第3章　OSなしで実現する 割り込みを使ったプログラミング

本章では，RTOSを使用しないで，割り込みハンドラとmain関数だけでプログラミングする手法について説明する．

最近では，RTOSを使った開発案件が増加している．しかし，RTOSを使用するには，ある程度の容量をもつRAMなどの十分なシステム資源を用意する必要がある．そのため，1チップ・マイコンなどの資源の少ない環境では，今後もRTOSを使用しない開発が続くだろう．

割り込みハンドラ(ISR：Interrupt Service Routine)とmain関数だけのプログラミングは往年の8080の時代を髣髴(ほうふつ)とさせる．しかし，そのころと比べればソフトウェアの技術はかなり進歩している．マイコンの世界を知ることは，RTOSのありがたみを理解するうえでも重要だ．この手法をきちんと理解して活用すれば，ブラックボックスを含まず，少ない資源で，きびきびと動くプログラムを作ることができる．

1　割り込み負荷の見積もり

●「割り込みが入る最小間隔」と「最悪応答時間」が重要

第2章の割り込み回路の項で説明したように，割り込み中に新たな割り込みが入った場合には，最初の割り込み要求が保留される．しかし，キューイングはされないので，割り込み処理中に2回以上割り込みが入っても，システムは1回分しか認識できない．ようするに，割り込みをロストする(取りこぼす)可能性がある．また，割り込み中に割り込みが入ると，割り込みハンドラを終了した途端に，再度割り込みハンドラの実行が始まるので，main関数側の処理ができなくなってしまい効率も良くない．つまり，割り込みハンドラが使用するCPU時間をある程度，予測したり，

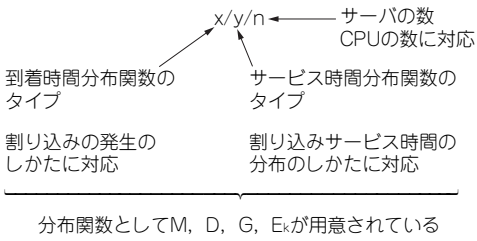

図1　待ち行列記号

制限したいという要求が出てくる．

組み込みシステムに対して，要求として与えられるのは，「割り込みが入る最小間隔」と「最悪応答時間」である．この二つは要求なのでとりあえず受け入れるとすると，ソフトウェアで対応可能なのは割り込みハンドラの実行時間になる．実行時間は短いほどよいのは当然なので，設計するうえで知りたいことは，逆にどの程度まで遅くできるかだ．

● 割り込みの各定数をおおまかに見積もることは可能

待ち行列理論を使うことで，割り込みハンドラの平均応答時間や多重割り込みになる確率を，マクロに見積もることができる．ここでのマクロの意味だが，マクロ経済などというときのマクロと同じである．大ざっぱと言うか，大きく全体を見たいときの手法である．開発の初期では，このマクロな視点しか使えない．要求として割り込み最小間隔が与えられると述べたが，実際は平均割り込み間隔しかわからない場合が多いようだ．待ち行列理論は，平均割り込み間隔がわかれば適用できる．

平均の割り込みサービス時間をt_{serv}，平均割り込み間隔をt_{arrive}，平均割り込み応答時間をt_{res}とすると，式(1)が成り立つ．

$$t_{res} = \frac{t_{serv}}{1 - \frac{t_{serv}}{t_{arrive}}} \quad \cdots\cdots (1)$$

たとえば，平均割り込み間隔が5ms，平均サービス時間が3msとすると，平均応答時間は7.5msになる．

$$\frac{3}{1-\frac{3}{5}} = 7.5$$

この値が妥当かどうかは後から考えるとして，多重割り込みになる確率を計算してみる．計算式は，

$$\left(\frac{t_{serv}}{t_{arrive}}\right)^2 \quad \cdots\cdots (2)$$

となることが知られている．これは，サービス中の顧客が二人以上になる確率で，実際に計算すると0.36になる．割り込みが三つ重なってロストしてしまう確率は，3乗すれば得られる．計算すると0.22となる．

多重割り込みの確率を1% にしたいとする．この場合，サービス時間の平均をいくつにすれば良いかは式(3)で計算できる．

$$t_{serv} \leq \sqrt{0.01 t_{arrive}^2} \quad \cdots\cdots (3)$$

実際に計算すると0.5msになる．念のため平均サービス時間を$t_{serv} = 0.5$として，応答時間を式(1)によって計算し直してみると0.56msになる．このようにして，とりあえず，0.5を割り込みハンドラ実行時間の設計目標とすることができる．このときのCPU使用率は，$0.5/5 = 0.1$になるので，CPU時間の10%を割り込み処理に使用することになる．CPU使用率は，実行時間t_{exe}と起動周期Tを使って式(4)によって計算される値である．これはCPUの負荷を見積もる基本的な量になる．

$$U = \frac{t_{exe}}{T} \quad \cdots\cdots (4)$$

以上のことから，もともと3msと見積もった処理を，実行時間0.5msの割り込みハンドラと，実行時間2.5msのmain関数で処理すれば，割り込みの取りこぼしを防げるだろうと予測できる．また，この条件で，

$$\frac{t_{serv}}{t_{arrive} - t_{serv}} \quad \cdots\cdots (5)$$

を計算すると，系の中の処理すべき割り込みの平均数になる．平均数を実際に計算すると1.5になる．これは，割り込みハンドラとmain関数の間にバッファを入れなければならないことを意味している．つまり，1.5個の割り込みデータがmain関数に渡されずに溜まってしまう可能性があるので，バッファ・サイズは2にしなければならないこともわかる．

● ポアッソン到着――割り込みがポアッソン分布で発生

これらの計算は，M/M/1という待ち行列のモデルに基づいている．M/M/1の意味は，割り込みがポアッソン分布で発生し(M)，それを指数サービス時間内に(M)，一つの(1)CPUで処理するということだ．CPUが一つというのは問題ないが，残りの二つの意味を吟味する必要がある．

ポアッソン到着とは，割り込みの発生はランダムだが，単位時間に発生する平均数はわかっているような到着パターンのことである(**図1**)．割り込み密度だけ

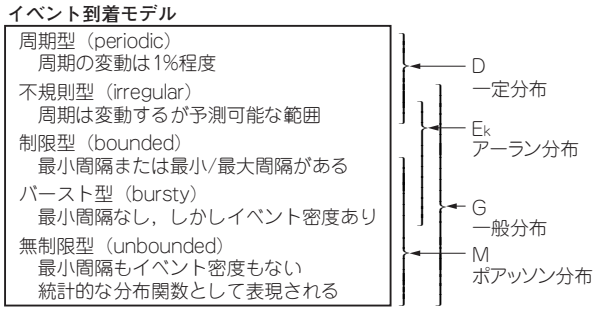

図2　イベント到着モデルの位置付け

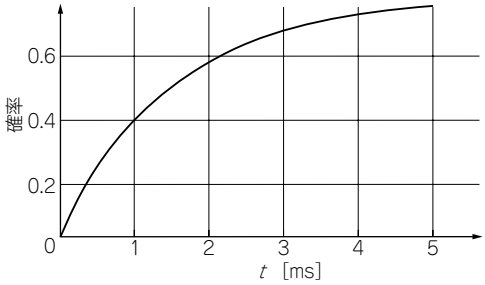

図3　tms以内に終了する確率

が決まっていて，いつ割り込まれるかまったく予測できない場合だ．自然界にはよくある分布で，1日に発生する交通事故件数，不良品の個数，放射性元素の崩壊数などがこれに当てはまる．イベント到着モデルとしては，周期的に発生するもの以外にも，不規則に発生するもの，バースト的に短時間にたくさん発生するものなどがある．このようにさまざまなイベント到着モデルの中でのポアッソン到着モデルの位置付けを**図2**に示す．

平均して1秒間に2回の割り込みが発生する場合，1秒間に4回割り込みが起こる確率はどれくらいか，と聞かれたとする．この確率は，ポアッソン分布を前提とすれば0.09になる．この値はExcelでもPOISSON(4,2,FALSE)で計算することができる．最後のFALSEは，ちょうど4回起こるということを指定している．ちなみに，TRUEにすると，0回から4回起こる確率として0.95が計算される．

指数サービス時間とは，処理時間の平均は3msだが，ランダムに終了するようなサービス時間の分布だ．ランダムなので0.001msで終了するかもしれないが，平均すると3msになるような分布である．指数分布は，すぐに終わる確率がいちばん高いので，割り込み処理時間へ適用するには無理があるかもしれない．割り込みを受け付けてからt時間後までに終了する確率が，

$$1-e^{-\mu t}$$

になるようなサービスだ．ここで，μは平均サービス時間の逆数となる．具体的にいうと，μは単位時間のサービス回数である．この場合は$\mu = 1/3$になるので，3ms以内に終了する確率を計算すると，0.63になる．0.001ms以内に終了する確率は0.0003だ．これは少ない数値だが，0ではない．これをグラフにすると**図3**のようになる．この値もExcelでEXPONDIST(3,1/3,TRUE)と計算することができる．

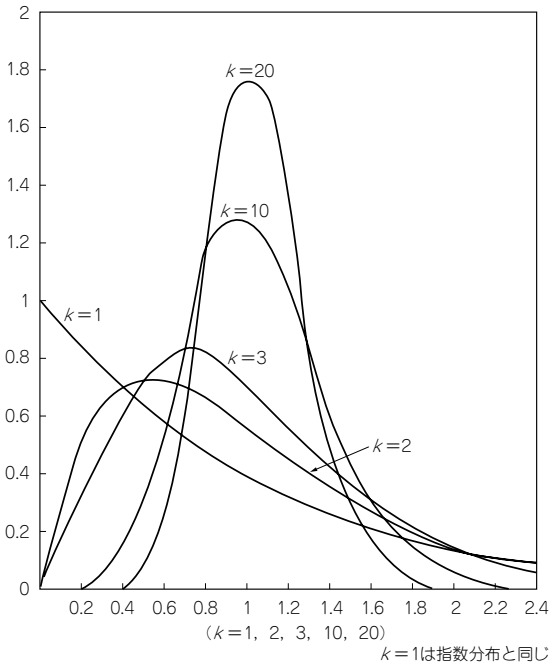

図4　アーラン分布――複数の指数分布を加え合わせたもの

処理時間の分布に指数分布を使えないと思える場合には（**図3**を見ればたいていだれでもそう思う），指数分布を複数加え合わせたアーラン分布（**図4**）や，分布の形にはこだわらずに平均値と標準偏差だけを使う一般分布などを使用する．実行時間に関するキャッシュなどの確率的要素がまったくなければ，標準偏差が0の一定分布を使用する．それぞれの分布に対しては，同様の計算を行うことができる．

M/M/1待ち行列モデルは，個別の割り込みに対して適用するには無理がある．しかし，計算が非常に簡単なので，システム全体の負荷を見積もるときに使用する．つまり，システムが処理する割り込みが全部で20あれば，それらの平均発生間隔から多重割り込み

の確率を設定して，それを満足する割り込みハンドラ全体の平均サービス時間を求められる．あるいは，すべての割り込みではなく，非周期割り込みだけを対象として負荷を計算するのにも利用できる．開発初期の，平均値ベースでしか議論ができない条件でも，アーキテクチャ決定に対する指針を得ることができるのだ．

M/M/1以外の待ち行列モデルは開発が進んで，割り込みごとの統計的性質や，処理時間に関する情報が使えるようになった段階で使用すればよい．

2 割り込みにおけるデータ共有時の問題

● 割り込みハンドラで並行処理が可能になる

スタートアップ・ルーチンから起動されたmain関数と，外部割り込みやタイマで起動される割り込みハンドラは，並行に実行される（**図5**）．つまり，RTOSを使用しなくても並行処理ができる．そして並行実行が行われるとかならず問題になるのが，データ共有である．

必要な処理をmain関数側と分担せずに，すべてを割り込みハンドラだけで処理するのなら，データを共有せずともよいだろう．しかし，この方法では，割り込みをロストする可能性が大きくなる．また，割り込み遅延時間を長くしないために，割り込みハンドラは最小限の処理のみを割り込み状態で行って，すぐに制御をmain関数に返すようにしなければならない．

● 割り込みのタイミングによっては正しいデータが得られなくなる――データ共有問題

一方，デバイス側から見た場合，デバイスが割り込み要求を行うのは，何らかの動作が終了した場合である．そして，その結果の入力データなどを早くCPUに渡して，次の動作を開始する必要に迫られている場合も多い．そのため，割り込みハンドラは，早く入力データを受け取ってやらなければならない．したがって，割り込みハンドラが行う最低限の処理とは，デバイスを開放して，次の割り込みを受けられるようにすることである．

たとえば，**図6**のように二つのデータdata1とdata2を，割り込みハンドラでデバイスから取り込んでmain関数で処理する場合を考える．データの変化があると割り込みが発生し，data1とdata2が更新されるとする．A-D変換終了で割り込みが入るとか，タイマ割り込みでデータを取りにいく場合などが考えられる．

ただし，データにはdata1 + data2 = 1という制御モデル上の制約があるとする．この制約は，1回の測定では保証されるが，別々の測定タイミングでは保証されないとする．たとえば，身長と体重から肥満度を計算するには，同時に測定した身長と体重でなければならない．main関数側でdata1とdata2を参照している間に割り込みが入ると，main関数では古いdata1と新しいdata2を使って計算することになり，制御モデル上の前提条件が保証されなくなってしまう．つまり，今年の身長と去年の体重で肥満度を計算するようなバグの原因となる．data1とdata2を使用する計算を，すべて割り込みハンドラで行っていれば発生することのない問題が発生してしまう．これは計算式がまちがっているわけではないので，ターゲット上で割り込みを入れながらテストしないと発見できないバグである．

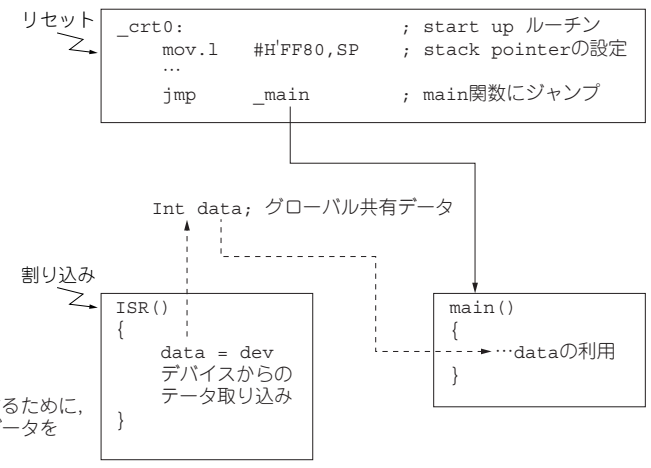

図5
main関数と割り込みハンドラは並行に実行される
ISRの実行時間を短くするために，ISRではデバイスからデータを取り込むだけにする

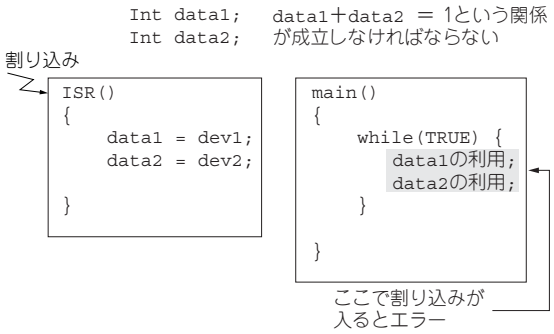

図6　データ共有問題――割り込みのタイミングによって正しいデータが得られない

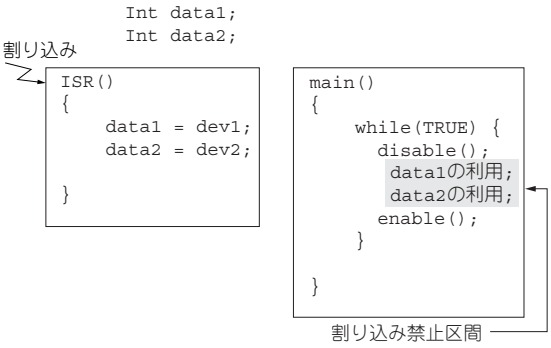

図7　割り込み禁止による解決

この解決策だが，割り込みそのものを禁止すればよい(割り込みがNMIでない場合)．組み込み用コンパイラは，割り込みを禁止・許可するライブラリ・ルーチンをもっているので，それを挿入すればよい．ただし，**図7**のようにdata1とdata2の利用部分を丸ごと割り込み禁止にすると，割り込み遅延時間が大きくなって，何のために計算ルーチンをmain関数に移動したのかわからなくなってしまう．これでは割り込みハンドラの中でやっているのと同じになってしまう．

割り込みハンドラの実行時間をなるべく短くしたのと同様に，割り込み禁止時間もできるだけ短くしなければならないので，**図8**のようにする．

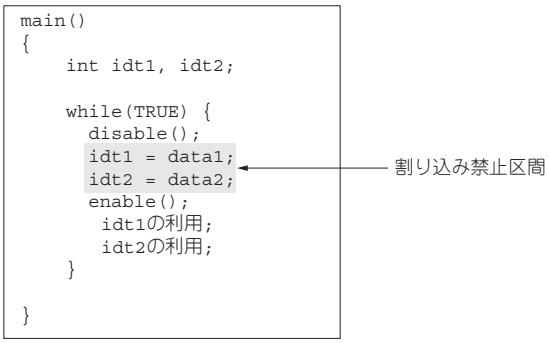

図8　割り込み禁止区間は短くする

● 割り込みのネスト時の問題を解決する

main関数は，複数の仕事をしなければならないため，個別の処理はサブルーチン化することが多くある．このとき注意しなければならないのは，割り込み禁止区間がネストする場合だ．main側が複雑になって，モジュール化が進むと関数が細分化されるため，割り込み禁止区間から別の関数を呼び出すことがある．そうすると**図9**に示すように，内側の割り込み禁止区間を抜ける際に割り込み禁止が解除されてしまう．

これを避けるためには，disable()の返り値をチェックして，失敗した場合はenable()を呼び出さないようにする必要がある(**図10**)．disable()をアセンブラで自作してCから呼び出すときには，実際に設定されたステータス・レジスタの値を返すような仕様にする方法がある．あるいは，disable()とenable()をクラスにまとめてネスト状況をモニタする方法もある．本当のエラーと，すでに割り込み禁止になっていた場合とは区別する．

● 割り込み禁止を使用しない方法：リトライ

割り込みが入るのは，現在実行中のCPU命令が終

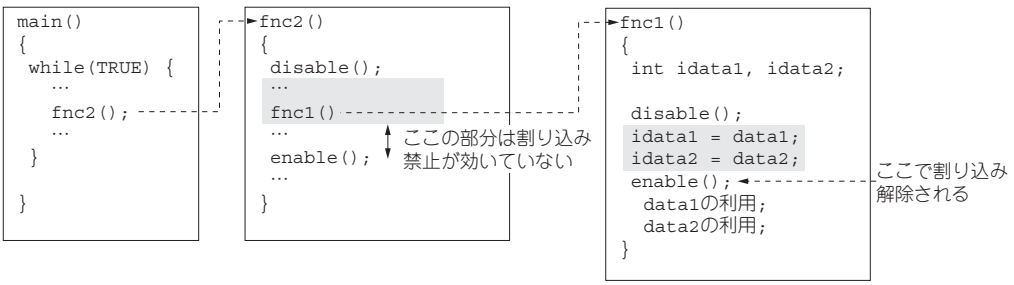

図9　関数化する場合の注意

```
fnc1()
{
    status = disable();
    …
    if (status == OK) enable();
    …
}
```

図10　ネスト対策――disable()の返り値をチェック

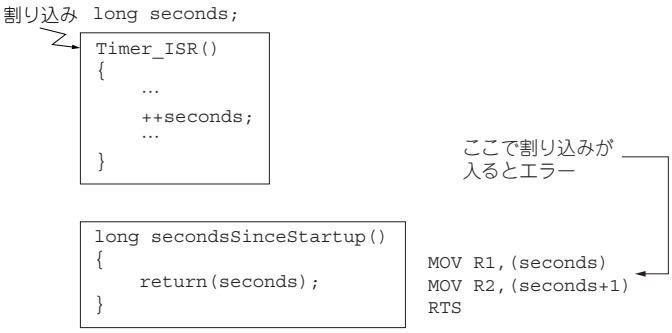

図11　秒数をカウントする場合

```
volatile long seconds;
```

```
long secondsSinceStartup(void)
{
    long work;

    work = seconds;
    while (work != seconds)
        work = seconds;
    return(work);
}
```

図12　割り込み禁止を使わない方法：リトライ

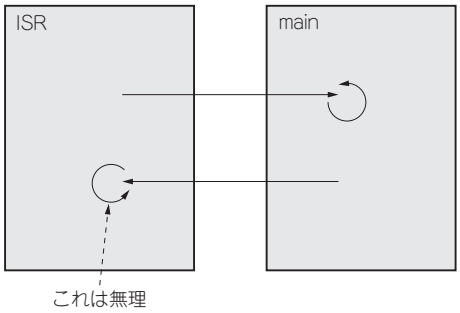

図13　ISRでリトライは無理

了してからなので，1命令で記述できる処理であれば割り込み禁止にする必要はない．16ビットCPUで16ビット整数を共有する場合は問題ない．

しかし，16ビットCPUで32ビット整数を共有する場合には，データ共有の問題が発生してしまう．たとえば，1秒ごとに割り込みが入り，それをカウントしている場合，16ビットCPUではlongを使用すると，**図11**のようにすることはできない(longが32ビットの場合)．とは言っても，これだけのために割り込み禁止の関数呼び出しを使うのも効率が悪いだろう．CPUによっては，確実に割り込み禁止や解除になるまでに数マシン・サイクルを要するものもあり，あまり短いと本当はどこが割り込み禁止なのかがわからないこともある．ほとんどのCPUで，割り込み禁止はすぐに有効になるが，割り込み解除に何サイクルかかかることも珍しくないようだ．

このような場合は，**図12**のように，リトライ・ベースの方法を使うことができる．この方法であれば，NMIルーチンとのデータ共有も可能だ．ただし，コンパイラが本当にメモリを読むようにvolatile宣言する必要がある．volatileを付けないと，最初の代入文で汎用レジスタに入れた値を使ったり，whileループ自身をなかったことにしてしまうことがある．volatileを付けても，念のためにアセンブラ・ソースで確認したほうが良いだろう．

注意しなければならないのは，リトライできるのはプライオリティの低いほうだけということだ．ISRが動いている限りmain側は動けないので，ISR側で何回リトライしても，同じ結果になり，そのまま無限ループになってしまう(**図13**)．これと似た方法として，RTOS管理下のタスク間でも使用可能なノンブロッキング・プロトコル(第8章)を使用するということがあげられる．割り込み禁止を使用できるのであれば，ロックフリー・プロトコル(第8章)も使用できる．

● バッファを使った共有データ保護

そのほかの手法としては，**図14**～**図17**のようにバッファを使用することで共有データを保護することができる．バッファを入れることで，割り込み処理以降の処理も含めた挙動は，待ち行列理論が扱う確率的なものになる．

バッファ・サイズも待ち行列から決めることができる．一つの共有変数で割り込み禁止を使用するか，ダブル・バッファにするか，リング・バッファにするか，という選択肢があるわけだ．まず，要求される処理速