、20cm-300cmの範囲の距離が取得可能となっている。

ソフトウェア構成を Fig.11 に示す。前節のケースと同様、ロボットの姿勢、位置情報、アクチュエータの制御等の実時間で制御する必要があるプログラムはカーネルの一部として組み込み、ユーザモードのプログラムとは Shared-Memory 及び RT-FIFO を介して通信する構成とした。

行動決定プログラムはユーザプログラムとして開発されており、ロボットのミッション開始、目的地の更新、超音波センサからの障害物情報等の外部イベントによりロボットの状態が推移して行動を決定する。また、インターネット上のコンピュータで実行される GUI プログラムに対し、ロボットの現在位置、目的位置、状況等が PPP 及びソケット通信を介して伝送され、外部から監視できる構成となっている。GUI は OpenGL 及び Java を用いて開発している。

Fig. 9 移動ロボットの外観

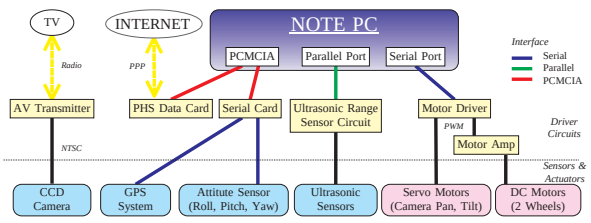


Fig. 10 移動ロボットのハードウェア構成

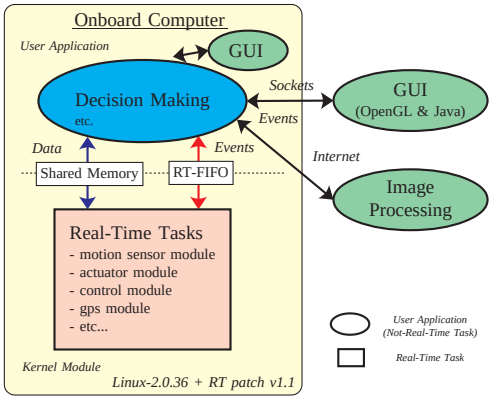


Fig. 11 移動ロボットのソフトウェア構成

7. 終わりに

本原稿では、RTLinux インストールから実際にロボットに適用する方法について述べた。筆者は、RTLinux の DOS プログラミングに近い感覚で I/O プログラミングが行え、UNIX の特徴であるネットワークへアクセスが容易であることに魅力を感じる。

問題点として、root としてカーネルモジュールのプログラミングをしている以上、システムクラッシュが上げられる。実際に筆者も何度もシステムクラッシュを体験している。複数の人数で作業する場合や、共同でプログラミング開発を行う場合など、安全対策が必要となることを念頭において頂きたい。上記の環境では ART-Linux の方が適していると考えられる。

RTLinux の強みは、利用者が多く、ユーティリティープログラム等の開発やシステムのデバッグ等、ネットワークにつながった人々が鋭意行っていることにあるといえる。

参考文献

(1) 藤井他：自己訓練と学習に基づく海中ロボットの運動制御、日本ロボット学会誌 Vol. 13 No. 7 pp. 1006-1019、1995/7

(2) http://www.rtlinux.org/rtlinux.new/index.html

(3) Linux Japan 1999 3 月号、レーザー 5 出版局

(4) ビル・ルイス、ラニエル・バーグ：P スレッドプログラミング、プレンティスホール、1999

(5) Victor Yodaiken, Michael Barabanov：RTLinux Version Two、http://www.rtlinux.org/rtlinux.new/index.html
(6) 船木、羅：LINUX リアルタイム計測/制御開発ガイドブック”、秀和システム、1999 年
(7)インターフェース、1999 年 11 月号、CQ 出版社
(8)遠藤：GNU C Compiler、秀和システム、1998
(9) 八木他：可変受容野を備えた超並列アナログ知能視覚センサ、電子情報通信学会論文誌 Vol.J1-P-I No.2, pp.104-114