第 7 回 デバイスドライバ

こんにちは。
まず最初に前回の問題の解を考えてみたいところですが、今回も分量が増えそうなので、分量が少ないであろう、次回最終回に回したいと思います。

さて、今回は、通常は組込みソフトウェアの中では最も難しいもののなかの一つとみなされ、おそらくみなさんが最も興味を持っているであろう、デバイスドライバに関する話題です。

デバイスとはペリフェラル、コントローラ、ドライバ、周辺、周辺チップあるいはチャンネルとよばれる
CPU とは独立して動作できるもの？CPU とは並列に実行できるものです。
そしてデバイスドライバには当然 CPU とデバイスのやり取り、通信も含まれます。
基本的には速度の速い CPU が速度の遅い I/O とのやり取りを直接実行する代わりに、CPU とは独立して実行できるデバイスに I/O とのやり取りを依頼する、もちろん依頼しっぱなしではなく、処理終了時にはその結果を受け取らなければなりません。
当然デバイスとのやり取りは一種の通信であり、CPU とデバイスの同期をとるために割り込みの制御も必要です。そしてデータに関しても CPU とデバイスによる同じ領域へのアクセスが必然的に発生するため排他制御も必要です。
排他制御にしてもタスク間の排他制御ではなく、タスクと割り込み処理、あるいは OS と割り込み処理の排他制御が必要な場合もあり、RTOS を使用しているからと言って単純にセマフォが使えるわけでもありません。
また CPU とデバイスでは実行速度も異なりますが、そこにはたとえば Ethernet 通信などのフローコントロールのように規格化された手順があるわけではなく、使用するデバイスごとに固有の手順を用いて制御をおこなわなければなりません。
あるいは速度の差を吸収するために、リングバッファなどの方法を用いなければならないかも知れません。
これらすべての制御を一定の時間制約のもとで行わなければなりません。
これらの処理を行うソフトウェアがデバイスドライバであり、さまざまな要素が絡み合ってデバイスドライバの作成を難しいものにしています。
しかし組込みソフトウェアにおいてはデバイスも含めた最適化が必要であり、デバイスドライバの開発作成技術もが非常に重要となります。

図 1 に改めて CPU とデバイスの関係を示します。
ハードディスクコントローラやネットワークコントローラなどのデバイスは、CPU のように命令をデコードして動作することこそできませんが、デバイスの内蔵レジスタに書き込まれたビットパターンやアドレスデータなどにより、CPU とは独立して並列に動作を行うことが出来ます。
CPU はデバイスに指令のみを与えて、自分は別のことを実行しながら、デバイスが実行した結果を後で（適

当なタイミングで）受け取ればよいことになります。
ただし、多くの場合最初にデバイスに対する要求を出すのはアプリケーション（タスク）です。そのため RTOS はタスクに対してもデバイスドライバを呼び出す仕組みを提供しなければなりません。

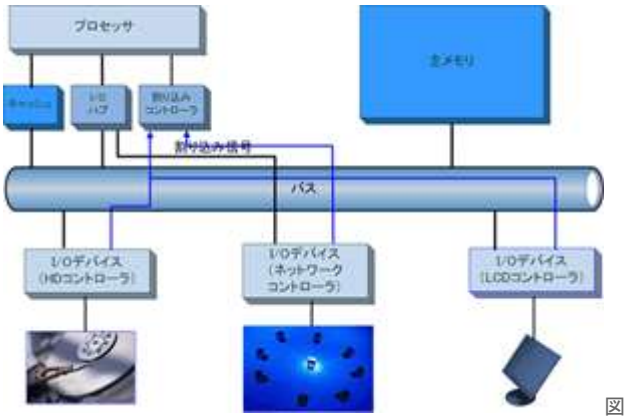

図 1 CPU とデバイス

一般的にデバイスドライバは以下の二つの部分から構成されています。

- アプリケーション（タスク）に提供され、タスクコンテキストで実行されるライブラリ
  I/O が終了するまで待ち状態になる処理などもここに含まれます。
- 対象とするデバイスからの割り込みに対応する割り込みハンドラ、割り込みサービスルーチン(ISR)、この ISR の中でデバイス処理の終了後、待ち状態のタスクの起床処理なども行います。
  プロセスごとに独立したアドレス空間を持つような汎用 OS の場合は、デバイスドライバ終了後 OS に一旦制御が戻り、そこでデバイスから OS 空間に生成されたデータを、改めてプロセス空間にコピーするなどの処理を行ったのち、プロセスに制御が戻る場合が多いのですが、そもそもアドレス空間が一意の RTOS の場合はデータのコピーも必要ではなく、デバイスドライバから直接タスクに制御が戻る場合が多いようです。 と言ってももちろんディスパッチャは経由します。RTOS 環境で多くの処理を行わないという意味です。

図 1 にはデバイスから CPU への通信手段として割り込み信号も示しています（バス経由の割り込み信号も、バスを経由しない割り込み信号もあります）。
デバイスが CPU から要求された処理を終了した時や CPU に対して新たな処理を要求する場合の手段と割り込みを使うことが最も一般的だからです。
もちろんポーリングやスピンロックを使う場合もありますが、最も一般的な割り込みを使う場合を説明します。
この割り込みに対応するために、デバイスドライバには割り込みハンドラや割り込みサービスルーチン（以後両者をまとめて ISR と呼びます）も必要になります。

```
アプリケーション（タスク）{
 char buffer[1024];
:
:
 read("/usr/local/data1",  buffer, ...);
 x = buffer[0];
 y = buffer[1];
 :
```

図 2 アプリケーション（タスク）からの呼び出し

図 2 に示したものはアプリケーション（タスク）からのデバイスドライバの呼出し手順です。VxWorks は標準 I/O をサポートしているので、ほとんどのデバイスドライバの呼び出しは (create)、open、read、write、ioctl、close、(delete) で実行できます。
詳しい説明は省きますが、大部分のデバイスドライバの実行は read/write で実現できます。
ネットワークである TCP socket からの呼び出しも、ハードディスクからの呼び出しもどちらも read で実現できます。これは実際に使ってみると大変便利です。
I/O を行うデバイスの変更を行う場合も open の引数の変更のみで済む場合も多くなります。

```
read(char* name,  char*  buf, option...)
{
   initialize  devices;
   start devices;
   semTake(...);
   return  num;
}
```

図 3 ライブラリ

図 3 は RTOS がアプリケーションに提供するデバイスドライバ呼び出しのための大枠を示したものです。このライブラリはタスクの実行環境で実行され、最終的にはデバイスドライバの実行が終わるまで待ちに入る場合が多いです。
待ち状態から復帰するのはデバイスドライバの処理が終了したか、あるいは何らかのエラーにより処理が中断した場合です。

```
decice_ISR(...)
{
   char temp[1024];
   while(...) {
      copy data from register to temp;
   }
   semGive(...);
   jump dispatcher;
}
```

図 4 割り込みハンドラ（割り込みサービスルーチン）

図 4 がデバイスドライバの処理の大枠を示したものです。これはデバイスからの割り込みにより呼び出されます。たとえばデバイスの処理が終了した時の呼び出しではデバイスのレジスタにあるデータを RTOS やタスクの管理下にあるメモリにコピーしたりします。
デバイスドライバはデバイスによって異なりますし、処理を詳しく説明すると分量が大幅に増えてしまうので、ここではごく概要の説明だけにとどめます。
ひとつだけ注意しておきますが、待ち状態のタスクを実行可能状態に復帰させるのは、ISR だということです。そのため VxWorks ではセマフォに対する操作のうち semGive は割り込みからも呼び出せるようになっています。一方待ちに入る可能性のある semTake は呼び出せません。

個々の説明だけではイメージが掴みにくいと思うので、一例として UART デバイスのドライバの動きの概要を説明します。UART デバイスも昔に比べるとバッファリングなど高度な機能を持つのが普通ですが、ここでは説明を単純化するためにごく単純な機能しか持たない UART デバイスのドライバとして説明します。
図 5 をもとにデバイスドライバの動作について説明します。

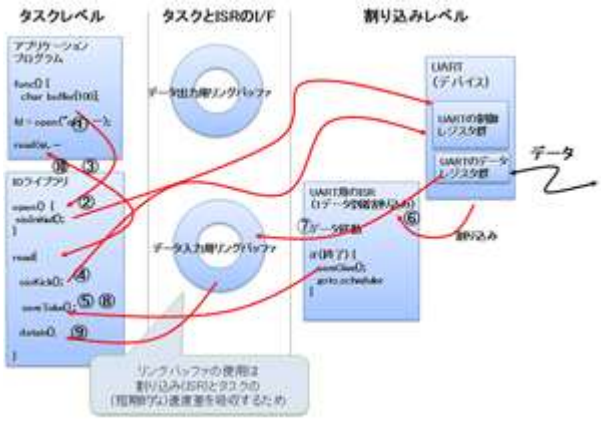

図 5 ドライバと RTOS

1. デバイスは使用される前に open しなければなりません。その処理を行います。open を行うことにより以後そのデバイスを特定するための識別子（通常ファイルディスクリプタ fd と呼ばれます）が帰ってくるので、以後の操作にはそれを用いて操作対象のデバイスを指定します。

2. アプリケーションの open によって呼び出され、実際のデバイスの初期設定を行います。その他、デバイスを制御したりデータの保管などに必要な領域があればそれをメモリ上に確保します。図におけるリングバッファの作成などです。
3. ファイルディスクリプタにより指定されたデバイスの動作（今回は読み込み）開始を指示します。
4. デバイスレジスタの動作開始ビットをオンにするなどデバイスハードウェアの動作を開始します。
5. その後ドライバの処理が終了するまでタスクは待ち状態となります。
6. デバイスによって定まっている何らかの動作の終了による割り込みが発生します。この割り込みとそれに対応した ISR の対応付けは（たとえば割り込みベクタの設定などは open 時あるいはそれより前の create 時など）何らかの適切なタイミングで行っておきます。
7. ISR ではデバイスのデータをタスクとのインターフェース領域（この場合はリングバッファ）にコピーします。またすべてのデータを受け取った時、あるいは図では省略していますが、デバイスエラー発生時には終了処理を行います。終了処理とは、待ち状態のタスクを実行可能状態に戻したのち、スケジューラーに制御を移すなどの処理です。
8. 優先度に応じいずれはタスクに制御が戻ってきます。
9. ISR が作成したデータをタスクが用意した領域（ここでは buffer）にコピーします。
10. 指定した領域にデータが用意された状態で、タスクが発行した read から戻ってきます。

    以後タスクは読み込んだデータを使用すればよいことになります。

以上がデバイスドライバの動作の一例です。

並列動作がありわかりにくいと思うので説明は省略しますが、少し別の観点からデバイスドライバの read 時の動作を図 6 として書いてみました。これも参考にしてみてください。
なお、この図に示した処理はあくまで一例であり、図に示した処理は最終データが来るまでは ISR が直接デバイスから次のデータを受け取るための処理を行い、スケジューラーを呼び出さないようになっていますが、ISR の最後に毎回スケジューラーを呼び出す場合のほうがむしろ一般的です。

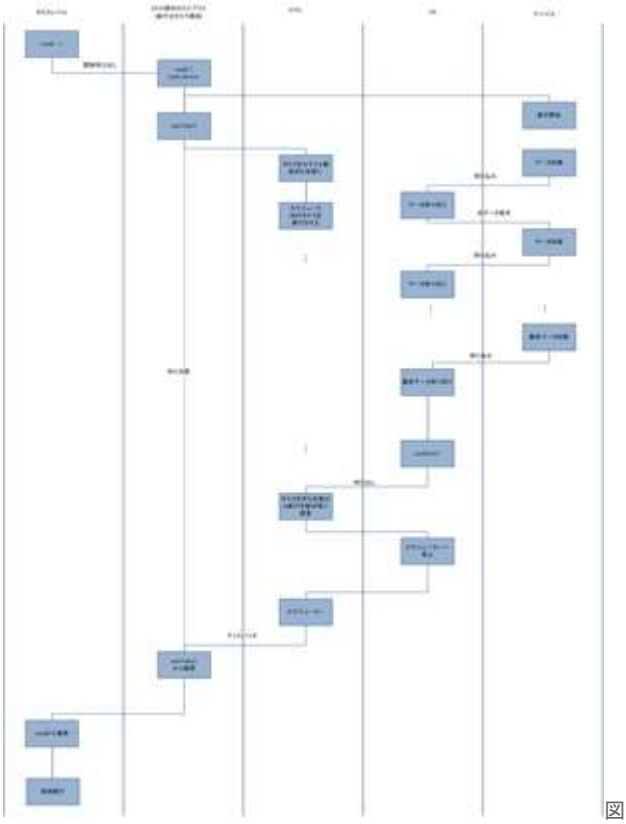

図 6 デバイスドライバの動作(read 時)

最初に組込みシステムにおいてはデバイスなどのハードウェアも含めた最適化が必要であり、その処理内容もデバイスによって様々です。
しかしデバイスドライバが標準 I/O で隠ぺい化されていることは、アプリケーション（タスク）からは非常に使いやすく、また固有のデバイスのドライバこそ自作しなければなりませんが、VxWorks には多くのデバイスのドライバはあらかじめ用意されています。
また使用するデバイスのドライバが提供されていない場合も、似たデバイスがある場合も多く、その場合は提供されているデバイスドライバを参考にすることもできます。
また個々のデバイスドライバを標準 I/O 化するための API やその他、デバイスドライバでも利用できるリングバッファやリストライブラリなども各種用意されており、デバイスドライバ作成を容易にするための工夫が随所に見られます。

この連載にも長い間お付き合いいただきましたが、次回は最終回です。
思いつくままに RTOS に対する希望などを書いてみたいと思います。

第 7 回おわり