SPring-8 加速器制御系 VMEbus 計算機の 64-bit OS 環境への移行について

MIGRATION EXPERIENCE OF VMEBUS COMPUTERS TO 64-BIT OS ENVIRONMENT IN THE SPRING-8 ACCELERATOR CONTROL SYSTEM

増田剛正#
Takemasa Masuda#
Japan Synchrotron Radiation Research Institute (JASRI)

Abstract

In the SPring-8 accelerator control system, VMEbus computers operated by Solaris OS have been used as front-end computers. Since 2013 when VME multi-core CPU boards were introduced, the problem that the CPU load was increased and the running processes were frozen for over a dozen seconds has been periodically occurred. As the future-oriented approach to solve this problem, these VMEbus computers were migrated to 64-bit OS environment. 64-bit compliance of device drivers was indispensable to achieve these migrations. The migrations of about forty VMEbus computers have been completed after porting fourteen kinds of existing device drivers. My adopting approach to reduce the porting cost of the device drivers, the evaluation of these migrations, and the obtained knowledge through the experience of the migration work are reported in this paper. In addition, consideration of the future device drivers is discussed.

1. はじめに

1 GeV リナック、8 GeV ブースターおよび 8 GeV 蓄積リングから構成される SPring-8 加速器複合体の制御系では、加速器コンポーネントを制御するためのフロントエンド計算機として VMEbus 計算機が採用され、サイト内に分散設置されている。およそ 120 台の VMEbus 計算機が SPring-8 加速器制御系全体で使用されている（2018 年 2 月時点）。VMEbus 計算機の CPU ボードとしては x86 CPU を、OS としては Solaris を採用している。

1997 年の共用運転開始時は、リナックとブースターでは 68000 系の CPU ボードと OS-9[1]、蓄積リングでは PA-RISC CPU ボードと PA-RISC 版 LynxOS[2]である HP-RT を採用していた。当時は十分な性能の CPU や十分な容量の物理メモリが無かったため、全世界的にコンパクトでリアルタイム性に優れた OS を採用していた。リアルタイム OS の条件として、優先度に基づくタスクスケジューリング、決定論的なタスク切り替え時間などがあり、またリアルタイムタイマが使用可能であるということも含まれていた。

2000 年のリナック制御系更新のタイミングで、高い CPU 性能と十分なメモリ容量を備える x86 系 CPU ボードと Solaris OS の組み合わせを新たに採用することとなった。Solaris は優先度に基づくスケジューリング、決定論的なタスク切り替えが可能なリアルタイム OS であり[3]、当時の Solaris 7 より Linux 同様フリーOS として利用できるようになった（OS-9 や LynxOS はライセンス料が必要）。また OS の HZ 値をデフォルトの 100 から変更することで、タイマー割り込みの周期を 10 ミリ秒からより小さな値に変更することができる。例えば HZ 値を 1000 に変更することで、1 ミリ秒刻みのきめ細かい時間制御が可能となる。これらが決め手となって Linux を含む幾つかの候補の中から Solaris を選択した。

近年、VMEbus 計算機においても、ソフトウェアの並列処理が可能なマルチコア CPU が使用可能となった。並列処理が可能な MADOCA II[4]フレームワークへの移行に合わせて、2011 年から積極的にマルチコア CPU ボードの導入を始めた。

2. Solaris デバイスドライバ

2.1 構造と特徴

VMEbus 上の入出力ボード等のデバイスの制御を行うためには、OS のカーネル空間で動作するデバイスドライバを用意する必要がある。Solaris におけるデバイスはツリー構造になっており（Figure 1）、VMEbus や PCIbus 等のコンピュータバスに関わるネクサスノード部の制御にはネクサスドライバを、個別の入出力ボード等のリーフノードの制御にはリーフドライバをそれぞれ用意する必要がある。リーフドライバはコンピュータバスからは独立した形式で記述され、割り込みや Direct Memory Access (DMA) 転送などバスアクセスが必要な処理は基本的にネクサスドライバを介して行う。そしてネクサスドライバやカーネルとの取り合いは、Device Driver Interface / Driver-Kernel Interface (DDI/DDK)と呼ばれるインターフェースを介して行う。DDI/DDK は長年にわたって殆ど形を変えることなくサポートが継続されているため、Solaris のリーフドライバはカーネルが更新されてもほぼそのまま動作する。

Solaris におけるデバイスドライバの階層構造、

#masuda@spring8.or.jp

DDI/DKI インターフェースの提供は、作成したデバイスドライバの長寿命化に非常に有効である。実際に我々は Solaris の利用を開始した 2000 年から現在に至るまで、OS のバージョンで言えば Solaris 7 から Solaris10 まで、使用する x86 VME CPU ボードが替わっても、殆どのデバイスドライバがそのまま継続して利用出来ている。ただし DDI/DKI インターフェースがサポート外となった関係で、一部のデバイスドライバについてはコードの修正を行っている。

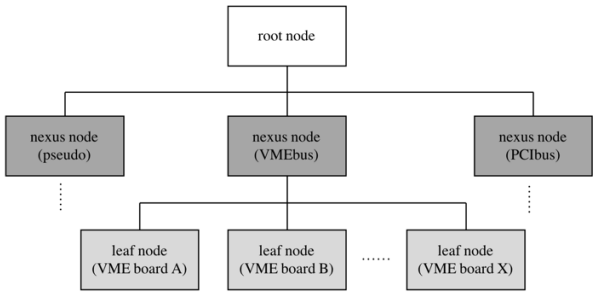

Figure 1: An example of the Solaris device tree.

2.2 バスブリッジチップ用ネクサスドライバ

初期の x86 VME CPU ボードにおいては、PCIbus から VMEbus へのブリッジチップとして、IDT 社の Universe II チップ[5]がデファクトスタンダードとして使用されていた。我々は Universe II 用のネクサスドライバを作成し、その上で動作する各種入出力ボード用のリーフドライバを多数整備した。

後期の x86 VME CPU ボードにおいては 64-bit 環境への移行が可能となるよう PCI-Xbus から VMEbus へのブリッジチップである Tsi148 チップ[6]がデファクトスタンダードとして使用されるようになった。これまで多数整備したリーフドライバ群がそのまま使用できるよう、我々は Tsi148 チップ用ネクサスドライバを作成した。実際にこの試みは成功し、どちらのバスブリッジチップを乗せた x86 CPU ボードであっても、対応するネクサスドライバを用いるだけで、同一の入出力ボードは同一のリーフドライバを使用して制御することができるようになった。

2.3 64-bit Solaris への適用について

ただし、この状況には一つだけ例外がある。それは 64-bit Solaris への適用についてである。デバイスドライバはカーネル内部に組み込まれて動作するため、64-bit OS においては 64-bit 対応コードで記述されなければならない。残念ながら我々が作成したデバイスドライバは 64-bit 対応のコードにはなっていない。したがって 64-bit の Solaris 10 や 64-bit のみ提供されている Solaris 11 で使用するためには、デバイスドライバのソースコードを 64-bit 対応コードに書き換える必要がある。

3. 32-bit OS 環境で発生した問題

マルチコア CPU ボードと 32-bit Solaris を導入した VMEbus 計算機において、周期的に CPU 負荷が上昇して十数秒間制御プロセスがフリーズする問題が発生した（Figure 2）。この間制御プロセスが動作しないため、収集データに欠落が生じたり、機器の制御に遅延が生じたりする現象が発生した。

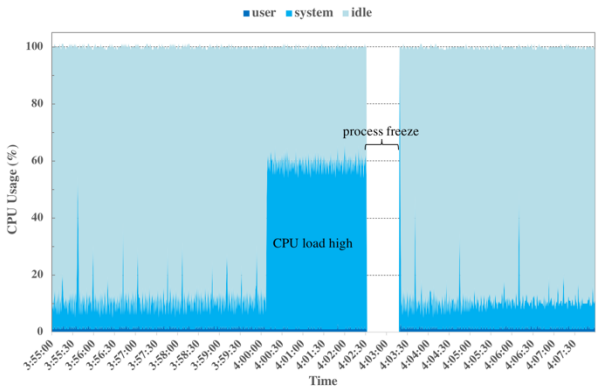

Figure 2: So-called “24 days problem” that the CPU load became high and running processes were frozen.

調査をしたところ、これは uptime カウンタ（32-bit OS では 32-bit 幅）の最上位ビットが反転する際に発生する問題であることがわかった。我々の環境では 1 ミリ秒の精細な時間制御を有効にするため HZ 値を 1000 としていることから、uptime カウンタは 1 秒間に 1000 カウントアップする。そのため最初の問題はシステム起動後 24.8 日目に発生し、以後は 49.7 日周期で問題が発生する。これは実際に問題が発生した周期と一致する。この問題は世間一般では 248 日問題、あるいは 497 日問題と言われている現象であるが、我々の環境では HZ 値を通常の 100 から 1000 にしているため問題発生の周期が 1/10 となっている。ここでは以後この問題を 24 日問題と記すことにする。

この 24 日問題を解決するためには OS を 64-bit 化するのがもっとも有効な方法である。OS を 64-bit 化するためには、先に述べた通り関係する全てのデバイスドライバを 64-bit 化しなければならない。具体的には 14 種類のデバイスドライバを 64-bit 化する必要があった。これには膨大な時間とコストが掛かるため、先ずは 32-bit OS のままでの問題の解決を試みた。具体的には Oracle とサポート契約を結び、技術サポートを受けながら関係のありそうなパッチを当てるなどの対策を試みた。同時に CPU ボードメーカーからも BIOS の設定等で問題の解決に繋がるものはないか技術サポートを受けたが、結局問題の解決には至らなかった。そのため最終的には、最も未来志向的な解決方法として OS を 64-bit 化する方法を採用することとした。また 64-bit 環境への移行にあたっては、最新の Solaris 11 をターゲットとした。

4. 移行の方針

4.1 方針

先述の通り、64-bit OS（Solaris 11）への移行にあたっては、Tsi148 バスブリッジチップ用のネクサス

ドライバ 1 種類を含めて 14 種類ものデバイスドライバを 64-bit 化する必要があった。

通常、デバイスドライバの作成・改修は簡単なものを除きアウトソーシングしている。その場合、入出力ボードと組み合わせて動作試験まで行うのが一般的である。簡単なリーフドライバも幾つか含まれているとはいえ、動作試験を含めて 14 種類ものデバイスドライバ改修をアウトソーシングするとなると膨大なコストと時間が必要となる。一方、内部スタッフによる改修は、マンパワーや改修に掛かる時間、必要とされる知見等を考慮した場合現実的ではなかった。デバイスドライバの 64-bit 化は、改修すべき箇所や改修方法のパターンはほぼ決まっているが、実際に改修を行うには後述のように高度な知見が必要とされるためである。そこで移行にあたっては以下のような方針を立てた。

- Solaris デバイスドライバについての高度な知識・豊富な経験を有する会社にアウトソーシングする。
- アウトソーシング先での改修は机上での作業に留め、ハードウェアを含めた動作確認は我々の責にて行う。
- 動作確認で不具合が見つかった場合には、アウトソーシング先において無償で修正する。

4.2 改修項目

デバイスドライバの 64-bit 化にあたっては、以下の事項を確認し、必要に応じて改修する[7, 8]。基本的にはデータ型の変更に起因するものである。

- long 型の確認
- ポインタのアドレス計算の確認
- ビットシフトの確認
- 構造体の確認
- 定数の型指定
- ioctl エントリポイントの確認
- ioctl パラメータの確認
- 32-bit/64-bit 共用マクロ
- DDI アクセス関数の確認
- 書式付き出力の確認

このうち、改修へのインパクトが大きいのが ioctl のパラメータである。基本的には、64-bit 化への改修はデバイスドライバ内で閉じるのであるが、これについてはアプリケーション側にまで改修の影響が及ぶことがあるためである。

通常、ioctl のパラメータには構造体を使用することが多いのだが、構造体のメンバに long やポインタが含まれる場合には、呼び出し側のアプリケーションが 32-bit か 64-bit かによってデバイスドライバの参照の仕方を変える必要がある。また、パラメータとして構造体ではなく配列を用い、その中でアドレスを渡している場合も注意が必要である。アプリケーション側で int 型に cast してしまっているような場合、64-bit アプリケーションでは上位 32-bit のアドレスが省かれ、デバイスドライバ側に正しいアドレスが伝わらなくなる（Figure 3）。

このような問題に対しては、新規に構造体を用意してそのメンバの中でアドレスを渡すことで回避することができる（Figure 4）。ただしこの改修はアプリケーション側に影響を与えてしまう。幸い、SPring-8 ではデバイスドライバを呼び出すための API 関数を用意しているため、改修は API 関数までで済ませられる。したがって、アプリケーションのソースコードまで改修する必要はないのだが、実行形式は作り直す必要が生じる。またアドレス渡しの部分を改修するため、この処理に誤りがあると動作テスト時にシステムのハングアップやシステムクラッシュを引き起こす。このような状況に陥るとデバッグが非常に困難になり、システムの復旧に非常に時間が掛かってしまう。改修にあたって高度な知見が必要となる理由である。

```
int dev_bpmdsp_write_dpram_data( int fd, u_char dsp, uint loc, uint num, int *data )
{
   int arg[4];
   …
   arg[0] = (int)dsp;
   arg[1] = (int)loc;
   arg[2] = (int)num;
   arg[3] = (int)data;   /* pass the address of data variable */
   if ( err = ioctl( fd, BPMDSP_REQ_WRITE_DPRAM, arg ) ) {
      …
}
```

(a) API function

```
…
static int bpmdsp_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *credp, int *rvalp)
{
   …
   switch( cmd ) {

      case BPMDSP_REQ_WRITE_DPRAM:
         int param[4];
         /* receive the address of user space variable in param[3] */
         if( copyin( (void *)arg, param, sizeof(param) != 0 )
            …
}
…
```

(b) Device driver

Figure 3: An example of a device driver which doesn't work in the 64-bit environment.

5. 移行の実際

5.1 机上改修について

3.1 項で述べた方針に従って、14 種類のデバイスドライバ全てをアウトソーシングによって改修した。24 日問題が発生するホストをできる限り早く減らせるよう、改修は多くのホストで使用されている VME ボードのデバイスドライバから始めることにした。全てのホストで使用される Tsi148 ネクサスドライバからスタートし、光伝送ボード、リフレクティブメモリボード、RIO ボード、BPMDSP ボードの 4 種類のリーフドライバを 64-bit 化することにした。これで移行が必要な 37 ホストのうち 25 ホストの移行が可能となる。その後、残りの 12 ホストの 64-bit OS 環境への移行が可能となるよう、使用数の少ない残り 9 種類のリーフドライバの 64-bit 化を行なった。

ハードウェアを含めた動作確認を行わず机上での改修に留めたことで、通常の 1/7~1/10 程度の予算で済ませることができた。アウトソーシング先の技術者が Solaris デバイスドライバに関する高度な知識や経験を持っていたことで、机上での改修にも関わらず、多くのデバイスドライバが 64-bit Solaris においてほぼ問題なく動作した。動作確認において幾つか

問題はあったが、その多くが 3.2 項で述べた ioctl パラメータに関わるものであった。

```
typdef struct {
        int dsp_num;
        int pos_num;
        int data_num;
        int* start;
    } Bpmdsp_dpram;
#ifdef _MULTI_DATAMODEL
    typedef struct {
        int dsp_num;
        int pos_num;
        int data_num;
        uint32_t start;
    } Bpmdsp_dpram32;
#endif /* ! _MULTI_DATAMODEL */
```

(a) Newly introduced data structure

```
int dev_bpmdsp_write_dpram_data( int fd, u_char dsp, uint loc, uint num, int *data )
{
        Bpmdsp_dpram arg;
        …
        arg.dsp_num = dsp;
        arg.pos_num = loc;
        arg.data_num = num;
        arg.start = data;

        if ( err = ioctl( fd, BPMDSP_REQ_WRITE_DPRAM, &arg ) ) {
                …
        }
        return DEV_BPMDS_OK;
}
```

(b) Revised API function

```
…
static int bpmdsp_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *credp, int *rvalp)
{
  …
  switch( cmd ) {
    …
    case BPMDSP_REQ_WRITE_DPRAM:
      Bpmdsp_dpram dpram;
#ifdef _MULTI_DATAMODEL
      Bpmdsp_dpram32 dpram32;
      switch (ddi_model_convert_from(flag & FMODELS)) {
        case DDI_MODEL_ILP32:
          /* copy 32-bit args data shape */
          if (ddi_copyin((void *)arg, &dpram32, sizeof(Bpmdsp_dpram32), flag) != DDI_SUCCESS)
            return (EFAULT);
          /* convert 32-bit to 64-bit args data shape */
          dpram.dsp_num = dpram32.dsp_num;
          dpram.pos_num = dpram32.pos_num;
          dpram.data_num = dpram32.data_num;
          dpram.start = (int*)(uintptr_t)dpram32.start;
          break;
        case DDI_MODEL_NONE:
          /* application and driver have same data model. */
          if (ddi_copyin((void *)arg, &dpram, sizeof(Bpmdsp_dpram), flag) != DDI_SUCCESS)
            return (EFAULT);
          break;
      }
#else /* ! _MULTI_DATAMODEL */
      if (ddi_copyin((void *)arg, &dpram, sizeof(Bpmdsp_dpram), flag) != DDI_SUCCESS)
        return (EFAULT);
#endif /* ! _MULTI_DATAMODEL */
  }
  …
```

(c) Revised device driver

Figure 4: An example of the revised device driver which work in both 32-bit and 64-bit environment.

5.2 64-bit OS への移行作業

実際の 64-bit OS への移行作業は、加速器の長期運転停止期間に順次実施した。先に 64-bit 化を終えた Tsi148 ネクサスドライバと 4 種類のリーフドライバ（光伝送ボード、リフレクティブメモリボード、RIO ボード、BPMDSP ボード）で構成されたホストの移行作業から開始した。

4 種類のリーフドライバのうち、光伝送ボードとリフレクティブメモリボードについては改修がデバイスドライバ内部に留まったため、デバイスドライバのみを入れ替えるだけで済んだ。このため光伝送ボードならびにリフレクティブメモリボードのみから構成される 13 ホストの移行については全く問題なく実施できた。

一方、RIO ボードと BPMDSP ボードについては先に述べた ioctl パラメータに関わる改修を行ったため、デバイスドライバを呼び出すための API 関数も修正した。その関係でアプリケーションを再リンクする必要が生じた。結果的にはデバイスドライバ自体には全く問題はなかったのであるが、5.3 で述べるように、アプリケーションのビルド環境に起因する問題で、再リンクし直したアプリケーションに問題が発生した。そのため RIO や BPMDSP を含むホストについては何度か 64-bit 環境への移行を中止する事態を経験した。

残りの 9 種類のリーフドライバ、12 ホストの移行作業ではアプリケーションの再リンクが必要な場合も含まれていたが、上述の経験を元に問題を回避することが出来た。デバイスドライバ自体に問題が含まれているケースが含まれていた関係で、デバッグや修正には時間が掛かったが、移行自体は問題なく実施することが出来た。

5.3 移行時に発生した問題

64-bit OS への移行時に以下のような問題が発生した。いずれも改修したデバイスドライバの問題ではなく、開発環境や OS 自体の問題であった。

1. アプリケーション側ではデバイスドライバのシステムコールが正常に実行されているにも関わらず、実際には対応するデバイスドライバがコールされておらず、ボードのレジスタ操作が行われない問題。
2. 時々5~10 秒程度プロセスがフリーズする問題。

1 については明確な原因が特定できずにいるが、デバイスドライバの API 関数を再コンパイルした開発環境に問題があったようである。別の開発環境で API 関数をコンパイルした場合にはこの現象は起きなくなった。この問題は全てのドライバコールで起きたわけではなく、一部の特定のドライバコールでのみ発生した。また改修箇所とも無関係であった。

2 の事象は、Solaris10 までには見られなかった現象で、Solaris11 から新たに出現した問題である。詳細に調査を行うと、10 秒前後のプロセスのフリーズと、5 秒前後のプロセスのフリーズの 2 種類の問題に分けられることが分かった。前者は Solaris11.2 以降で起こる問題で、システムパラメータの変更によって回避可能である。後者は Solaris11.2 以降で、かつ特定の CPU ボードでのみ起こる事象であり、現時点で原因は不明である。両者とも Solaris11.1 では発生しないため、OS を Solaris11.1 に戻すことで回避した（移行作業時は Solaris11.3 を選択）。

いずれの問題も事前に見つけ出すのは難しかったと考えられるが、これらの問題が起きたために 64-bit 環境への移行が大きく遅延してしまった。改修したデバイスドライバ自体の問題ではなく、別の要因で移行作業が遅れたことは非常に残念であったが、64-bit 環境への移行は改修したデバイスドライバが問題なく動作すれば良いというわけではなく、開発環境やビルドしたアプリケーション、OS まで含めて考えなければならないことを改めて認識した。

6. 移行から得られた知見

今回は移行のためのコストを抑えるため、デバイスドライバの改修を机上で行ったが、やはり机上の改修作業だけでは完全な移行は難しい。オリジナルのデバイスドライバの出来が余り良くないと改修量が多くなってしまうし、特に ioctl のパラメータ部分の改修、すなわちユーザー空間とのデータの受け渡しに関わる部分の改修は困難が伴う。可能であるのならば、デバイスドライバ改修時にボードレベルでの動作確認試験を行うのがベターである。

上述の移行時の問題はいずれもビーム診断系のホストで発生したが、ビーム診断系のホストは運転停止期間中の動作試験が容易ではなく問題の発見が遅れてしまった。他の機器は運転停止期間中でも動作確認試験が行えるので問題ないが、可能であれば、電子ビームの蓄積中に動作試験が行えるとよりスムーズな移行が実現できたと思われる。

また新しい OS の導入は慎重になるべきである。今回 64-bit OS として Solaris11、厳密には Solaris11.3 を導入したが、これは既に SACLA において Solaris11、厳密には Solaris11.1 の運用実績があり問題ないと考えたためである。しかし、実際には Solaris11.3 において上述の問題が生じてしまった。デバイスドライバの 64-bit 化自体が非常に困難な作業であるから、OS の選択肢としてはより安定な動作が期待できる Solaris10 の 64-bit 版を選択した方がベターであった。

32-bit OS 環境から 64-bit OS 環境への移行を実際に行って強く感じたことは、この移行は容易ではないということである。ioctl のパラメータ部の改修はアプリケーション側の改修が必要となる点で特に困難である。故に 64-bit OS への移行は必要最小限にすべきである。問題を回避する別の方法があるのであれば、先にそちらを検討した方が良い。問題の解決は、出来る限りアプリケーションサイドで行うべきである。

今後新たにデバイスドライバを製作する場合には、当然 64-bit OS への対応を考えるべきである。更に言うならば、デバイスドライバ内の記述量は出来る限り減らすべきである。すなわち、デバイスドライバにはレジスタアクセスや割り込み、DMA 転送などカーネル内部でしかできない必要最小限の処理にとどめ、ユーザー空間でも可能な処理であれば、ユーザー空間で動作する API 関数として提供した方が良い。デバイスドライバを出来るだけシンプルにすることで、 64-bit 化の改修量を減らすことが出来る（アプリケーション側は 32-bit のままでも問題ないため）。

7. デバイスドライバの在り方について

今回の 32-bit OS 環境から 64-bit OS 環境への移行の経験を通して、移行作業の中心であったデバイスドライバの在り方について、ここで考察したい。

Solaris が提供するデバイスドライバの階層構造は、保守性や堅牢性の向上を考慮した場合、非常に有益なアプローチである。バスアクセス部分をネクサスドライバに集約することで、デバイス固有の制御を行うリーフドライバがシンプルになり、コードの可読性が向上する。64-bit 化のような改修作業も容易になる。そして手順が複雑なバスアクセス部の制御を担うネクサスドライバの信頼性を向上させることで、システム全体の堅牢性が増す。

Solaris よりも一般的に利用されている Linux においても同様のアプローチでデバイスドライバを作成し、バスアクセス部分のデバイスドライバを共同で開発・利用するのが良いと考える。既に DESY で MicroTCA プラットフォームをターゲットにした PCI Express Universal Driver (upciedev)が開発され、オープンソースで公開されている[9]。これは MicroTCA がターゲットではあるが、通常の PC をはじめとした PCI Express ベースのプラットフォームであれば適用出来るはずである。この upciedev の活用を世界的な取り組みとして拡大させ、デバイス固有のデバイスドライバを upciedev の上に作成する。これらデバイス固有のドライバもオープンソースで公開し、誰でも利用出来るようにすることで、ドライバ開発の時間とコストを削減し、対象デバイスの導入までの時間を短縮できるようになる。他のバスアクセスドライバについても、upciedev と同様の手法で標準化を進めては如何であろうか。

謝辞

デバイスドライバの 64-bit 化改修作業においては、有限会社平成の河原章氏に多大なる貢献を頂きました。デバイスドライバの机上での改修は、彼のソフトウェアスキルを以って実現できた手法であると言えます。この場を借りて御礼申し上げます。

参考文献

[1] http://www.microware.com/
[2] Lynx Software Technologies, Inc.; http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/
[3] T. Masuda *et al*., “SPring-8 線型加速器ビーム位置モニター用同期データ収集システムの更新”, Proceedings of the 10th Annual Meeting of Particle Accelerator Society of Japan, Nagoya, Japan, Aug. 3-5, 2013, pp. 1118-1121.
[4] T. Matsumoto *et al*., “Next-Generation MADOCA SPring-8 Control Framework”, Proceedings of ICALEPCS2013, San Francisco, California, USA, 2013, pp. 944-947
[5] https://ja.idt.com/products/interface-connectivity/vme/ca91c142d-vme-pci-bridge
[6] https://ja.idt.com/general-parts/tsi148-vme-pci-x-bridge
[7] Solaris 64bit Developers Guide: https://docs.oracle.com/cd/E53394_01/html/E61689/index.html
[8] Writing Device Drivers, Appendix C Making a Device Driver 64-Bit Ready: https://docs.oracle.com/cd/E36784_01/html/E36860/lp64-35004.html#scrolltoc
[9] https://github.com/MicroTCA/upciedev