＜研究ノート＞

RT-Linux/Linuxを用いたリアルタイム制御に関する研究

職業能力開発総合大学校東京校　　中　村　信　也

A Study on Real-Time Control using RT-Linux and Linux

Shinya NAKAMURA

要約　本研究は、Linuxを用いたI/O制御を中心に、非リアルタイム制御、デバイス・ドライバ作成法、Linux上にリアルタイム処理機能を追加したリアルタイムLinux（以下にRT-Linuxという）の制御方法とその実装方法を段階的に述べる。

研究対象の中心は、Linux上のioperm関数、iopl関数の記述方法、デバイス・ドライバの記述方法、RT-Linuxの導入と移植方法、モジュールフォーマットの作成法、動作方法、Windows（Microsoft）と比較した周期性能検証である。

Ⅰ　はじめに

Linuxはサーバ機能を提供でき、安定性、セキュリティに優れている。マルチタスク、マルチユーザー機能を持ち、FTP、Web、File、DHCP、DNS、メールサーバなどの各種サーバ機能が提供できる。また、TELNETを用いたリモートアクセスが可能である。

本研究は、Linuxを用いたI/O制御を中心に、非リアルタイム制御、デバイスドライバ作成法、Linux上にリアルタイム処理機能を追加したリアルタイムLinux（以下にRT-Linuxという）の制御方法とその実装方法を段階的に述べる。

本研究で取り扱う対象の中心は、Linux上のioperm関数、iopl関数の記述方法、デバイス・ドライバの記述方法、RT-Linuxの導入と移植方法、カーネルの再構築、Windows（Microsoft）と比較したリアルタイム周期性能検証、モジュールフォーマットの作成法、動作方法である。

本研究の最終目標は、RT-Linuxの実装方法が非常に複雑なため、非リアルタイム上のLinuxを用いたI/O制御からRT-Linuxの制御方法と実装方法を提示し、その利便性を検証しつつ段階的に制御教育に寄与することである。

RT-Linuxは、Linux本体の機能とリアルタイム制御機能の共存が可能なため、ネットワークを用いた遠隔操作(1)とリアルタイム制御の混在したシステム構築が可能である。この技術は情報技術系、電子技術系の遠隔制御技術の教育内容に有効と思われるため、リアルタイム制御教育を前提とし実装方法を段階的に述べることにした。従って、この分野に携わる情報電子制御系教育の発展に寄与できるものであると思われる。

しかし、本論文では、遠隔操作に用いられるネットワークプログラミングには触れないこととした。

Ⅱ　研究の背景

近年、プラントは自動化が進み、その制御はコンピュータに依存している。また、遠隔監視を行うための情報の共有化とリモートメンテナンスを行うための制御データが同一のネットワーク上に混在するシステムを構築している。このシステムを実現するに当たって、プラントの自動化はリアルタイム制御が要求され、遠隔操作、リモートメンテナンスにはネットワークプログラ

論文受付日　H16.3.26

ムが要求される。ここで、問題になるのはネットワークプログラムによって遠隔監視、リモートメンテナンスを実現すると同時に、プラント制御のリアルタイム性が常に確保されていなければならないことである。プログラマーがこのようなシステムを構築しようとするならば、ネットワークプログラムと複雑な制御プロセスを記述しながらリアルタイム性を実現しなければならないだろう。

そこで、複数のタスクを実行でき、また、リアルタイム処理が可能なOSを用い、プログラマーは、タスクを記述するだけで実現可能となるシステムについて述べる。本研究は、リアルタイム制御可能なリアルタイムOSにRT- Linuxを用いることにする。

Ⅲ　リアルタイムシステムの定義

リアルタイムシステムは必ずしも、実際に高速であることを意味しなく、適切な時間に起き、その処理が可能であることである。

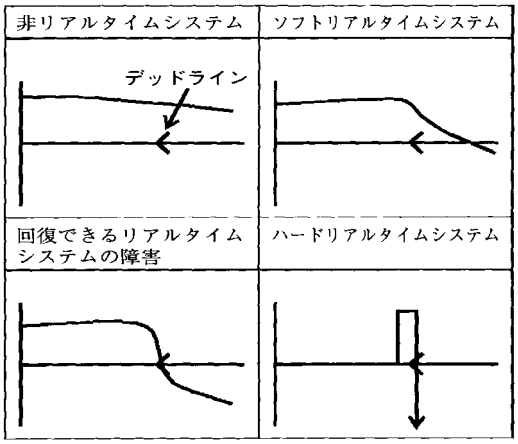


図１　時間厳守ユーティリティ関数

Peter C.Dibble[(2)]は、リアルタイムはハードリアルタイム、ソフトリアルタイム、回復できるリアルタイムの障害、非リアルタイムに分類している。図１に示す。リアルタイムシステムがハードからソフトそして非リアルタイムまでの連続した分類のどこにいるのかで決定されると述べている。

リアルタイムの定義は、デッドラインを境にそのシステムの終了が、完全にデッドライン前に終了する場合と後に終了する場合で区別を定義している。

また、Alan Burns & Andy Wellings[(3)]は、以下に示すような定義を述べている。

“Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. Soft real-time systems are those where response times are important but the system will still function correctly if deadlines are occasionally missed.”

「ハードリアルタイムシステムは、デッドライン内で起こる応答時間を厳守しなければならない。ソフトリアルタイムシステムは、応答時間が重要であるがデッドラインが少々遅れてもこのシステムは正しく機能する。」

Peter C.Dibble、Alan Burns、Andy Wellingsは、ハードリアルタイムとは、厳密な応答時間内の動作を保証することあり、ソフトリアルタイムとは、デッドラインが少し遅れても機能は正しく動作をすると定義している。例えば、ハードリアルタイムは、制御対象が厳密性を問うような自動車のブレーキシステムであり、ソフトリアルタイムは、音声・画像システムの機能に実装可能である。

本研究で、制御対象をプラントの自動化に焦点をあてる。リアルタイム要求応答時間を図２（REAL-TIME UNIX SYSTEMS Design and Application Guide[(4)]）に示す。

図２からIndustrial Automationは、100us～100ms内である。この値を満足できるリアルタイムシステムを前提とする。

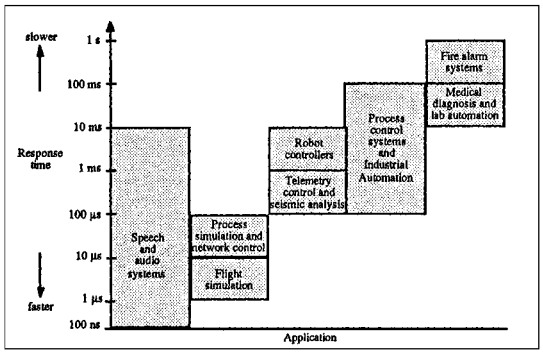


図２　リアルタイム要求応答時間

Ⅳ　制御システムOSの選定

制御システムを設計する場合、Windowsを使用することが一般的に多い。しかし、このOSはI/Oアクセスに関し特権レベルを持っている。WindowsMEの特権レベルは３であり、ME上のVisual Basicでは、DLL（Dynamic Link Library）を作成することで入出力を可能にする。VisualCでは、MS-DOSとほぼ同様なC言語記述で実現できる。

しかし、特権レベル２以上をもつOSのWindows

NT4.0、Windows2000/Professionalでは直接I/Oアクセスが不可能である。そのためマイクロソフト・ツールでドライバーを記述することで可能になる。また、Windowsはラウンドロビン方式を用いて構成しているため、リアルタイム性は保証されていない。

Linuxでは特権レベルが0なので、直接I/Oアクセスが不可能である。そのため、特権レベルを変更できる関数を使用し直接I/O制御をすることにする。しかしリアルタイム性は保証されていない。以上の結果、リアルタイム可能なOSとして、RT-Linuxを選定した。特に、Linux上のドライバはカーネルバージョンの違いにより記述方法がかなり異なる。そのため、比較的文献が多いカーネルバージョン2.2を選定した。

RT-LinuxのベースにLinux（LASER5 Linux6.2カーネル2.2.4）を用い、これに準拠したRT-Linux V2.2を使用した。

I/O制御インターフェースは、PCIとISAバスがあり、現在ではPCIバスが主流である。しかし、PCIバスを用いたデバイス・ドライバの記述は複雑なため、教育目的として指導する場合を考慮し、ISAを用いることにした。今後、Linuxのバージョンが益々進化し、また、ISAバス用のマザーボードが極めて少ない傾向にあるので、PCIバス用のドライバの開発実装方法を教育する必要があるとも思われる。

しかし、工業用「IBM-PC互換機」では、ISAとPCIバス両仕様が今だ健在である。

Ⅴ　Linuxによる制御

一般に制御システムにLinuxを利用するという発想は少なくあまり実績がないようである。しかし、Linuxを用いてパラレルI/O制御が可能であるため、その方法について述べる。

ISAバスに、パラレルインタフェース（Programable Peripheral Interface）PPI8255（Intel製）を接続し、Linux上で制御を行う。表1に動作環境を示す。

表1　動作環境

CPU	Celeron533MHz
MotherBord（DOS/V互換機）	3702EV/ICP製440BX
VGA	ATI Xpert98 PCI 8M
NIC	RealTek RTL8139
Memory	128MB
パラレルI/O：PPI8255	AB10EXE/MITEC
Linux:RdHat6.2	RT-Linux V2.2
デジタルオシロスコープ	Tek TDS340AP

Linux上でI/Oポートを制御する方法は、以下の2種類に大別される。

・I/Oポートの直接アクセス

・デバイス・ドライバ

I/Oポートの直接アクセス方法を述べる。記述はC言語を用い、コンパイラはgccを使用する。

1．I/Oポートの直接アクセス

I/Oポートの直接アクセス方法には、ioperm関数、iopl関数の2種類があり、その用途は、I/Oポートアドレス空間の違いにある。

「IBM-PC互換機」上のコンピュータで使用する場合のアドレス空間の範囲を以下に示す。

・ioperm関数はポートアドレスが0x000～0x3ff

・iopl関数はポートアドレスが0x400以降の場合

(1) ioperm関数

フォーマットは、はじめに、ioperm関数、iopl関数どちらも同様にインラインアセンブラを用いout関数、in関数を記述する。

次に、main関数の中でioperm関数を記述し、ポートの入出力許可を設定する。unistd.hとio.hのヘッダーをインクルードする。

関数は、

```
int ioperm(unsigned long from, unsigned long
    num, int turn_on);
```

で記述され、ioperm は、引数fromのI/Oポートアドレスからnumバイト分の領域に対するアクセスの許可ビットをturn_onに設定する。ioperm の使用にはスーパーユーザー権限が必要である。

ioperm の返り値は成功した場合に0を返す。エラーの場合は－1を返し、

```
fprintf(stderr,"IO-Port,can't open!¥n");
```

で、エラー処理を行う。

リスト1に、ioperm関数の記述例を示す。

(2) iopl関数

iopl関数は、I/O特権レベル0～3に設定でき、通常プロセスのI/O特権レベルは0である。

unistd.hとio.hのヘッダーをインクルードする。

書式は、

```
int iopl(int level);
```

で記述され、引数levelでプロセスのI/O特権レベルを指定した値に変更が可能である。この関数は65536個のI/Oポート全てにアクセス要求をするために設けら

```
static void inline port_out(unsigned char
                    value, unsigned short port) {
__asm__ volatile ("outb %b0,%1"
                    ::"a" (value) ,"d" (port)) ;
}
static unsigned char inline port_in(unsigned
                                    short port) {
                              unsigned char value;
__asm__ volatile ("inb %1,%0"
                    :"=a" (value) :"d" (port)) ;
                return value;
}
int main (void)
{
    if ( ioperm (PORTA,4,1)) {
        fprintf(stderr,"IO-Port,can't open!¥n");
                    return;
    }
        /* 8255 */
    port_out (CWORD,CTLPORT) ;
    port_out (0x5a,PORTC) ;
}
```

リスト１　ioperm関数

れた関数である。

ioperm の返り値は成功した場合に 0 を返す。エラーの場合は－１を返し、

fprintf(stderr,"IO-Port,can't open!¥n");

で、エラー処理を行う。

リスト 2 に、iopl関数の記述例を示す。

```
static void inline port_out(unsigned char
              value, unsigned short port) {
__asm__ volatile ("outb %b0,%1"
                      ::"a" (value),"d"(port));
}
static unsigned char inline port_in(unsigned
                                    short port) {
                    unsigned char value;
__asm__ volatile ("inb %1,%0"
                      :"=a" (value):"d" (port));
return value;
}
int main (void)
{
    if ( iopl(3)) {
        fprintf(stderr,"IO-Port,can't open!¥n");
        return;
  }
  /* 8255 Init port a,b,c=out */
  port_out (CWORD,CTLPORT) ;
  port_out (0x5a,PORTC) ;
}
```

リスト２　iopl関数

２．デバイス・ドライバによるI/O制御

デバイス・ドライバ[5](kernel2.2）の作成方法を述べる。

Linux上で、アプリケーションプログラムから外部機器に特に高速なアクセスを要求する場合、入出力デバイスを直接アクセスすることは難しい。アクセスするにはデバイス・ドライバを経由しなければならない。

Linuxでは、アプリケーションプログラムからデバイスドライバを呼び出すときに、open/close/read/write/ioctl等のシステムコールを用いる。

OSがシステムコールを受けるとOSがデバイス・ドライバを呼び出す。デバイス・ドライバは一定の処理を行った後、OSにいったん制御を渡し再びOSが呼び出し元のアプリケーションに制御を渡す。このようなシーケンスで処理される。

デバイス・ドライバの抜粋をリスト 3 に示す。

デバイスドライバのフォーマットは、

```
static ssize_t dev_write( struct file *file,
                              const char *buf,
                              size_t count,
                              loff_t *x) {
    return (count) ; // not support yet
}
static ssize_t dev_read (struct file *file,
                              char *buf,
                              size_t count,
                              loff_t *x) {
    return (count) ; // not support yet
}
static int dev_ioctl (struct inode *inode,
                              struct file *file,
                              unsigned int cmd,
                              unsigned long arg){
int err = 0;
int size = _IOC_SIZE(cmd) ;
struct io_info io;
void* virt_adrs_page;
unsigned int io_page, io_offset;
if(_IOC_TYPE(cmd)!=dev_MAGIC) return-EINVAL;
if(_IOC_DIR(cmd)&_IOC_READ){
   err=verify_area(VERIFY_WRITE,(void*)arg, size);
}else if(_IOC_DIR(cmd)&_IOC_WRITE){
   err=verify_area(VERIFY_READ,(void*)arg,size);
}
```

リスト３ デバイス・ドライバ

```
int open_env(void)
{
  // open isaio driver
if((fd=open("/dev/isaio0", O_RDWR))<0){
    printf("error:can not open device driver¥n");
    return (FALSE) ;
}else{
return(TRUE);
    }
}
int close_env (void)
{
  close(fd);
  fd = -1;
  return(TRUE);
}
void outport_b(unsigned int port, unsigned char value)
{
struct io_info io;
if (fd < 0) return;//nothing to do
    io.adrs = port;
    io.data = value;
    ioctl (fd, dev_IO_WRITE_8, &io);
}
int main (void)
{
    unsigned char n;
    open_env ();
    outport_b (0x317,0x80);
    outport_b (0x316,n);
    close_env ();
}
```

リスト4　呼び出しプロセス

・open/close/read/write/ioctl

・int init_module(void)

・void cleanup_ module(void)

で構成され、init_module()では初期設定を記述する。

rmmodを使用してモジュールを削除するとモジュールのcleanup_module()が実行され、特にI/O等のリセット、返却コードを記述する。

・デバイス・ドライバ設定=insmod

・デバイス・ドライバ設定=rmmod

で設定、解除を行う。このデバイス・ドライバをユーザーが呼び出すプロセス例をリスト4に示す。

Ⅵ　RT-Linuxの導入、移植、再構築、動作

Linuxはサーバ用に使えるオペレーティングシステムである。しかし、リアルタイム性能は保証されていない。そこで、計測制御に応用するために、Linuxにリアルタイム性能を付加したものがRT-Linuxである。

RT-Linuxは、New Mexico Institute of TechnologyのVictor Yodaiken[6]らのグループによって開発されたものである。特徴は、Linux上の従来プログラムとリアルタイムプログラムが共存できることである。従来プログラムとは、ネットワークプログラム、telnet、Web等が使用できることである。そして、計測制御用プログラムを効率よく記述でき、遠隔メンテナンスに用いられているリモート計測制御が実現できることである。また、Linuxの特徴であるマルチタスク、マルチユーザー機能と、RT-Linux環境下でリアルタイム計測制御が実装できることである。

RT-Linuxは、

Linux2.0系に対応したRT-Linux V1.3

Linux2.2系に対応したRT-Linux V2.2

Linux2.4系に対応したRT-Linux V3.2

があり、本研究ではRT-Linux V2.2を使用する。

導入に当り、インターネット上のRT-Linux公式サイト

http://www.fmslabs.com/

から

rtlinux-2_2-prepatched_tar.gz

rtlinux-2_2_tar.gz

をダウンロードする。

前もって、Linux6.2をインストールし、X-Windowが起動できるようにしておく。

その後、/usr/srcディレクトリでRT-Linuxを展開し、カーネル、モジュールの再構築を行うことでRT-Linuxの移植が完了する。ここで、モジュールの再構築にあたり、

make xconfig

を起動し、140種類以上のモジュールから、コンピュータのハードウェアに適切なドライバを選択し、カーネルコンフィグレーションを行う。以下に手順を示す。

make dep

make bzImage

make modules

make modules_install

make install

で再構築し、Linuxローダー

/sbin/lilo

再定義する。

Linuxのカーネルは、コア部分とカーネルに追加機能プログラムを自由に組み込み削除が可能なモジュールで構成されている。

モジュールの

組み込みはinsmod

削除は　rmmod

確認は　lsmod

を使用して自由にカーネルの拡張機能や不要な機能の削除ができる。

LinuxとRT-Linux間のデータ処理の流れは、ハードウェアからの割り込みをRT-Linuxのリアルタイムカーネルで受け、リアルタイムタスクに処理を渡す。

リアルタイムタスクとLinuxプロセス間の情報交換機能はFIFOで行われる。

動作方法は、リアルタイムタスクとLinuxプロセス間通信、スケジューラ、デバイス・ドライバインターフェース、プロセッサクロック制御のモジュールを組み込みする。以下にモジュールを示す。

- fifo.o
- sched.o
- posix.o
- time.o

このモジュールを組み込み後、ユーザープロセスモジュールを組み込むことで実行可能である。

Ⅶ　WindowsとRT-Linux周期性能比較

RT-Linux基本性能を検証するために、リアルタイム基準周期の最小時間を計測実験する。コンピュータにパラレルI/Oを接続する。I/Oから方形波周期出力を行うリアルタイムモジュールを作成し、周期性能を測定し動作を確認する。

はじめに、シングルタスク上の周期性能を計測し、その後、マルチタスク上で同様な計測をすることで周期性能動作[(7)]の安定性とその時間を測定する。

シングルタスク上で15us幅方形波出力では特に問題は生じない。しかし、14us幅方形波出力では波形に乱れを生じ不安定な動作を生じる。図3、図4に示す。結果から、15usでは安定した動作を確保している。

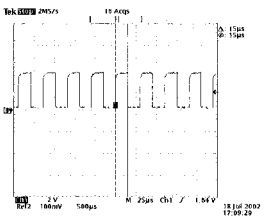


図3　方形波出力15us

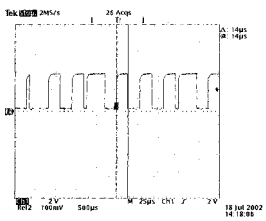


図4　方形波出力14us

次に、マルチタスクにおける周期性能を計測する。バックグランドプロセスとして文字表示タスクを追加し、同様な実験を行った結果、図3、図4と同様な結果が得られた。図5と図6に示す。

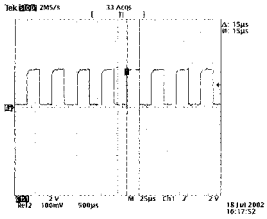


図5　マルチタスク15us

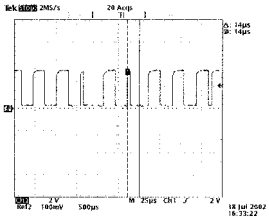
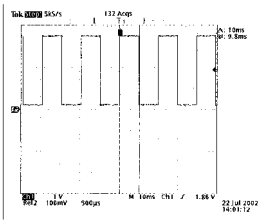
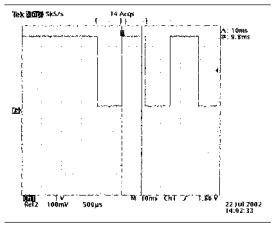


図6　マルチタスク14us



Windows ME上
図7　シングルタスク方形波出力10ms



Windows ME上
図8 マルチタスク方形波出力10ms

RT-Linux上では、リアルタイム制御プロセスにマルチタスクの実装が可能であるといえる。複数のタスクを時間制約の中で実行すること、即ちハードリアルタイムな制御を行うためには、基準時間が常に一定でなければならない。この検証は、最小実時間の計測といえる。

続いて、WindowsME上で同様に、シングルタスクとマルチタスク計測を試みた結果、シングルタスクでは、10msが安定し、マルチタスクで、バックグランドプロセスを追加した場合、安定した状態が得られなかった。図7と図8に示す。これは、Windowsはラウンドロビン方式を用いたタスク処理を行っているためと判断できる。

Ⅷ　モジュールフォーマット

RT-Linuxのモジュールフォーマットは

① void *thread_code(void*t)

② int init_module(void)

③ void cleanup_modules(void)

で構成される。

①はユーザープログラム記述、②は初期設定プログラム記述、③返却プログラムを記述する。デバイス・ドライバ記述と同様である。

insmodによってモジュールを追加すると、

init_module()

が実行される。ここでは、I/O等の初期設定を記述することができる。

rmmodを使用してモジュールを削除すると、

cleanup_module()

が実行され、特にI/O等のリセット、返却コードを記述することができる。

ユーザーのプロセスは

thread_code()

の中で記述され、リアルタイム実行条件は、

pthread_make_periodic_np()

周期実行の抑制

thread_wait_np()

関数を記述することができる。

リスト5に示す。

```
#include <rtl.h>
#include <pthread.h>
#include <asm/io.h>
#define ppi_a 0x410

void *thread_code(void*t)
{
    user program
}
int init_module(void)
{
    init programu
}
void cleanup_modules(void)
{
    返却  program
}
```

リスト5 RT-Linuxモジュール

Ⅸ おわりに

Linuxを用いたI/O制御を、非リアルタイム制御、デバイス・ドライバ作成法、Linux上にリアルタイム処理機能を追加したRT-Linuxの制御方法とその実装方法を段階的に述べた。

本研究の成果は、第1に、Linux上から直接I/O制御を行う方法を、ioperm関数、iopl関数を用いる方法を提示した。

第2に、デバイス・ドライバを用い、その記述例と組み込み方法、ユーザープロセスからデバイス・ドライバの呼び出し方法を提示した。

第3に、RT-Linuxの導入と移植方法、カーネルの再構築、モジュールフォーマットの作成法、動作方法を提示した。

第4に、RT-LinuxとWindowsMEの周期性能を比較し、その結果、RT-Linuxのリアルタイム処理、マルチタスク処理は、最大15usの周期性能が得られた。

しかし、WindowsME上では、ラウンドロビン方式を用いているため、リアルタイム制御には使用できないことが確認できた。

本研究の最終目標は、非リアルタイム上のLinuxを用いたI/O制御からデバイス・ドライバ、そしてRT-Linuxの制御・実装方法の利便性を検証することである。

また、RT-Linuxの実装方法が非常に複雑なため、リアルタイム制御を段階的に教育する課題の作成にも寄与できるものと考えられる。

本研究終了後の技術的な課題として、制御装置をリアルタイム化し、遠隔操作技術を付加することである。特に、TCP/IPプロトコル確立、ソケットを用いたネットワークプログラミング技術の実装である。

[参考文献]

(1) W. RICHARD STEVENS, UNIX NETWORK PROGRAMINNG VOLUME1 SECOND EDITION, Prentice Hall PTR, Prentice-Hall. Inc, 1998, pp.3-28.

(2) Peter C. Dibble,（訳）音川 英之, 白川 洋充, リアルタイムJavaプログラミング, PEARSON Education Japan 2003, p.8.

(3) Alan Burns & Andy Wellings, REAL-TIME SYSTEMS AND THEIR PROGRAMMING LANGUAGES, ADDISON-WESLEY PUBLISHING Company 1990, p.2.

(4) Borko Furht, Dan Grostick, David Gluch Guy Rabbat, John Parker, Meg McRoberts, REAL-TIME UNIX SYSTEMS Design and Application Guide, KLUWER ACADEMIC PUBLISHERS 1991, p.2.

(5) ALESSANDRO RUBINI, JONATHAN CORBET,（訳）山崎 康宏、山崎 邦子, LINUXデバイスドライバ, O'REILLY:オライリー・ジャパン 1998.

(6) Victor Yodaiken, Michael Barabanov, A Real-Time Linux, New Mexico Institute of Technology, 1997.

(7) 技術者のためのUNIX系OS入門 Vol.5 TECHI CQ出版 2001,p.120.