第1章

組み込みプログラミングの基礎知識

――スタートアップ・ルーチンから割り込みまで

坂本直史，二上貴夫，三浦 元，山崎辰雄

組み込みソフトウェアには，IT(infomation technology)系ソフトウェアにはない考えかたが存在する．ここでは，「組み込み」ならではのいくつかの概念について解説する．すなわち，プログラムのランタイム構造やスタートアップ・ルーチン，配列の実体，ビット演算，volatile，同期/非同期呼び出し，割り込みなどを説明する．（編集部）

1．組み込みソフトウェアとは何だろう

皆さんの身の周りにあるさまざまな機器，例えば家電製品や携帯電話に始まり，自動車，工場で使う産業機器などの内部に格納されている（組み込まれている）ソフトウェアを総称して「組み込みソフトウェア」と呼びます．組み込みソフトウェアは，それらの機器を動作させ制御している，縁の下の力持ち（または影の支配者）です．**図1**を見てもわかるように，組み込みソフトウェアは現代社会を支える重要な技術です．

もう少し詳しく見ると，組み込みソフトウェアは「機器に組み込まれ，ハードウェアを制御する専用のソフトウェア」であるという特徴があります．各種機器に使われているハードウェア（マイコンやメモリ，論理IC，入出力装置など）は，製品やメーカ，生産時点で入手できる部品，価格などによって異なります（同一製品でも，バージョンによって内部のボードに搭載されるハードウェアが変わっていることがよくある）．そのため，組み込みソフトウェアは各製品ごとの専用ソフトウェア（一品もの）です．そして，組み込みソフトウェアはハードウェアと密接に連携しながら，ハードウェアと一体になって製品の要求仕様を実現します．

一方，皆さんが日ごろパソコン上で使っている業務ソフトウェアや業務の自動化・解析ソフトウェアを総称して「IT（infomation technology）系ソフトウェア」と呼びます．大半のIT系ソフトウェアは，情報技

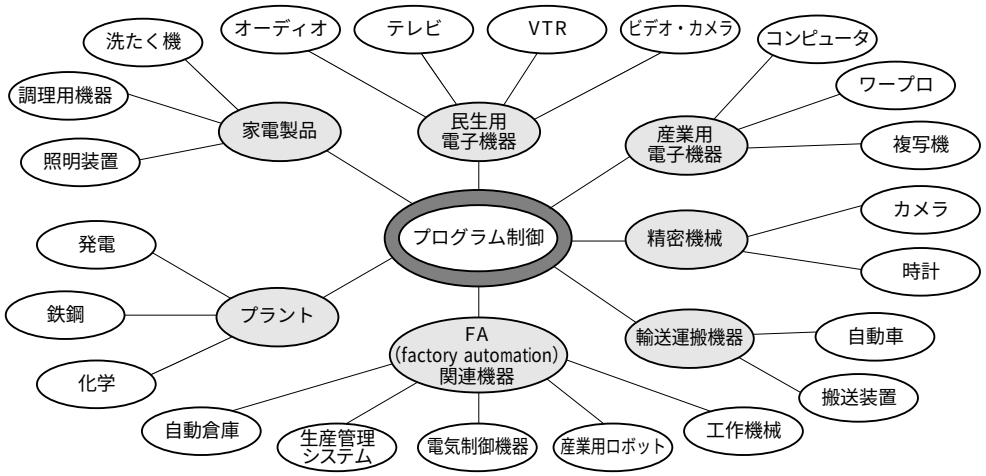


図1　組み込みソフトウェアが用いられている主要分野
（出典：特許庁のホームページ，http://www.jpo.go.jp/shiryou/のプログラム制御技術に関する技術分野別特許マップ 図1.1.2-1）
主要分野を見渡すだけでも，幅が広いことがわかる．

術に特化した（つまり，自然現象などを直接扱うことがない）ソフトウェアです[注1]．文書作成ソフトウェアや会社の業務処理関連（伝票処理，受発注処理など）のソフトウェアがこれにあたります．中でも業務処理関連のソフトウェアを「ビジネス・アプリケーション・ソフトウェア」と呼びます．

● 四つの基本要件を押さえよう

組み込みソフトウェアは，次の四つの基本要件を満たす必要があります（**図2**）．

1）自然法則

組み込みソフトウェアは論理だけで完結する世界を相手にしているわけではありません．組み込まれる製品には，風や水といった自然界のものを利用する機器や，落雷による電力の瞬停という突発的な状況を考慮したうえで安定して動作させなければならない機器もあります．あるいは「わがままでうるさいようだが，気があちこちに分散していて意外とアバウトな生き物（人間）」の相手をする機器（家庭用ロボットなど）もあります．このように，組み込みソフトウェアではいろいろな自然現象を扱います．そのため，プログラムが難しくなる面もありますが，その機器の用途と自然の特性を限定して考えると，汎用を想定するよりもモデルをシンプルにできて簡潔なプログラムになる場合もあります．このように，「自然法則を考える」というのが組み込みソフトウェアの特徴ある要件の一つです．

2）リアルタイム制御

一般的に組み込みソフトウェアでは，搭載される製品の仕様により，プログラムの中で実行する処理に

注1：表層部はIT系であっても，内部にはリアルタイム制御を行うための組み込みソフトウェアが使われている場合もよくある．ルータなどがこれにあたる．

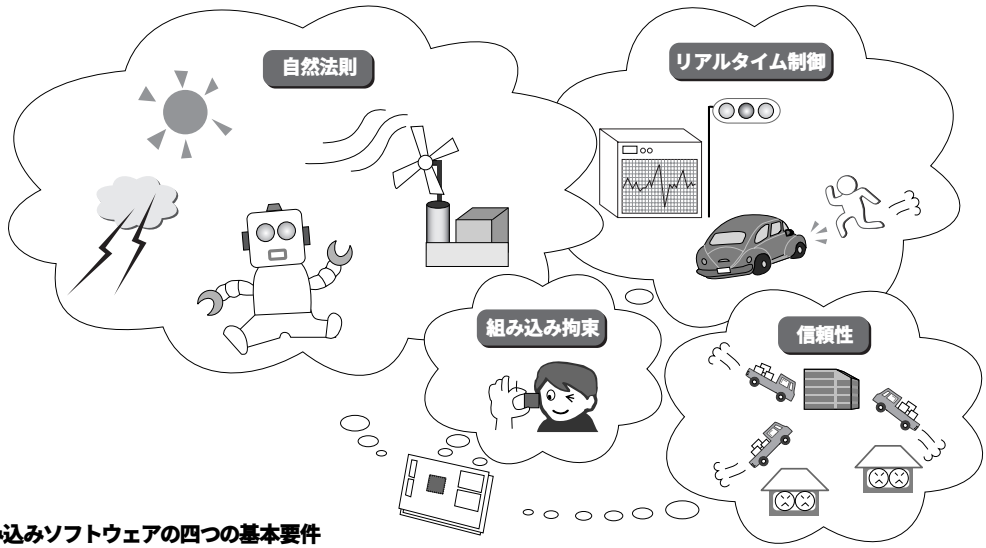


図2
組み込みソフトウェアの四つの基本要件

一定の時間的制約が課せられます．その製品を使う人が操作したときに処理を待ってくれなかったり，一定時間内にデータを読み込みにいかなければ欲しいデータがなくなってしまうからです．この時間的制約を守ってソフトウェアを作ることをリアルタイム制御と言います．例えばμs（マイクロ秒）の範囲まで規定され，それらの厳しい制約をかならず守らなければならないもの（ハード・リアルタイムと呼ぶ）や，比較的緩やかな制約のもの（ソフト・リアルタイムと呼ぶ）など，さまざまな応用があります．

3）組み込み拘束

組み込み機器はコストを抑えるために必要最小限のハードウェア・リソースしか搭載されていないのが普通です．また，ハードウェアは，いったん設計すると修正するのに費用と時間がかかるため，問題が発生した場合，ソフトウェアで解決できないかどうかを検討する傾向にあります．そのため，「ソフトウェアを設計したがメモリが足りない」という状況になっても，メモリ増設という対策は最終手段で，なんとか現状のメモリで対応するようにソフトウェア設計を見直すことになります．CPU性能が足りない場合も同じです．与えられた資源の範囲で（リアルタイム制御を含め）いろいろな要求を実現する，そのために，自然法則の検討も含めたあらゆる創意くふうが必要になります．

4）信頼性

組み込みソフトウェアには高い品質と信頼性が求められます．それは，機器に搭載されたソフトウェアにバグがあると，例えば人命に影響を与えるような大きな影響を及ぼす可能性があることや，ソフトウェアを更新するためには，いわゆる製品のリコールを実施しなくてはならず，リコールの費用はもちろんのこと，製品や会社のイメージに大きな影響を与える可能性があることなどの理由によります．組み込みソフトウェアの場合，「バグが発覚したらユーザに連絡して，ソフトウェアをアップデートしてもらおう」と

いう方法はたいていの場合使えません（これはIT系ソフトウェアの悪しき慣習）．ご存じのとおり，すべての機器がネットワークに接続できるわけでもなく，CD-ROMなどのメディアと接続できるわけでもありません．第一，ソフトウェアがROM（read only memory）に書かれていれば，変更のしようがありません．

● 開発環境について

組み込みソフトウェアの開発は，パソコン上で動くソフトウェアの開発とは若干異なります．パソコン上で動くソフトウェアを開発する場合，ソース・コードを書いて，開発ツール（コンパイラなど）でコンパイルし，リンクすれば，生成されたソフトウェアはそのままパソコン上で動作します[注2]．組み込みソフトウェアの場合，動作させたいターゲット（各種マイコン，ASICなど）の上で動作するソフトウェアを，パソコン上で開発します[注3]．このような開発には，クロス・ツールと呼ばれるコンパイラやシミュレータ，エミュレータなどを使用する必要があります（**図3**）．

以前と比べるとシミュレータやエミュレータのGUI（graphical user interface）は格段に良くなり，パソコンのソフトウェアを開発する感覚で使用できるようになっていますが，動作させるための環境のセットアップはたいへんなことが多く，たんに「C言語でプログラムを書いたから，すぐリンクしてデバッグする」というわけにはいきません．また，実際にプログラムを動作させるためには，マイコンやASICに含まれる周辺部を，定められた手続きで初期化する必要があります．この作業を行えないと，組み込みソフトウェア開発には手も足も出ません．

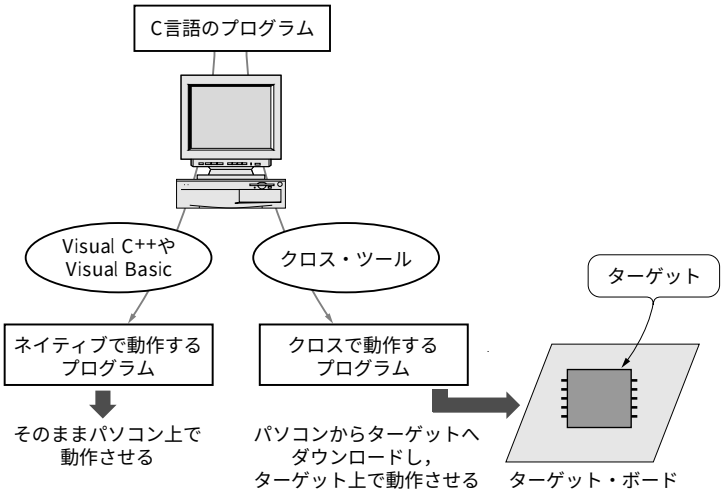


図3
組み込みソフトウェアの開発
パソコン上で動くソフトウェアの開発では，ソース・コードを書いて，開発ツール（Visual C++，Visual Basicなど）でビルドすれば，生成したソフトウェア（ネイティブ・ソフトウェア）はそのまま動作する．組み込みソフトウェアの場合，クロス・ツールを利用して，動作させたいターゲット上で動作するソフトウェアをパソコン上で開発する．その後，シリアル通信などを利用して，生成したソフトウェア（クロス・ソフトウェア）をターゲット・ボードにダウンロードし，プログラムを動作させる．

注2：このようなソフトウェアを「ネイティブ・ソフトウェア」と呼ぶ．
注3：このようなソフトウェアを「クロス・ソフトウェア」と呼ぶ．また，クロス・ソフトウェアを開発する環境（ツール）をクロス環境（クロス・ツール）と呼ぶ．

● プログラムが動くための基本メカニズムを知る

前述した四つの基本要件を満たす組み込みソフトウェアを作るためには，IT系のソフトウェアを作れる技術や経験だけでは（ましてや大学でC言語の講義を受けただけでは）足りません．例えば，Cプログラムのmain関数が終了してしまうプログラムを書いて何が悪いのかわからない[注4]ITプログラマはどこにでもいます（これは組み込み拘束の常識問題の一つ）が，それでは困るわけです．以降の節では，皆さんが組み込みソフトウェア技術者として活躍するために，まず押さえておいていただきたいプログラムの動作メカニズムについて解説します．

市販されているC言語に関する本の内容は，ほとんどがC言語の文法や使いかたの解説です．WindowsやLinux/UNIX上でプログラミングするだけならそれで十分でしょう．例えば，LinuxやUNIXならば，おまじないのように，

```
% cc example.c
```

とコマンド入力すれば，ファイルexample.cに書かれたC言語のプログラムがコンパイルされ，a.outという名まえの実行可能なファイルが生成されます．そして，

```
% ./a.out
```

とコマンド入力すれば，ファイルexample.cに書いておいたプログラムが（あわよくば意図したとおりに）実行されます．この手順は，使用するOS環境やコンパイラなどによって若干変わりますが，WindowsやLinux/UNIXの環境ならば，基本的にはmain関数以降のプログラムを書き，そのファイルをコンパイルして実行するだけです．プログラムを動かすために，プログラムの起動や動作時のしくみなどを気にする必要はありません．

ところが，組み込みシステムの場合はそうはいきません．つまり，魔法のようなccコマンドはないのです．すでに実機でプログラムを動かした経験があって，「普通にコンパイルしてプログラムが動いたよ」という方もおられるかもしれません．でもよく思い出してください．先輩から「コンパイル環境」をもらって，コンパイル対象のファイル名を書き換えただけではないですか？ そのコンパイル環境に，ccコマンドのような秘密が隠されているのです．その秘密を理解しないことには，いつまでたっても一人前の技術者にはなれません．

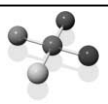
また，さらに経験を積むうちに，「プログラムの動作がおかしい．実行するたびに動作が異なっているため，どうしてよいかわからない」とか，「ソース・レベルのデバッグができない」といった場面に出くわすことでしょう．こうしたときのために，プログラムの動作メカニズムを理解しておくことはきわめて重要です．

注4：main関数が終了すると，機器が制御不能になる．

組み込み機器は，例えばユーザがボタンを押したといったようなON/OFFの情報や，光や音などのアナログ的な情報を取り込みながら動作しています．それらのデータは，通常，マイコンの周辺機能を使って取り込まれ，プログラムはシリアル・インターフェースやポート，あるいは割り込み信号を介してデータを受け取ります．このようなデータの取り込みは，ある程度ハードウェアの知識を持っていると，ぐっと理解しやすくなります．

さかもと・ただし
ふたがみ・たかお
組込みソフトウェア管理者・技術者育成研究会（SESSAME）

2．プログラムはどのように動くのか



プログラムがどういったしくみで動いているかということ（プログラムのランタイム構造）は，一般のC言語の解説書には書かれていません．それは，WindowsやLinux/UNIXといったOS上で動くプログラムを開発する場合は，ランタイム構造はOS側で決まっており，C言語を使ってプログラミングする人はそれを気にしなくてもいいからです．

しかし，組み込みソフトウェアの開発ではそうはいきません．それは，使用できるメモリ量に制限がある（つまりメモリ量が少ない）のと，デバッグ環境がかならずしもWindowsやLinux/UNIXのときのように整っているとは限らないからです．

● 例：再帰呼び出しのプログラム

プログラムのランタイム構造を説明するために，まずは階乗計算を考えます．自然数nの階乗$n!$は，

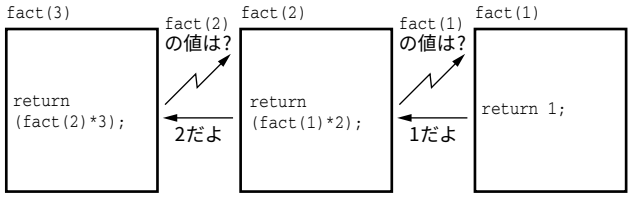
$$n! = n \times (n-1) \times \cdots \times 2 \times 1 \quad \cdots\cdots(1)$$

と計算します．また，$(n-1) \times \cdots \times 2 \times 1$とは$(n-1)$の階乗のことですから，

$$n! = n \times (n-1)! \quad \cdots\cdots(2)$$

図4
fact(3)の計算過程

fact(3)の計算にはfact(2)の値が必要で，fact(2)の計算にはfact(1)の値が必要である．fact(1)の値は1と決まるので，これによってfact(2)の値は2，fact(3)の値は6と決まる．計算過程は，fact(3)→fact(2)→fact(1)→fact(2)→fact(3)となる．



とも書けます．さて，nの階乗を計算する関数factをコーディングしてみましょう．ここでは式(2)を使ってコーディングします．

```
int fact(int n)
{
    if (n == 1)
        return 1;
    return(fact(n-1) * n);
}
```

関数factの中で自分自身を呼び出していますね．これを再帰呼び出しと言います[注5]．

例えば，fact(3)の計算は次のように行われます．

fact(3)： 3==1 ではないのでfact(2) * 3を返す．そのためにfact(2)を計算する
　fact(2)： 2==1 ではないのでfact(1) * 2を返す．そのためにfact(1)を計算する
　　fact(1)： 1==1 なので1を返す
　fact(2)： fact(1)の値が1だったので，fact(1) * 2 = 1 * 2 = 2を返す
fact(3)： fact(2)の値が2だったので，fact(2) * 3 = 2 * 3 = 6を返す

つまり，fact(3)を計算するためには，fact(2)とfact(1)の計算が必要になります．計算過程を**図4**に示します．これを見ると，まるでfact(1)，fact(2)，fact(3)という関数factのコピーが三つあるようです．さて，この関数factが実際にどのように実現されているか，わかりますか？

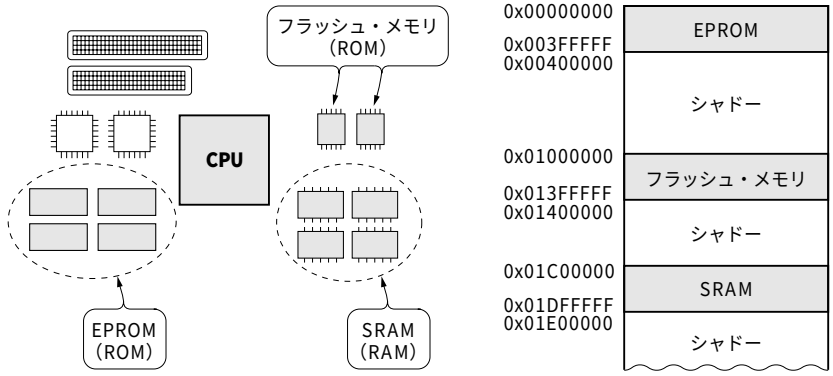
● プログラムはコード部と変数部に分けて格納される

プログラムがコンパイルされると，プログラムの基本となる制御部分はコードと呼ばれる機械語に変換されます．コードは，通常，プログラムの実行中に書き換えられることはありません．このコードを読みながらCPUはプログラムを実行していきます．

一般的に，プログラムの実行には，コード以外に定数と変数が必要になります．コンパイル時に初期値が決まっていて，その後いっさい変更のないものが定数で，プログラムの実行中に値が変更されるものが変数です．コードと定数，変数は，メモリの内部に格納します．その内容が変わることのないコードと定数(以下ではまとめてコード部と呼ぶ)はROMに格納しておけます．一方，実行中に値が変わる変数(変数部と呼ぶ)はRAM(random access memory)に格納する必要があります(**図5(a)**)．

先ほど「メモリの内部に格納する」と書きましたが，メモリ領域内にプログラム(コード部および変数部)を配置するための論理的な場所を「メモリ空間」と呼びます．メモリ空間の実体は，マイコンの内蔵メモリやマイコン外部にあるRAMやROMに存在します．メモリ空間には，通常，1バイト(＝8ビット)ごとに

注5：再帰呼び出しを使うとプログラムをエレガントに書けるが，組み込みソフトウェアでは通常は使わない(関数factの例なら，式(1)をwhile文を使って書く)．ただし，ここではあえて再帰呼び出しを例にしている．



(a) CPUとメモリの構成例　　(b) メモリ空間へのメモリ配置

図5
メモリとメモリ空間
実際のメモリがメモリ空間にどのようにマッピングされているのかは，ボード上の構成(CPUやメモリ，アドレス・デコーダなどの部品構成)によって異なる．

番地(アドレス)が割り振られ[注6]，それによってユニーク(一意)にどこのメモリかを指定できます[注7]．メモリ空間の番地は，例えば32ビットを使って表現すれば，0x00000000番地から0xFFFFFFFF番地まで割り振ることができますが，実際に使用する際に，このすべてのアドレスが使えるわけではありません．アドレスに対応するメモリが実装されている部分のみ使えることになります．**図5(b)**の例では，0x00000000番地～0x003FFFFF番地，0x01000000番地～013FFFFF番地，0x01C00000番地～0x01DFFFFF番地が使用できることになります．

さて，ランタイム構造の話に戻りましょう．fact(3)の計算過程を見ていると，関数factのコピーが三つあるようでした．すると，factの計算では再帰呼び出しが起こるたびに，コード部と変数部を動的に(その場その場で)作って実行しているのでしょうか？

残念ながらそうではありません．関数factは小さな関数なのでそのコード部をコピーしてもたいしたことはありませんが，一般にコード部のサイズが大きいと考えると，この方法はあまり効率的ではありません．実際には，**図6**に示すように変数部のみを動的に作って実行しています．三つの変数部のメモリを確保する部分はスタックと呼ばれるメモリ空間(RAM)上の領域です．スタックは，組み込みソフトウェアを開発している間はずっとつきあうことになるものです．

● スタックへのアクセスにはフレーム・ポインタを利用

スタックはデータを0番地方向に積んでいくのが習わしで，スタックの先頭位置(スタックをどこまで使っているのか)は「スタック・ポインタ」と呼ばれるCPUのレジスタで管理されています．スタックにデータを積むことをpushと呼び，スタックからデータを取り出すことをpopと呼びます(**図7**)．データをpop

注6：例えば，0x00000000番地が最初の1バイトの領域を指し，0x00000001番地が次の1バイトの領域を指す，といったぐあいになる．なお，これはバイト・マシンの場合であり，ワード・マシンの場合は1ワードごとに番地が割り振られる．詳しくは，p.90のコラム「バイト・マシンとワード・マシン」を参照のこと．
注7：なお，MMU(memory management unit)を搭載するマイコンでは実メモリ空間のほかに仮想メモリ空間があり，話はさらに複雑になる．ここでは説明を簡単にするために，MMUのないマイコンを想定して解説している．

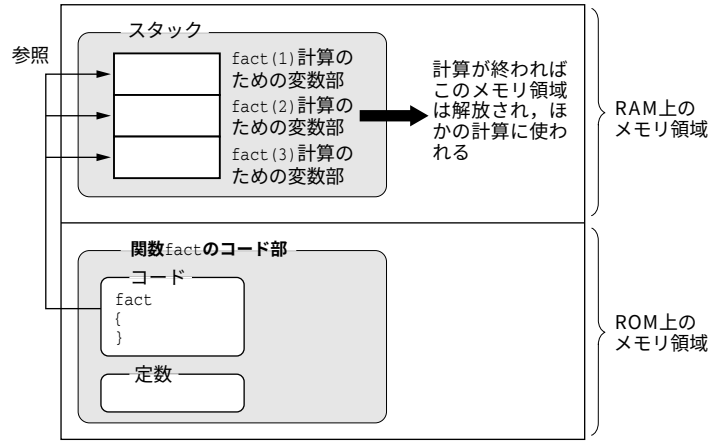


図6
関数factの変数部はスタック内に確保される

関数 fact のコード部は一つだけ存在する．関数 fact に異なる引き数が与えられるたびに，変数部用のメモリ領域がスタック内に確保される．計算が終わればこのメモリ領域は解放される．

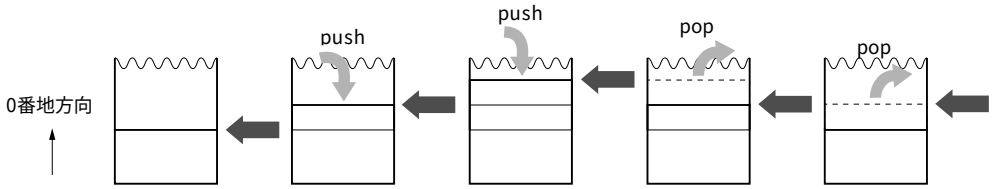


図7　スタックのpushとpop

黒矢印（⬅）で示されたスタックの先頭位置は，つねにスタック・ポインタで管理されている．スタックに積まれた途中のデータはpopできない．

するとスタックの先頭位置が変わり，pop済みの領域にアクセスできなくなるので，実質的にそのメモリ領域は解放されたこと（再利用可能）になります．

では，変数部をスタックに確保しながら具体的にどのようにプログラムが動いていくのかを説明します．関数 fact の実行に必要な変数部をスタックに積んでみましょう．ここでは，関数 fact の引き数はこの変数部に格納されて呼び出されることとします（実際には，関数の引き数をどう渡すかはコンパイラに依存する）．さて，プログラムがこの関数 fact の計算を行うには，引き数のほかにどのような情報が必要でしょうか？

プログラムは，fact(1) の計算をするために fact(1) の変数部に着目して動きますが，fact(1) の計算が終わったら，fact(1) を呼び出した fact(2) の変数部に着目して動く必要があります．そのための情報，つまり，fact(2) の変数部の始まる場所（これをベース・アドレスと呼ぶ）を知っている必要があります．具体的には fact(2) の変数部のうち，いちばん下のアドレスがベース・アドレスになります（**図8**）．

ベース・アドレスの切り替えを統一的に取り扱うために，今着目している変数部のベース・アドレスを格納するポインタ（これをフレーム・ポインタと呼ぶ）を作っておきます．プログラムのコードが変数部のデータにアクセスする際には，このフレーム・ポインタからの相対位置（フレーム・ポインタと変数部のオ

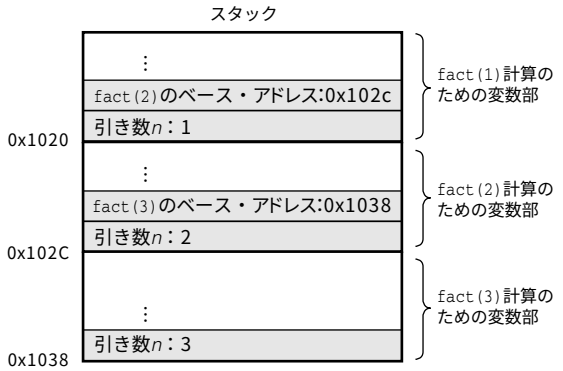


図8
fact(1)を計算中のスタックのようす
fact(1)を計算するための変数部には，fact(1)を呼び出したfact(2)の変数部のベース・アドレスを格納しておく必要がある．

フセット)でアクセスするようにします．これにより，フレーム・ポインタの内容を切り替えることによって着目する計算を切り替えられます．

また，フレーム・ポインタの値をfact(1)のベース・アドレスからfact(2)のベース・アドレスに切り替えたとたんに，fact(1)のベース・アドレスの情報はどこにもなくなります．つまり，fact(1)の変数部にアクセスできなくなり，実質的にそのメモリ領域は解放されたことになります．

ここでは，ランタイム構造の基本的なしくみだけを説明しました．実際にはローカル変数も変数部に格納します．また，fact(1)が完了した時点でfact(2)の計算を引き続き実行するために，関数factのコードのどの部分に戻るかという情報を変数部に入れる場合もあります．具体的な実装はコンパイラに依存します．プログラムの開発に際しては，これらのことを確認したうえでコンパイラを使用することをお勧めします．

● スタック・オーバフロー

ここまでは関数factを例に説明しましたが，ここで，複数のタスクが動作するリアルタイムOSを利用する場合のメモリ空間を考えてみましょう．多くのリアルタイムOSでは，各タスクやタスクのスタックをメモリ空間内に連続的に配置します(**図9**)．そのため，あるタスクがスタックの使いかたをまちがえると，ほかのタスクやOSの領域を壊してしまうことがあります．

組み込み機器で使用するマイコンには，メモリ保護機能が十分ではないものが多くあります．そのため，スタック上の変数部がどんどんふくらみ，スタックとして指定した領域を超えても，見かけ上は何も起こりません．「実機の動作がおかしいなあ」とデバッグを行って初めて，スタックがふくれあがってほかの領域を壊していることがわかるのです．これからきっと何度も経験することと思います．「またスタック・オーバフローか」と．これは，WindowsやLinux/UNIX上では味わえない楽しみ(？！)です．

なお，ここでは再帰呼び出しを例に説明しましたが，通常，組み込みソフトウェアでは再帰呼び出しは使いません．再帰呼び出しを使うと，何度も再帰的に呼び出されることでスタックを大量に消費し，また，