GCCによる X680x0 ゲームプログラミング

吉野智興 著

X680x0 Game Programming by GCC

SOFT BANK

●『付録ディスク』の使い方 ●

注意：『付録ディスク』は，必ずバックアップを作成してから，お使いください。

1. DISK A をドライブ 0 に，DISK B をドライブ 1 に入れ，マシンを起動，あるいは再起動してください。
2. ディスクの説明 (README) が順に表示されます。
   README には，GCC 実行環境をフロッピーベースで使用する場合の方法や，ハードディスクへインストールする方法などが記述されています。必ずお読みください。
3. README の表示が終了すると，本ゲームプログラムのコンパイル/リンクが自動的に始まり，ゲームの実行ファイルが作成されます。

- 本書の本体価格には，フリーソフトウェアファウンデーション (FSF) への寄付金が含まれています。
- 本書に記載したすべてのプログラムは，「GNU 一般公有使用許諾書」を適用しています。
- その他のシステム名，CPU 名などは一般に各社の登録商標です。本文中では，特に TM，® は明記しておりません。

はじめに

この本は,「C マガジン」に連載された『X68k 活用講座 GCC によるゲームプログラミング』をもとにして，初めて C にさわる人でも読めるように書き直したものです。

X68000/X68030 では，ゲームプログラムはほとんどアセンブラで記述されています。ですが，アセンブラは初めてプログラムをする人には難しい言語であり，しかも CPU に依存するために，知識を使い回しすることが比較的困難です。

本書では，リアルタイムでの処理を要求されるゲームプログラミングをすべて C 言語で作成してあります。C 言語は，最もアセンブラに近い高級言語として，最近ではゲーム作成にもさかんに用いられています。本書では，この C 言語を一般的な入門書とは少し違った方法で，ゲームをプログラムできるレベルまで解説をしてみました。完全な初心者向けにはなっていないのではないかという危惧もありますが，ひととおりの基礎知識は身につくように書いたつもりですので，途中で投げ出さないで最後まで読んでいただければ幸いです。

1993 年 10 月

吉野　智典

CONTENTS

Appendix 247

CHAPTER 1

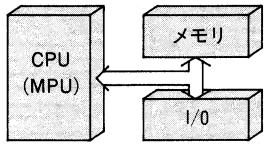
第1章

ゲームプログラミング入門

本章では，**X68000/X68030** でゲームをプログラムする準備に必要な基礎知識を，できるだけ平易に解説しています。ある程度の知識がある人は，この章を読む必要はないかもしれません。

■1.1　電子計算機(こんぴゅーた)

X68000/X68030 は，たくさんの目に見える物体から構成されています。目に見える物体は，それぞれさまざまな役割をもっています。たとえば，フロッピーディスクは，いろいろな情報を記録するのに用いられます。フロッピーディスクドライブは，このフロッピーディスクを読んだり書いたりするための物体，ハードウェアです。**X68000/X68030** は，たくさんのハードウェアから構成された１つのハードウェアであり，その１つ１つが役割をもっているのです。



●Fig. 1.1　X68000/X68030 の構成 (1)

X68000/X68030 をごくおおざっぱに役割別に分類したのが Fig. 1.1 です。実は，世の中に現存しているコンピュータと呼ばれているものの，ほとんどがこの構成になっています。

CPU (MPU)　コンピュータの脳にあたります。ここが計算をしたり判断したりする中枢です。

メモリ　人間でいうところの記憶を司る部分です。人間と違うところは，故障でもしない限り，電源が入っていれば忘れたりしない点です。電源が切れても半永久的に内容を忘れないメモリもあります。

I/O　人間でいえば目や耳や手や足で，外界との連絡をとる部分です。ディスプレイやキーボードやマウスがこれです。

この構成は，マイコン制御電気洗濯機に入っている「マイコン」でも同じです。ただ規模が違うだけで，I/O が洗濯機のスイッチであったり，最近流行のセンサであったりするだけのことです。電気洗濯機では，スイッチから人間の命令を受け取って，センサからの情報によって，すすぎの水の量やモーターの回転数等を CPU が判断して調節しているのです。その気になれば，**X68000/X68030** で洗濯機を制御することもできますが，**X68000/X68030** にはモーターを調節する機能や，水の量を測ったりするセンサがないので，そのままでは洗濯機を動かすようなことはできません(^_^;;;)。

電気洗濯機に組み込まれているコンピュータは，１チップマイコンと呼ばれ，たいてい

の場合，CPU とメモリは 1 つの部品になっています。これに対して，**X68000/X68030** では，CPU やメモリも複数の部品で構成されています。スーパーコンピュータのような超高速で動くコンピュータでは，この個別に構成された部品を結ぶ電線に電気が伝わる時間すら問題になるほどの速さになっていますが，普通の人には関係のない世界ですね (どんなに速くても，基本的な構成はやっぱり同じなのです)。

コンピュータは，このような目に見えるハードウェアだけでは動きません。電気洗濯機では，水の量を調節したり，スイッチが押されたのを判断して動かなければいけません。このような判断や制御を行うのがソフトウェアで，電気洗濯機の組み込み部品になっているコンピュータの場合には，このソフトウェアが部品の中のメモリに消えない形で書き込まれています。このようなハードウェアに形を変えたソフトウェアを，ファームウェアと呼んで，特殊なソフトウェアとして区別することがあります。このようなファームウェアを作る仕事もプログラマの仕事です。コンピュータと名前がつくものはたいてい Fig. 1.1 の構成をしていますので，一部の例外を除いて，コンピュータには消えないメモリに書かれたソフトウェア (ファームウェア) が存在しています。非常におおざっぱな言い方ですが，コンピュータの規模が大きくなるに従って，このファームウェアの役割は小さくなっていきます。たとえば，電気洗濯機のコンピュータはこのファームウェアだけで機能するコンピュータですが，**X68000/X68030** は電源を入れただけではゲームはできませんよね。ある程度規模が大きなコンピュータでは，ファームウェアは普通，別の記憶装置，たとえばフロッピーディスクやハードディスクから，別のソフトウェアを読み出して実行する機能と，若干のテスト機能くらいを備えた小規模なものになっています。ワープロでは，このファームウェアがパソコンでの「ワープロ」機能を行うソフトウェアになっています。これが，見た目はそっくりでも，パソコンがワープロになれるのに，ワープロがパソコンになれない理由の 1 つです。パソコンは，まさにソフトがなければただの粗大ゴミです。

それでは，**X68000/X68030** の CPU とメモリ，I/O について，もう少し細かくみてみましょう。

■1.2 X68000/X68030のCPU (その1)

X68000/X68030 の CPU (人間の脳にあたる) は，アメリカのモトローラ社が開発した CPU です。実はこの CPU 自体も，1 つのコンピュータとして考えることができます。これを理解するにはソフトウェアについての知識が少々必要です。1.1「電子計算機 (こんぴゅーた)」で述べましたが，コンピュータが機能するにはハードウェアとソフトウェアが必要です。ハードウェアにネジや電線といった細かい部品があるように，ソフトウェアにも階層があります。

アバウトな言い方をすれば，最も上位にあるソフトウェアは人間の言葉です。今は SF の世界でしかありませんが，究極のプログラミングとは「○○をやってくれ」といえば機械が勝手にその処理をしてくれることでしょう。今のコンピュータはそこまで進歩していま

せんから,「○○をやってくれ」という人間の願いを，プログラマと呼ばれる人々が苦労してコンピュータに「教えている」のです。人間の言葉を機械は理解できませんから，人間が機械の理解できる言葉で教えてあげなければ，機械は仕事をしてくれません。厄介なことに，機械が理解する言葉を人間が理解するのはたいへんな仕事なのです。そこで，昔の偉い人々は，少しでも人間が理解しやすい言葉で機械に「教える」ことを考えました。当然賢い人たちが考えたことですから，人間が理解しやすい言葉から機械の言葉に変換する仕事は機械にやらせます(ここで鶏が先か卵が先か？という問題が出てくるのですが，悩まないで，そういうものなんだと思ってください)。機械が理解できる言葉を「機械語」，人間が理解しやすい言葉を「高級言語」と呼びます。高級言語にも，機械語に非常に近くてわかりづらいものから，かなり平易なものまで，たくさんの種類があります。

さて，CPU の話を進めるにはここまで知っていれば十分です。これから後の話はまた別のページですることにして，話を戻します。実は，この人間がわかりづらい機械語を理解して実行する方法には，いくつか種類があります。実は **X68000/X68030** で使われている CPU の内部では，この機械語の実行をさらに低レベルの機械語を用いて行っているのです。**X68000/X68030** で使われている機械語は，人間には理解しづらい言葉ですが，CPU が実行するレベルの言葉としてはかなり高級な部類になっています。CPU 内部では，さらに低次元な処理しかできない部分をハードウェアで作成しておいて，それを CPU 内部のメモリに書かれたソフトウェアで動かして，機械語を処理しているのです[1)]。最近の高速コンピュータでは，この方法は時間がかかるために，最初から全部の機械語の処理をハードウェアで行うようにするのが流行です。このような CPU[2)]では，命令自体が単純になっているので，人間の言葉と機械語との格差はさらに広がっています。このような機械では，高級言語から機械語に変換する部分の性能が重要になります。たぶん，**X68000/X68030** は，人間が直接，機械語レベルで扱える機械の限界位置に属するコンピュータといえましょう。

■1.3　2進数の話

実際にコンピュータを使うときには，1.1「電子計算機 (こんぴゅーた)」でお話しした CPU を抽象化します。抽象化の典型が数学です。物事を数値で置き換えて考えてしまう癖は，コンピュータに触る人間の特徴でもあります。

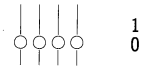
1.3.1 ■　2進数の基礎

CPU の中に「ソロバン」があると考えてください。ただし，このソロバンは，各桁に玉が１つしかない変なソロバンです。

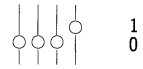
---

1) このようなソフトウェアをマイクロプログラムと呼びます。
2) RISC アーキテクチャと呼ばれます。

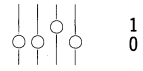
(1) 数値 0 を表す



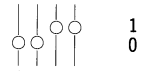
(2) 数値 1 を表す



(3) 数値 2 を表す



(4) 数値 3 を表す



●Fig. 1.2　変なソロバン

Fig. 1.2 の (1) のように，玉が全部下なら数値の 0 を表します。(2) のように，いちばん右の玉が上に上がると 1 を表します。普通のソロバンでは，1 を足すときには，さらに玉をもう 1 つ上に上げるのですが，CPU のソロバンの玉は 1 つしかないので，1 を足すとすぐに繰り上がりが起こって，その左の玉が上がります。これが 2 になります。コンピュータの世界では，すべてこのように数値を 1 と 0 で表す 2 進数の世界になっています。ソロバンの左方向の幅は CPU によって決まっています。電気洗濯機に入っているコンピュータでは，幅が 4 つしかないものがよく用いられています。あのファミコンではこの横幅が 8 つ，スーパーファミコンになると 16 になります。**X68000/X68030** では 32 です。ソロバンの横幅が広いということは，それだけ大きな数が一度に扱えることになります。

このようなソロバンを CPU ではレジスタと呼びます。レジスタは同じようなソロバン (レジスタ) 構成になっていますが，CPU がもっているソロバンにはそれぞれ名前と役割があります。この名前と役割は CPU によって千差万別です。**X68000** が CPU として使っている 68000 と，**X68030** が CPU に使っている 680EC30 でも，このソロバン，つまりレジスタの数は違っています。この違いは，BASIC や C といった普通に使われるプログラム言語でプログラムを行う場合には意識する必要はありませんが，本書で扱うような機械を極限まで使うようなゲームプログラミングでは，この違いを意識することが必要な場合があります。

ここでは，まだ基礎的な知識を学ぶ段階ですから，細かい違いはとりあえず無視して話を進めましょう。ここから先はソロバンにたとえないで，レジスタと，ちゃんと用語を使って説明を進めていきます。レジスタという言葉が出てきたら，「あぁ，1 つしか玉のないソロバンだな」と思ってください。先ほど，レジスタの幅は CPU によって長さが違っていて，**X68000/X68030** の CPU ではその幅は 32 あると述べました。通常のコンピュータでは，この幅を 8 つの単位で扱います。2 進数で 8 桁ですから，2^8，すなわち 0～255 までの数字を表すことができます。この 1 単位をバイトといいます。バイトを 2 つ並べて 16 の幅にしたデータをワード，4 つ並べて 32 の幅にしたものをロングワードといいます。話

が前後しますが，レジスタの桁１つをビットといいます。

1．バイト（8 ビット）　　2 進数 8 桁 0〜255 まで
2．ワード（16 ビット）　　2 進数 16 桁 0〜65535 まで
3．ロングワード（32 ビット）　2 進数 32 桁 0〜4294836225 まで

ここで，数値の計算を考えます。n 進数の計算を知ることは，コンピュータを理解することでもあります。大げさかもしれませんが(^_^;;)。

そもそも n 進数というのは，数が n になった時点で上の桁への繰り上げが起こることを表しています。なぜ普通 10 進数が使われているのかは，人間の手の指の数を数えればわかるでしょう。もし，人間の指が 8 本しかなかったら，きっと世の中では 8 進数が使われていたでしょう。人間が物を数えるときに，指を 1 本，2 本と曲げて数えて，足りなくなったら 10 本を 1 まとめにして，隣の人に指を 1 本曲げておかせます。また，1 本，2 本と数えていって，隣の人が指 3 本，数えた人の指が 6 本曲げてあった，これが 10 進数の 36 です。この隣の人が 1 つ上の桁になるわけです。普通に使われる 1，2，3，4，5，…といった数値は，人間にとってわかりやすいように決められた，ある数に対応する記号にすぎません。1 なら指が 1 本曲げてある，ただそれだけのことです。数 1 を a，数 2 を b，数 3 を c，…と対応させても，数の本質はなにも変わっていないのです。コンピュータが難しいといわれるのは，こういった幼いころから体で覚えた概念を，まったく別の視点からとらえることが難しいからだ，といってもいいすぎではないでしょう。

まず，体に染みついた 10 進数思考から脱却することから始めなければいけません。1 の次の数が 2 だといった固定観念を壊すのです。9 の次は，10 ではないかもしれないのです。事実，コンピュータの世界では，2 進数，8 進数，10 進数，16 進数の 4 つが時と場合によっ

●Table 1.1　数値の記述方法

10 進数	2 進数	8 進数	16 進数
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12

て使い分けられていて，9の次が10ではないことがあります。Table 1.1を見てください。

Table 1.1は，コンピュータで用いられる数値記述方法を，同じ数に対応させて並べたものです。16進数では10進数の10〜15と同じ数を1桁で表す10進での記号がないので，アルファベットのA〜Fまたはa〜fを用いて表すのが一般的です。普通，10進数以外で数値を表すときには，「12」は「じゅうに」とは読まないで，「いちに」といった読み方をします。これだけでは何進数かわからないので，「ヘキサ[3]で」などといった接頭子をつけたりして表現します。「ヘキサで12」と書いても，すぐにはピン! とはこないでしょう。そこで，いくつかの数列を暗記しておきます。

1つは

1，2，4，8，16，32，64，128，256

もう1つは

1，16，256，4096

です。1，2，4，8，…は，2進数を換算するときに使います。たとえば，2進数1101は

2進数		1			1			0			1	
10進数	8×1	+	4×1	+	2×0	+	1×1	=	13			

と頭の中で計算します。また，16進数A13は

16進数		A			1			3				
10進数	256×10	+	16×1	+	1×3	=	2560	+	16	+	3	= 2579

です。ただ，これを頭の中で即計算するのは，さすがにソロバン検定でももっていないと難しいでしょう。16進数に慣れてくれば，このような換算の必要は少なくなってきますが，16進数で0〜FFまで，一目で換算できると本当に楽です(だからといって，無理に覚える必要は全然ありません。プログラミングは受験ではないのですから)。先ほどの数列は，桁上がりの順で，その桁が'1'のときに，対応する10進数を並べたものです。慣れるまで，いろいろな数値を2進数や16進数に直したり，逆に換算したりして，練習してみてください。

さて，次はn進数での計算です。やり方は10進数の筆算とまったく同じです。2進数の加算についてやってみましょう。

```
    1
+   1      ← 1＋1で即桁上がり
-----
   10
```

[3]16進数のことです。

```
   101
+   11      ← 連続して即桁上がり
------
  1000
```

2進数での計算は楽ですね。なにせ，1が2つあれば即桁上がりなので。では，16進数です。こっちは計算が厄介です。

```
   AB13
+   1FD
------
   AD10
```

こうなると頭爆発ですね。なぜこうなるのか，わかるまで先に進まないで，何度もこの節の頭から読み直したり，Table 1.1をよく見てください。加算だけでなく，減算もやってみてください。この本は数学の本ではないので，あえて詳しく説明はしません。これらはすべて学校で習う基礎的な知識です。パソコン通信サービスの1つ，NIFTYのFSHARPというフォーラムでは，高校生くらいの方から「ゲームプログラマになりたいのですが，どうしたらいいですか?」という書き込みがときどきあります。率直にいえば，「学校で教わることを真面目に習いなさい」です。コンピュータでゲームを作りたいなら，まず「数学」，「英語」，「国語」は必須です。マニュアルを読む語学力と数学のセンスは，磨いておいて絶対に損はないのです(説教をタレるほど筆者が勉強をしたわけではないのですが…)。話が大きく脱線。ここまでの話には負の数がありませんね。実は，負の数を2の補数表現の2進数で表す場合にはいくつか方法があるのですが，詳しいことは数学の教科書にまかせて，ここでは**X68000/X68030**のCPUで扱う場合の方法についてだけみてみましょう。

1.3.2 ■ 負の数を2進数で扱う

0～255までを表すことができるバイトで負の数を表現する場合，**X68000/X68030**では「2の補数表現」を使います。

●Table 1.2　2の補数表現

2進数数値	対応する10進数
01111111	127
01111110	126
…	
00000011	3
00000010	2
00000001	1
00000000	0
11111111	-1
11111110	-2
11111101	-3
…	
10000000	-128

Table 1.2が2の補数表現です。数字の並びを見ると，単純に足し算を実行すれば計算が

できるようになっているのがわかるでしょう。

たとえば，

```
    11111110    = -2
  + 11111100    = -4
-------------------
(1) 11111010    = -6
```

いちばん左の桁から繰り上がる 1 を無視すれば，正しい結果になります。引き算をしてみましょう。

```
    11111110    = -2
  - 11111100    = -4
-------------------
    00000010    = 2
```

正しい結果ですね。ところで，10 進数の 100 + 100 を 2 進数で行う場合はどうなるでしょうか?

```
    01100100    = 100
  + 01100100    = 100
-------------------
    11001000    = -56
```

正しくありません。正の数同士を足した結果が負の数になってしまいます。これは，8 ビットの 2 の補数表現で表せる−128〜127 までの範囲を，足し算した結果が超えてしまったからです。これを「オーバーフロー」と呼びます。プログラムする場合には，このようなオーバーフローが起きないように注意する必要があります。完成したゲームなどで，このようなバグが発見されると，「裏技」と称してゲーム雑誌に掲載されます。有名なものとしては，ドラゴンクエスト 3 のカジノでコインを購入する際のバグがありますね。

購入するコイン枚数 × 1 枚あたりの金額

この値が，内部で表せる最大の数を超えてしまったために，とんでもなく安い値段で大量のコインが買えてしまうというバグです。バグはバグですが，あまりに有用なので，裏技として私も使わせてもらいました。もちろん，実際の銀行などで使われるプログラムでこんなことが起こったら一大事です。CPU は，このようなオーバーフローが起こった場合，別の「専用のレジスタ」のある桁が変化するように作られています。さらに安全性を重視するコンピュータでは，このようなオーバーフローが起こった瞬間に，そのプログラムを中止させる機能を備えたものもあります。ファミコンの CPU でも，もちろんこのオーバーフローを検出できます。前記のプログラムの場合，プログラムのほうがそれを忘れていただけです。この専用のレジスタというのが，コンピュータ用語でフラグレジスタと呼ばれるレジスタです。**X68000/X68030** の CPU では，コンディションコードレジスタとも呼ばれます。このレジスタは，演算結果が 0 だったら変化する (たいていは 1 になる) 桁や，オーバーフローが発生すると変化する桁などで構成されていて，CPU はそのレジス

タを参照して処理の場合分けを行うのです。このフラグレジスタは，普通はレジスタとして意識することはありません。ただ，演算をすると，その結果に応じて値が変化するレジスタとだけ覚えておけば今は十分です。

話を2進数の負の値に戻します。この2の補数表現による負数の表し方には，目立った特徴があります。それは，「ある負の値を，それを表現できる n ビットの2進数で表すときには，必ず n ビットの桁は'1'になる」という特徴です。このため，n ビットの桁を「符号ビット」と呼ぶことがあります。この符号ビットは，8ビットの値を16ビットの値に直したりするときに利用されます。たとえば，

```
 8ビット  -128(10進数)      10000000(2進数)
            ↓
16ビット  -128(10進数)      1111111110000000(2進数)
```

上のような拡張は，符号ビットを上位の桁のすべてに満たすことで簡単に実現できます。このように，数値のビット数を増やす行為を「符号つき拡張」と呼びます。符号ビットを無視して，上位桁を全部'0'で満たす拡張もあります。これが「符号なし拡張」です。拡張方法の違いは，その拡張の名前が表すように，元の数値を符号つきの整数として扱うのか，符号なしの数値として扱うのかで決まります。この符号のあるなしで拡張方法が異なるというのは，しばしばC言語での落とし穴になる点です。

ここまでの話で，コンピュータが数学でいうところの「整数」を扱う方法について，ある程度理解できたと思います。ゲームの世界では，たいていX68000/X68030が扱える32ビットの整数の範囲で事が足りるのですが，シミュレーションゲームのようなまさに数値主体の計算ゲームでは，いわゆる小数点以下の数値を扱う必要もあるでしょう。もう一度10進数での数値の扱いを眺めてみましょう。

数値31415は

```
31415 = 10000*3 + 1000*1 + 100*4 + 10*1 + 1*5
        ↑         ↑        ↑       ↑      ↑1が5つ
        |         |        |       └10が1つ
        |         |        └100が4つ
        |         └1000が1つ
        └10000が3つ
```

と表せます。これをそのまま3.1415に拡張します。

$$3.1415 = 1 \times 3 + \frac{1}{10} \times 1 + \frac{1}{100} \times 4 + \frac{1}{1000} \times 1 + \frac{1}{10000} \times 5$$

です。0.1は$\frac{1}{10}$が1つになるわけです。このように小数点以下の数値を扱えば，小数点をはさんで右と左に有限の桁数で計算をすることができますね。これを固定小数点演算と呼びます。この考え方をそのまま2進数にしたのが，2進数での固定小数点表記です。

$$10.1011 = 2 \times 1 + 1 \times 0 + \frac{1}{2} \times 1 + \frac{1}{4} \times 0 + \frac{1}{8} \times 1 + \frac{1}{16} \times 1$$

実はこの方法では，10進数固定小数点で有限桁で表せる数字のうち，2進数では有限桁

にならないものが存在してしまいます。話を簡単に理解していただくために，3 進数を考えてみましょう。$\frac{1}{3}$は，3 進数では

$$\frac{1}{3} \times 1 = 0.1 \text{ (3 進数)}$$

と有限桁で，3 進数における「れいてんいち」です。これが 10 進数では，0.3333333333$\cdots$と無限に続くのです。同じ関係が 10 進数と 2 進数の間にも存在します。このことをよく頭に入れてプログラムしないと，意味不明な動作をするプログラムを書く原因になったりします。コンピュータの用途によっては，このような変換に伴う誤差を避けるために，数値をすべて 10 進数として計算させる場合もあります。方法は単純で，4 ビットで表すことのできる数値 0〜15 を，0〜9 までしか使わないで，本当の 10 進数並びとして計算を行うのです。COBOL のような事務処理言語，つまり銀行などのようにわずかの誤差が問題になる世界では，このような計算方法を使うのが普通です。

固定小数点演算は，3D ゲームや音声処理など，高速に数値演算を行う必要があるときにときどき使いますが，固定小数点方式では，非常に大きな数や小さな数を扱うのがたいへんになります。これは，有効桁数のぶんだけメモリを用意しなければならないからです。しかも，極端に大きい数や小さい数では，小数点近辺の数値を表す部分がほとんど 0 で埋まってしまう，といったたいへんなムダが起きます。

そこで，科学の世界でよく使われる

$$3.1415 \times 10^{23}$$

といった表現方法をそのまま 2 進数にもち込んだ，「浮動小数点数値表現」が用いられています。これをどのように表現するかは，いろいろな規格で決められていますが，**X68000/X68030** では IEEE (あいとりぷるいー) フォーマットを普通使っています。この表現方法については本書の範囲外と思われますので，割愛させていただきますが，コンピュータによる数値計算の世界は，それだけを扱った書籍がたくさん出版されている興味深い分野の 1 つです。関心がある方は，その手の専門書を読んでみられてはいかがでしょうか?

次に，通常の加減乗除以外の演算を説明しましょう。これらの演算は，2 進数を数値としてでなく，0 と 1 の並び (ビット列) として扱う場合に便利な演算です。この演算は，「ある」「ない」といった 1 ビットだけで表現できる概念を扱う場合や，2 進数のある範囲の行だけ操作したいといった場合に用います。よく用いられる演算を以下に紹介しましょう。

AND 演算 (論理積)

op1	op2	op1 AND op2
0	0	0
0	1	0
1	0	0
1	1	1

OR 演算 (論理和)

op1	op2	op1 OR op2
0	0	0
0	1	1
1	0	1
1	1	1

XOR 演算 (排他的論理和)

op1	op2	op1 XOR op2
0	0	0
0	1	1
1	0	1
1	1	0

これらの例は，1 ビットの例ですが，演算自体はビット数に関係なく，**op1** と **op2** の対応する桁を単位に演算を行います。繰り上がりや繰り下がりは起こりません。以下に簡単な例を示しておきます。

```
  op1    op2   op1 AND op2
01101  01001       01001

  op1    op2    op1 OR op2
01101  01001       01101

  op1    op2   op1 XOR op2
01101  01001       00100
```

このような，数値を 0，1 の並びとして扱うビット操作には，他にシフト操作があります。この演算は，名前が示すように，0，1 の並びを桁をずらして新しい数値を生成する演算です。左 1 ビットシフトの例が以下です。

```
0010010110
    ↓        左に1ビットシフト
0100101100
```

このように，シフトの結果「空き」になるいちばん右の桁には 0 が埋められて，新しい数値を生成します。この例では左方向に桁を移動させましたが，逆方向に桁を移動することも考えられます。

```
0010010110
    ↓        右に1ビットシフト
0001001011
```

同じく「空き」になった左の桁には 0 を埋めます。

シフト操作では，その数値を「符号なし」として扱うか「符号つき」として扱うかで，異

なった操作になることがあります。P. 9で，n ビットの数値を 2 の補数表現で用いる場合には，最上位のビットが「正」「負」を表す符号ビットになることをいいました。このように数値を扱った場合，特に右にシフトする場合には，「数値演算」としてシフトを用いると，結果が期待しないものになることがあります。

実は，数値演算としてシフト操作をみた場合には，シフト数を n とすると左シフトは 2^n での掛け算に相当し，右シフトは 2^n での割り算に相当するのです。実際の例でみてみましょう。

```
00000011  (3)
   ↑          右1ビットシフト
00000110  (6)
   ↓          左1ビットシフト
00001100  (12)
```

ところが，この 8 桁の 2 進数で「2 の補数表現」で負の数値を使った場合には，右シフトが破綻します。

```
01111100  (124)
   ↑          右1ビットシフト
11111000  (-8)
   ↓          左1ビットシフト
11110000  (-16)
```

このような破綻を防ぐために，シフト操作には 2 種類あり，シフト操作の対象となる値を符号つきでシフトするのか，符号なしでシフトするのかを区別して使うことができるようになっています。「符号なし」でのシフトを「論理シフト」，「符号つき」でのシフトを「算術シフト」と呼んで区別します。

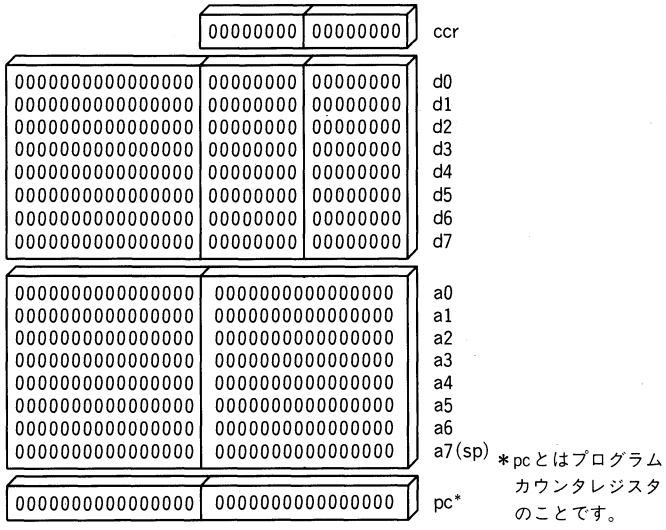
算術シフトでは，以下のように，論理シフトとは異なったシフトを行います。

1. 右シフトでは最上位ビット (符号ビット) は変化しません。いつも最上位のビットの値が，演算結果の最上位ビットにコピーされます。
2. 左シフトでは，符号ビットが変化するようなシフトが起こった場合に，オーバーフローを検出できるようになっています。

このように，シフト操作では，「符号」といったややこしい概念をよく理解して使わないと，思わぬ動作をすることがあります。少し前に述べた AND，OR，XOR といった演算で不用意に符号ビットを操作してしまうことも，バグの原因になります。プログラムに慣れない間は，シフトや論理演算といったビット操作は，すべて符号なしの値に対して行うようにしておくのが，バグを回避するよい方法です。

■1.4 X68000/X68030のCPU (その2)

ここからは，いよいよ **X68000/X68030** についての話を始めましょう。Fig. 1.3 が **X68000/X68030** の CPU にあるレジスタの構成です。ccr と書いてあるのがフラグレジスタで，d0～d7，a0～a7 の全部で 16 個の 32 ビットレジスタが CPU の中に用意されています。このうち，d0～d7 はバイト，ワード，ロングワードの幅で値を扱うことができ，a0～a7 はワード，ロングワードの幅で値を扱うことができます。**X68030** で使われている 680EC30 では，この他にもレジスタが増えていますが，普通にプログラムを行う場合には扱うことはないものなので，割愛させていただきます。



●Fig. 1.3 68000/680EC30 のレジスタ構成

X68000 に使われている CPU68000 と，**X68030** で使われている CPU680EC30 では，680EC30 が 68000 の上位互換ということになっています。「ということになっています」というのは，細かいレベルでみるとけっこう非互換な部分があるからです。**X68000** で動いて，**X68030** で動かないソフトウェアが存在するのはこのせいです。また，CPU のクロック (**初代**[4]では 10MHz，**XVI** では 16MHz，**X68030** では 25MHz) の違いもあって，非常にシビアなタイミングでプログラムされているソフトウェアは，互換ではなくなります。クロックの話が出たついでに。68000 と 680EC30 では内部演算の幅自体が違いますから，同じ命令を同じクロックで実行しても 680EC30 のほうが速く実行します。これは，ALU と呼ばれる CPU 内部の計算を行う部分が，68000 では 16 ビット，680EC30 では 32 ビットである点や，シフト操作が 68000 ではマイクロプログラミングで行われるのに対して，680EC30 ではハードウェア (バレルシフタと呼ばれる) で行われる点などの違いがあるせいです。CPU の内部についてのさらに詳しい話を進めるには，メモリと I/O につい

[4] 初代の X68000 のことです。

ての基礎知識が必要になります。ここまでの説明で難しいなぁと感じたら，もう一度この章を最初から読み直して，ここまでの話が理解できるまでは次に進まないようにしてください。

■1.5 メモリの話

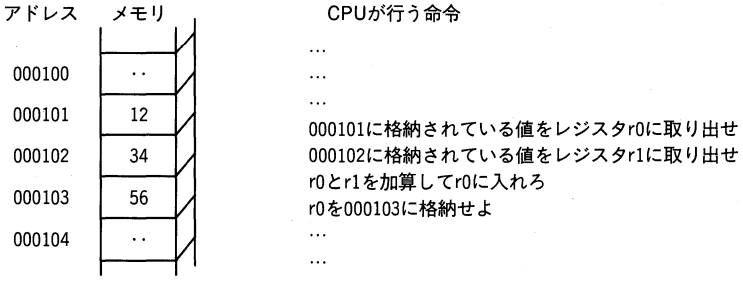
メモリといえば，半導体摩擦の代表選手といえるくらい新聞・テレビなどで有名ですが，実際のコンピュータでどのように使われているのでしょうか? 新聞やテレビでは「1個で新聞○×ページ分の情報を記憶します」などと表現していますが，コンピュータについての理解が深まってくると，こうした表現はあまりに抽象的な表現であるということがわかってきます。

1.3「2進数の話」(P.4) で説明したように，バイト単位 (8ビット) で表せる数字は0～255までです。英数字ならこの範囲ですべてを表すことができます。実際，パソコンで通常使われるアスキーコードと呼ばれる，どの数字をどの文字に割り当てるかを決めた規則では，英数字は全部割り当てられています。ですが，これはどの数字でどの文字を表すかを決めているだけであって，たとえばパソコンのディスプレイに表示されている文字そのものを表すわけではありません。コンピュータでの情報は，すべてこのように，ある状態や現象を数字に置き換えて扱います。同じAという文字を表現するのに，コンピュータの内部では単なる数字で表す場合もあれば，Aという文字をたとえば16×16のマス目に分解して絵文字で示す場合もあるのです。ですから,「新聞○×ページ分」といっても，新聞をグラフ用紙みたいに1mm単位で分割して，白い部分と黒い部分とに分けて記憶する場合と，記載されている文字だけをコード化して，それを記憶する場合とでは，記憶に必要とするメモリの量は全然違っています。メモリに扱う情報をどのように蓄えるかというのは，今でも重要な研究テーマの1つです。

普通パソコンでは，メモリは1バイトを単位として扱います。カタログなどに記載されているメインメモリ1Mバイトというのは1,048,576バイト，8,388,608ビットの記憶容量があることを示しています。ファミコンなどで宣伝されている8M ROMの大容量というのは，パソコン世界ではたかだか1Mバイトで，フロッピー1枚にも満たない小容量になります。今どきのゲームでは，フロッピー1枚に収まったゲームはそうお目にかかれないので，この点でもすでにパソコンとファミコンでは数値の次元が違っています。ただ，ファミコンのROMではメモリの大きさが即価格に結びつくために，データを高度に圧縮しておくことが普通ですので，一概に比較するのは問題がありますが。

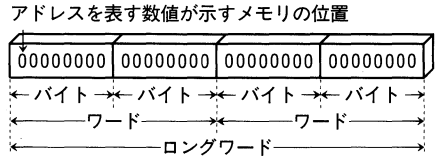
メモリには，1バイトを単位として，全部に住所を割り当てます。コンピュータ用語ではアドレスと呼びますので,「なんだ，そのまんまじゃないか」と思われるでしょう。この住所は，やはり数値で表されていて，数値の範囲はコンピュータによって異なっていますが，同じCPUのコンピュータなら同じです。CPUが同じでも，同じ数値が表すアドレスに同じようにメモリがあるとは限りません。これは，そのコンピュータを設計した設計者

の思想によって変わってきます。CPUは，アドレスを指定してメモリから数値をレジスタに取り出して加工することと，アドレスを指定してレジスタからメモリに数値を格納することを繰り返し実行します。これがCPUの仕事なのです。



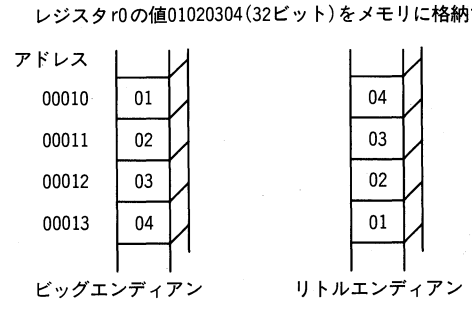
●Fig. 1.4　CPUがプログラムを実行するときの概念

アドレスは数値で表されるといいました。この数値は，バイト単位で割り当てられています。それでは，1.3「2進数の話」(P.4)で書いた，ワードやロングワードの値はどのように扱われるのでしょうか? 実は，**X68000/X68030** では，ワードやロングワードのデータは，バイト単位で割り当てられたアドレスの数値が大きくなる方向へ連続して記憶されるのです。



●Fig. 1.5　メモリ上での数値の並び方

ですから，同じアドレスを指定しても，そのアドレスをバイトとして扱うのか，ワード，もしくはロングワードとして扱うのかによって，表す意味がまったく違ってくるわけです。また，ロングワードの場合には，バイトの値が4分割されるわけですが，これをどのような順番でメモリ上に置くのかは，CPUによって異なっています。これをエンディアンと呼びます。**X68000/X68030** のCPUではビッグエンディアン，インテルのCPUを用いた機械ではリトルエンディアンになっています。



●Fig. 1.6　ビッグエンディアンとリトルエンディアン

Fig. 1.6 のように，ビッグエンディアンでは，上位の桁がアドレスの小さいメモリに格納されます。リトルエンディアンでは，まったく逆に，上位の桁がアドレスの大きいほうに格納されます。本書を書いている時点では，世の中の趨勢は圧倒的にリトルエンディアンに傾きつつあります。残念ながら (?)，リトルエンディアンを採用しているインテル系列の CPU が数の上で圧倒的に多くなっているからです。

さらに，**X68000/X68030** では，このアドレスが奇数の場合，ワードやロングワードとして値を取り出したり書き込んだりはできない仕組みになっています。これを行うと，**X68000** ではおなじみの白い窓が現れて，「アドレスエラーが発生しました」と表示されます。**X68030** では「たいていの場合」はアドレスエラーにはなりません。

メモリのアドレスは単なる数値なので，CPU のレジスタに格納して，それで扱うアドレスを示すのが普通です。16 ビット幅までのレジスタをもつ CPU では，このレジスタで直接扱える数値の範囲が実際に存在するメモリの分量に比べて狭いので，メモリのアドレスを指定するのに，2 つのレジスタを組み合わせたり，アドレスの桁数が足りない部分を補うレジスタを別に用意したりしてメモリを扱います。レジスタを 2 つ組み合わせる方法は，Z80 のような 8 ビットの CPU に使われていた方法であり，アドレスの足りない桁を補う方法は，あの悪名高いセグメントをもつ 8086 系の CPU に用いられている方法です。悪名は高いのですが，このセグメントの概念は，プログラミングで重要な論理アドレスと物理アドレスを理解する上で都合がよいので，少し説明してみます。

8086 では全部で 1M バイトのアドレスを指定できます。16 進数でいえば$FFFFFです。この CPU では，内部のレジスタは全部 16 ビット幅なので，全アドレスを単独のレジスタで指定することはできません。そこで，別にセグメントレジスタというレジスタを設けて，そのレジスタと組み合わせてアドレスを指定します。1 つのレジスタで指定できる範囲は 16 ビット幅 0～65,535 までなので，それにセグメントレジスタの値でゲタをはかせて，0～1,048,575 の範囲で任意の 65,535 バイトの範囲を指定するのです。

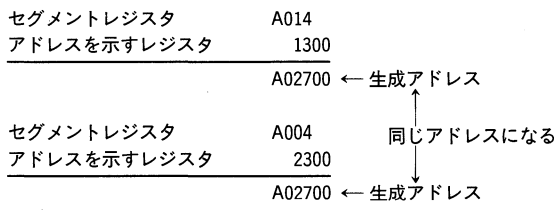
```
セグメントレジスタ          A014
アドレスを示すレジスタ        1300
                          ------
                          A02700 ← 生成アドレス
```

●Fig. 1.7　8086 のアドレス生成

ちょっと頭をひねればわかりますが，このゲタのはかせ方によって，0～1,048,575 に割り当てられたメモリの 1 つの位置を示すのに，何種類もの指定の方法が存在するのです。あるセグメントで，0～1,048,575 の任意位置の 0～65,535 だけ離れた範囲を指定するのですから。同じメモリのアドレスを示す数値が何種類かある場合は，それら何種類かあるアドレスを，互いに他のアドレスの「エイリアス」(英語で「別名」を意味する) と呼びます。

Fig. 1.8 を見てください。セグメントレジスタと，そのセグメント内の位置を示すレジスタ 2 組が，結果的に同じアドレスを生成しています。これは同じアドレスのメモリに別の指定の方法があるということを示しています。

ここで，1 つの仮定を行います。プログラマがこのセグメントレジスタの存在を知らないものとしたら，どうなるでしょうか? プログラマに与えられたメモリの範囲は，レジス



●Fig. 1.8　8086 のエイリアス

タで表現できる 0～65,535 の範囲でしかありません。小さいプログラムならこれだけあれば十分です。プログラマは，0～1,045,875 の範囲にあるメモリのどの部分を使っているのかを知る必要はありません (ただし，このプログラムを管理する側は，ちゃんとメモリの 0～1,045,875 のどこかを割り当ててあげて，プログラマがメモリを使えることを保証してあげる必要があるのですが，これはここでは重要ではありません)。前者のプログラマが意識すればよい 0～65,535 の範囲を「論理アドレス」，後者の実際のメモリのアドレスを示す範囲を「物理アドレス」といって区別することがあります。8086 では，この論理アドレスがたったの 16 ビットでしかなかったために，実際に大きなプログラムを組もうとすると，即座に 16 ビットの範囲を使い切ってしまい，セグメントレジスタを自ら操作しなければならない状況に陥るわけです。これが 8086 のセグメントを「悪名高い」ものにしてしまった原因です。セグメントの考え方自体は問題なく，そのセグメントの大きさが実用にそぐわなかっただけのことです。80286 や 80386，80486 では，こういった制限が事実上なくなるモードが用意されています。

X68000/X68030 の CPU では，レジスタは 32 ビットです。これで表せる値は 0～4,294,836,225 の範囲で，この大きさになれば，レジスタで直接アドレスを指定しても十分な大きさになります。実際には **X68000/X68030** では，レジスタの上位 8 ビットはアドレスの値として扱われない (無視される) ので，アドレスとして扱えるのは 0～16,777,215 の範囲です。事実，CPU が `010A0000` 番地を読んでも，`000A0000` 番地を読んでも，同じ値が読めます。これは 8086 でのエイリアスのようになっているのですが，ハードウェアの世界ではイメージアドレスと呼びます。**X68000/X68030** では，CPU が扱うアドレス (論理アドレス) は実際のメモリのアドレス (物理アドレス) と完全に一致しています。ただし，**X68030** で CPU を MC68030 に交換した場合には，CPU が扱うアドレスが外のメモリのアドレスと一致しないようにすることができます。この仕掛けは何のためにあるのでしょうか?

その昔，メモリは現在と異なりとっても高価で,「足りない!! 増設しよう」などとはすぐにはできない相談だったのです。そこで，賢い人は考えました。CPU がみえるアドレスと，実際のアドレスを分離してしまう方法です。たとえば，物理的には 1M バイトしかないメモリを，見掛け上は 4T (テラバイト) あるかのようにする方法です。CPU があるアドレスを指定して，メモリを書こうとします。そのアドレスは，メモリ管理ユニットと呼ばれるハードウェアに与えられます。このメモリ管理ユニットの内部では，その CPU の指定したアドレスが実際にメモリに割り当てられているかを調べます。実際にメモリが割り当

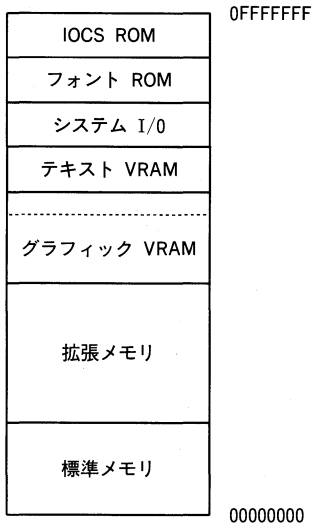
てられていれば，その論理アドレスから実際のハードウェアに割り当てられている物理アドレスに変換して，メモリの内容を書き換えるのです。もし実際にメモリに割り当てられていないなら，空いているメモリを捜して，その物理アドレスをCPUが指定した論理アドレスに変換して，メモリを書き換えます。もし空きがないなら，いったん現在のプログラムを中断させて，今までで使用頻度が低いメモリ部分を，たとえばハードディスクなどに退避し，そこを空きにしてメモリを割り当てて書き換えを行い，中断していたプログラムを再開します。このような使用頻度の低い部分を追い出す行為を「スワップ」といいます。このような管理をする場合には，CPUが指定したアドレスは本当に仮想的なもので，実際のメモリの中身は，実はハードディスクに退避されていたりして，本当のアドレスではなくなっています。現在のコンピュータでは，このような「仮想記憶」と呼ばれる手法は，メモリの見掛け上の大きさ(メモリ空間)の拡張よりも，不正なメモリの操作を防止してシステムダウンを防ぐ目的のほうが重要になっています。高速なワークステーションで「スワップ」が頻繁に起きるような場合は，本当に深刻なメモリ不足か，プログラムに欠陥があるのです。

X68000/X68030 では，このような仮想記憶は使われていませんが，不正なメモリ操作によるシステムダウンをできるだけ防ぐようになっています。このために，某国民機種とは違って，**X68000/X68030** では「安心して」プログラムを組むことができます。使われていない仮想記憶の説明をこんなにも長々と行ったのは，**X68000シリーズ**は**X68030**の登場によって，より高度なパソコンへと進化する兆しがあるためです。事実，一部の有志によって，CPUがMC68030に置き換えられた**X68030**上で**UNIX**が動きつつあります。

パソコンのメモリは，CPUが操作する数値データやグラフィックデータだけではなく，プログラム自身を記憶するのにも使われます。電気洗濯機に使われているようなCPUのように，プログラムが入っているメモリと，数値を処理するためのメモリがまったく別個に存在するものもありますが，一般にパソコンでは，プログラムを記憶するためのメモリと，数値や文字を入れておくメモリは同じです。ですから，あるアドレスに記憶されている値がプログラムであることもあるし，文字である場合もあるし，単なるゴミの値であることもあるのです。このメモリには，任意の値を書いたり消したりできるメモリと，書き込みができない，読むだけのためのメモリが存在します。1.1「電子計算機(こんぴゅーた)」(P.2)の冒頭で説明したファームウェアは，この書き込みができないメモリに入っています。書き込みできるメモリをRAM，書き込みできないメモリをROMといいます。カタログに記載されているメインメモリは，前者の書き換え可能なRAMの大きさを普通示してあります。このRAMを，現在市販されている**X68000シリーズ**では，最低でも2Mバイト実装しており，普通にプログラムするには(少々不便ではあるものの)問題はないでしょう。ですが，**X68000/X68030** では「積めるだけ積んでおく」のがベストです。本格的にプログラムを学習したい方は，ぜひとも6Mバイト程度は実装しておいてください。

さて，メモリの基本的な事項が理解できたところで，我らが**X68000/X68030**についてより詳しくみてみましょう。CPUがもっているアドレスの値に対して，どのようにメモ

リを割り当てるかを表したものをメモリマップと呼びます。



●Fig. 1.9　X68000/X68030 の簡略メモリマップ

X68000/X68030 の CPU では，24 ビット 16M バイトのアドレス範囲 (これをアドレス空間と呼びます) をもっています。16M バイトのアドレス空間は，今やそれほど広いとはいえませんが，**X68000/X68030** が発売された当時には驚異的に広い空間でした。**SHARP** の技術者さんは，この空間を惜しげもなく使っており，「バンク切り替え」とか「ラム・ウインドウ」などのおかしなハードウェアは，**X68000/X68030** にはまったく無縁になっています。512K バイトのグラフィック VRAM は，実に 2M バイトものアドレス空間に割り当てられ，CPU はそれに何の操作もせずに直接読み書きできるのです。漢字フォントを格納した ROM も，そのまま直接 CPU のアドレス空間に割り当てられていて，「その気」になればユーザが直接リードすることができます。某国民機種でプログラムしたことがある人ならわかるのですが，このストレートアクセス可能という機能は，涙が出るほど嬉しいものでもあるのです。

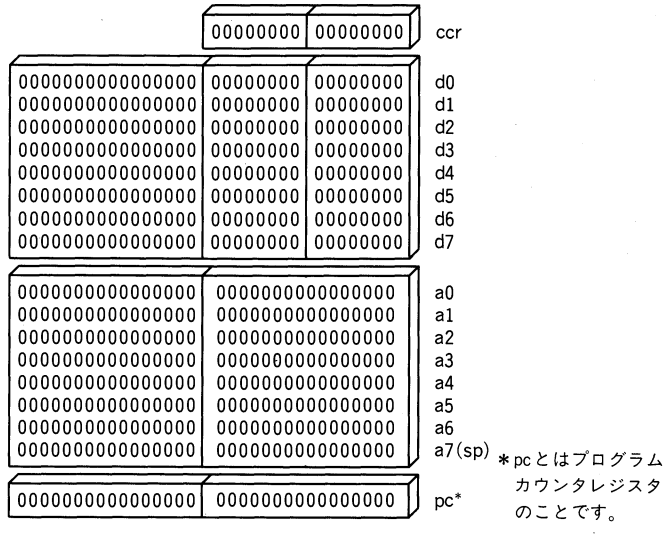
X68000/X68030 のメモリマップの概略が理解できたところで，I/O の話です。I/O は Input/Output の略です。CPU が外部と (それは人間であることが多々ある) 接触を行うためのハードウェアが，この I/O です。**X68000/X68030** の I/O は，某国民機種のそれとは違った扱いになっています。この違いは，「モトローラ系 CPU」のパソコンと，「インテル & ザイログ系 CPU」のパソコンでえんえん受け継がれてきたものです。この「違い」は，「CPU が I/O のための特殊命令をもつか」，あるいは「CPU が独立した I/O アドレス空間をもつか」という違いなのです。インテル& ザイログ系列の CPU をもつパソコンでは，メモリのアドレス空間とは独立して I/O のアドレス空間をもっています。モトローラ系列 CPU のパソコンでは，独立した I/O のアドレス空間は存在せず，I/O も「普通のメモリ」にみえるようになっています。この違いによる影響は，I/O を直接扱う C 言語でのプログラムで顕著になってきます。今の段階で，この違いを具体例で示すにはまだ勉強すべき

ことが多すぎるので，後のページに譲りますが，「**X68000/X68030** ではI/Oもただのメモリにみえる」という点だけを記憶しておいてください。ただし，「ただのメモリ」にみえても，I/Oはメモリではないのです。先ほど，**X68000/X68030** のグラフィックVRAMは512Kバイトが2Mバイトのアドレス空間に割り当てられていると述べました。グラフィックVRAMはメモリではありますが，広義の意味ではI/Oです。**X68000/X68030** の多くの画面モードに応じて，グラフィックVRAMは有効なアドレス空間の範囲が変化するのです。

また，ときには，I/Oは「値が勝手に変化する」メモリのように，CPUにはみえます。この「勝手に変化する値」をCPUは読み取って判断し，いろいろな操作を行うのです。また，「読む」操作と「書く」操作で意味が違ってくる場合があるのもI/Oの特徴です。普通のRAMなら，書いた値がそのまま読めるのが当然で，もしそうでないなら，その機械は故障していることになります。ですが，I/Oでは，「書いた値」がそのまま読めるとは限らないだけでなく，「読めても書けない」とか，「書けても読めない」ものがたくさんあります。**X68000/X68030** は，「I/Oがメモリのようにみえる」メモリマップドI/O方式なので，C言語で直接I/Oを操作できますが，I/Oはあくまで I/Oですので，その操作を記述する場合には，十分な注意が必要であることも頭に入れておいてください。

■1.6 X68000/X68030のCPU (その3)

メモリ，I/Oの話が終わったので，いよいよCPUの中身について詳しく話をすることにします。



●Fig. 1.10 68000/680EC30 の CPU のレジスタ構成

Fig. 1.10が68000/680EC30のCPUのレジスタ構成です。d0～d7はデータレジスタと呼ばれ，主に加算したり減算したりするのに用いる演算用のレジスタです。a0～a7はア

ドレスレジスタと呼ばれ，メモリやI/Oの話で述べた，アドレスを主に扱うためのレジスタです。8個あるアドレスレジスタのうち，a7あるいはspと書かれたレジスタはちょっと特殊で,「スタックポインタ」と呼ばれることがあります。

それでは，CPUの動き方をみてみましょう。プログラムは，つねにpcと記述される(プログラムカウンタと呼ばれる)レジスタに格納されたアドレスのメモリから読み出されます。読み出されたプログラムは，CPUによって解釈され，実行されます。プログラムが実行されると，プログラムカウンタは自動的に更新されて，次の命令が格納されているメモリを示す値になります。このように，命令を1つ1つ実行する形式のコンピュータを「ノイマン型」と呼ぶことがあります。ときどき新聞や雑誌で見かける「非ノイマン型コンピュータ」は，この形式ではないコンピュータをいいますが，**X68000シリーズ**は全部ノイマン型です。

本書の主な使用言語はCです。ですが，アセンブラの基礎的な知識は，C言語では不可欠であり，本書のテーマであるゲームプログラミングにおいては特に重要です。そこで，アセンブラの基礎的な知識もここで覚えてしまいましょう。コンピュータの命令には種類がたくさんありますが，基本的な命令は「移動」(move)命令でしょう。要するに，ある値をある場所から別の場所へ移動するのです。

```
move.l  d0,d1
```

これは，**X68000/X68030**でデータレジスタd0に入っている値を，32ビットでd1に移動する命令です。d1にもともとあった値は失われ，d0に入っていた値がd1の値になります。移動元のd0の内容はそのまま残ります。命令を分解して詳しく説明すると，次のようになります。

```
移動せよ  32ビットで  d0の内容を  d1に
move      .  l       d0 ,       d1
```

68000/680EC30では8ビット，16ビットの値も扱えますので，

```
移動せよ  16ビットで  d0を  d1に
move      .  w       d0 ,  d1
move.w    d0,d1

移動せよ  8ビットで   d0を  d1に
move      .  b       d0 ,  d1
move.b    d0,d1
```

のような命令もちゃんと存在します。この場合，d1の値で移動に関係のない部分はそのまま残ります。たとえば，d0の値が0000AABB(16進数)，d1がCCDD00EE(16進数)だった場合，

```
move.w  d0,d1
```

を実行すると，d1 はCCDDAABB(16 進数) となって，上位のCCDD(16 進数) はそのまま値を保持します。移動方向の記述が「左から右」なのは，モトローラの伝統です。インテル系では普通「右から左」に移動します (ことごとくインテルとモトローラとは反対ですねぇ)。これがレジスターレジスタ間での移動命令です。モトローラさんは，アドレスレジスタが転送先の場合にはmoveaという表記を用いるようにしていますが，**X68000/X68030** のアセンブラは寛容なので，moveと書けば「移動」と解釈してくれます。

移動命令の基本的なフォーマットは

```
move.[b][w][l]    転送元,転送先
```

です。[b][w][l]は転送サイズで，それぞれ 8 ビット，16 ビット，32 ビットです。省略も可能で，省略した場合には 16 ビットと解釈されますが，省略しないで記述したほうが後々のためです。転送元，転送先は，レジスタであったり，メモリであったりします。アセンブラの解説書では，転送元や転送先を示す部分を，<ea>とよく記述してあります。eaは「有効アドレス (effective address)」の英語頭文字を並べたものです。68000/680EC30 の移動命令では，有効アドレスは次のものが許されています。

(1) 即値 (イミーディエイト) 形式

これは有効アドレスの値そのものです。たとえば，move.l #0,d0は値 0 を 32 ビットで d0 に移動します。アセンブラでは，その値に '#' をつけて記述します。

```
move.l   #10,d0    : 値10を32ビットでd0に移動
move.w   #20,d0    : 値20を16ビットでd0に移動
move.b   #30,d0    : 値30を8ビットでd0に移動
```

値が指定されたビット幅で表せない場合は，アセンブラがエラーを指摘します。当たり前ですが，転送先にこれを指定することはできません。

(2) 絶対ショート (アブソリュートショート) 形式

有効アドレスを，16 ビットの符号つきの値として 32 ビットに拡張します。その値をアドレスとみなして，値が指すアドレスに格納されている値を移動します。

```
move.l  10.w,d0    : メモリ10番地に格納されている32ビット値をd0に移動
move.w  20.w,d0    : メモリ20番地に格納されている16ビット値をd0に移動
move.b  30.w,d0    : メモリ30番地に格納されている8ビット値をd0に移動
```

絶対ショート形式は，有効アドレスを示す表記に，'.w' をつけて指定しますが，**SHARP** 純正のアセンブラの **Ver.1.****では，これを指定しても **(3)** の絶対ロング形式としてアセンブルされます。有効アドレスは，必ず 32 ビットに拡張されてから，その値のアドレスのメモリを参照します。ですから，この形式で指定できるのは0(16 進数) 番地から7fff(16 進数) 番地，ffff8000(16 進数) からffffffff(16 進数) 番地になります (ここでこの意味が理解できなくても，大きな問題ではありません)。

(3) 絶対ロング (アブソリュートロング) 形式

有効アドレスをそのままアドレスとみなし，格納されている値を移動します。

```
move.l  10,d0   : メモリ10番地に格納されている32ビット値をd0に移動
move.w  20,d0   : メモリ20番地に格納されている16ビット値をd0に移動
move.b  30,d0   : メモリ30番地に格納されている8ビット値をd0に移動
```

絶対ショート形式と異なり，68000/680EC30 で許される全アドレス範囲を指定できます。

(4) アドレスレジスタ間接形式

アドレスレジスタに格納されている値をアドレスとみなし，そのアドレスに格納されている値を移動します。

```
move.l (a0),d0  : a0の値のアドレスに格納されている32ビット値をd0に移動
move.w (a0),d0  : a0の値のアドレスに格納されている16ビット値をd0に移動
move.b (a0),d0  : a0の値のアドレスに格納されている8ビット値をd0に移動
```

他にもたくさんの形式がありますが，本書はアセンブラの解説書ではありませんので，詳しく知りたい人は，別の参考書を参照してください。本書では，<ea>を有効アドレスと呼ぶことだけを知っておけば十分です。

■1.7 高級言語

前節で少し説明したアセンブラですら，実は高級言語とも呼べます。アセンブラ言語で記述されたプログラムですら，CPU が直接実行できるわけではないのですから。アセンブラ言語で記述されたプログラムは，アセンブラと呼ばれるプログラムで機械語に翻訳されます。機械語に翻訳された時点で，やっと CPU が直接実行できるプログラムになるのです。この CPU が直接実行できるプログラムがメモリに格納され，それを CPU が逐次実行していくのです。アセンブラ言語が低級言語に分類されるのは，その命令単位が直接 1 対 1 で機械語に対応しているせいでもあるのですが，最近のアセンブラ言語ではこの原則すら崩れつつあります。高級なワークステーションで用いられる RISC プロセッサのアセンブラでは，アセンブラが実行速度を向上させるために命令を並べ替えたり，挿入削除を行うため，人間が記述したアセンブラ言語と機械語とが対応しないことがあるからです。

一般に高級言語と呼ばれるものには，FORTRAN, COBOL, PL/I, Pascal, Ada, Modula2, APL, C, BASIC, LISP, Prolog 等の種類があります。こうした高級言語は，「インタプリタ」と「コンパイラ」の 2 つの種類に分けることができます。インタプリタは，高級言語で記述されたプログラムを逐次解釈しながら実行していきます。典型的なインタプリタが BASIC です。逐次解釈しながら実行しますので，実行速度は比較的低速ですが，作成→実行→デバッグの期間が短くて，「お手軽」なのが身上です。また，逐次解釈なので，巨大なプログラムでは，文法上のエラーがあっても，そのエラーのある部分がなかなか実

行されないためにエラーが発覚しない，というおもしろい特徴があります。

インタプリタがプログラムを解釈しながら実行するのに対して，コンパイラは，プログラム作成時にあらかじめすべてのプログラムを一気に機械語に翻訳してしまいます。プログラムの実行は翻訳された機械語で行いますから，インタプリタに比べると非常に高速になります。実行速度の違いは，インタプリタで同じプログラムを実行する場合に比べて100倍程度高速になることもあります。また，インタプリタでは，プログラムを逐次解釈するために，エラーが存在する部分を解釈実行するまでエラーを検出できないのに対して，コンパイラではプログラム全体を一度に解釈しますので，エラーは実行する前に検出されます。

コンパイラのよいことばかりを並べましたが，長所があれば短所もあります。まず，実際に実行するまでの手間と時間がインタプリタよりかかってしまいます。インタプリタでは普通「対話的」にプログラムやデバッグができますから，バグを発見してから修正実行するまで非常に短時間ですみます。コンパイラではプログラムを一括翻訳しますので，修正をかけてから実行するまでのサイクルに時間がかかります。

また，コンパイラが検出できるようなエラーは，インタプリタより楽に修正できますが，プログラムの論理的な誤りはコンパイラでは検出できません。「論理的な誤り」というのは，たとえば「ゾウを空に飛ばす」というのは，日本語として文法上は正しいのですが，事実の上では変になる，そういった類の誤りです。コンパイラは，プログラムがプログラムとしての方法と意味において誤っていなければ，何も文句をいいません。「ゾウを空に飛ばす」といったようなとんでもないプログラムでも文句はないのです。インタプリタでは，普通，こういったとんでもない操作に対してはある程度防護手段が設けられているので，プログラムは停止してデバッグを要求してきます。

コンパイラでは，どちらかといえば，こういった「速度を犠牲にする」ようなチェックはできるだけ行わないようにするのが普通なので，プログラムの誤った操作をそのまま翻訳して実行してしまいます。特にこの本で扱うC言語は，こういったチェックをいっさい行わない速度重視のコンパイラ言語です。

コンパイラとインタプリタは，このようにほぼ正反対の性格をもったプログラム開発手段です。このような正反対の性格をうまく中和させるような，コンパイラインタプリタと呼ばれるシステムも存在しています。このシステムは，プログラムをいったん機械語よりは高級で，ソースプログラムよりは低級な中間言語にコンパイルした後，別のプログラムがその中間言語を解釈実行するようなシステムです。ただ，このような「どっちつかず」のシステムはあまり受け入れられないようで，それほど普及していません。

昔，8ビットCPUがパソコンの主流だったころには，**UCSD–Pascal**のような優れたコンパイラインタプリタシステムが存在していましたが，CPUの実行速度が一昔前の大型コンピュータ並みになった現在では，コンパイラの「開発からデバッグ」の繰り返し時間もわりあい短くなったために，開発されたプログラムの実行速度が速いコンパイラ全盛の時代になっています。

CHAPTER 2

第2章

C言語入門

いよいよC言語について話を始めます。第1章での知識を踏まえて具体的に解説をしていきますので，頑張ってお読みください。

■ 2.1 プログラム作成方法について

「プログラム作成方法」といっても，プログラムの経験がある人なら常識的なことですが，本書は**X68000/X68030**ベッタリが許されていますので，**X68000/X68030**でのプログラム作成方法を1から解説します。本書は，「X68k Programming Series」（ソフトバンク）を対象として書かれています。**XC**は生成される機械語の性能などに問題があるため，ゲームを対象とした場合には向いていません。

2.1.1 ■ COMMAND.Xに慣れましょう

現在の**X68000/X68030**でプログラム開発を行うには，ビジュアルシェルや**SX–WINDOW**のようなグラフィカルな環境は使えません。そのような指向の開発ツールがないのも原因ですが，プログラムを組み上げるのに慣れてくると，どうもカーソルがペコペコ光っているオーソドックスな環境が便利に思えてくるせいもあるのでしょう。ただし，初心者には，このような環境は「わからない，触れない」状態であるため，敷居が高いのも事実です。でも，臆病になっていては進歩がありません。とにかく，COMMAND.Xに慣れてください。プログラミングの第一歩です。

C言語でゲームを組み上げるには，最低でもエディタが使え，ディレクトリやCONFIG.SYS，AUTOEXEC.BATくらいは理解できないと，先に進むことができません。本書でこういった基本的な知識を詳しく話すと，肝心のプログラミングの話ができなくなってしまいます。とにかく，「習うより慣れろ」です。もう1つ。特にゲームオンリーユーザの方に目立つのですが，**Human68k**の最近のバージョンをもっていない方がおられます。少なくとも，**Human68k Ver.2.01**より新しいバージョンのものを使ってください。「もらってきたフリーソフトウェアがどうやっても動いてくれません」と苦情をいう方の話をよく聞いてみると，「**初代X68000**です。**Human68k**は**Ver.1**…」ということがときどきあります。**Human68k**は，市販の基本ソフトウェアとしては破格ともいえる安さなのです。「いっちょゲームでも」と燃える前に，自分自身の**X68000/X68030**の環境をまずチェックしてください。

2.1.2 ■ やっぱりhello, world

Cプログラミングを行うには，コンパイラが正しく動作する環境を作る必要があります。『付録ディスク』ではこの点を考えて，基本的に「ディスクを入れてリセット」するだけでコンパイルできる環境を提供してあります。ただし，これは本当に最低の環境であって，各プログラムについてのマニュアルはいっさいありません。マニュアルを提供するのが本書の目的ではないからです。それでも「Cを体験する」には十分なので，自分で環境を構築

できない人でも実際にコンパイルを経験できます。コンパイルできる環境が整ったら，何はさておいても「hello, world」でしょうね。hello, world は，C 言語のバイブル「プログラミング言語 C」の最初に登場するプログラムです。でも，この簡単なプログラムでさえ，実際にコンパイルして動かすまでには，たいへんな努力が必要な場合もあるでしょう。でも，めげないで頑張ってください。

■ 2.2 関数

C 言語では，すべてのプログラムが「関数」と呼ばれるもので構成されています。一般的な BASIC (**X68000** での **X–BASIC** ではない) や，科学計算で多用される FORTRAN，事務系の大型コンピュータでは事実上これしか使われていないという COBOL には，多少 C 言語での関数みたいなものは存在しますが，C 言語ほど徹底していません。また，コンピュータの教育でさかんに使われる Pascal より，C 言語は文法がゆるやかになっています。ただ，個人的には，初めてプログラムを覚えるときは Pascal が最も適切ではないかとは思います。C 言語は，そのゆるやかな性格が，初心者にとっては大きな壁になることが多々あるのです。

話がちょっとそれました。関数というのは，アセンブラレベルでいえば「サブルーチン」に相当します。普通プログラムを作成する場合，ある役割をもった小さなプログラムを 1 つの単位として記述してやり，それらを組み合わせて完成されたプログラムにします。本書のテーマであるゲームについてちょっと考えてみましょう。ゲームでは普通，キーボードかジョイスティックで操作を行います。その操作を元にして，さまざまな反応をプログラムは行うわけです。

1. **ジョイスティックまたはキーボードを読み取る。**
2. **操作に応じた反応をする。**
3. **1. へ戻る。**

1.〜**3.** を繰り返すのが究極的にはゲームです。プログラム的にこれをみると，「繰り返し」と呼ばれる基本動作がみえています。**2.** の「操作に応じた」というのは，同じく基本動作である条件判断です。より C 言語らしく書き換えてみます。

```
1: main ()
2: 始まり
3:     loop:
4:     ジョイスティックまたはキーボードを読む
5:     操作に応じた反応をする
6:     loopの位置に戻る
7: 終わり
```

C 言語では，`main()`のように「関数名称」に‘()’をつけて関数を表します。いちばん最

初に実行される関数がmain()です。'()'の'('と')'の間に何か文字が入ることもありますが，これは後で述べます。2行目の始まりと書いた部分は，C言語では'{'で示します。7行目の終わりの部分は'}'で示します。前にちょっと出てきたPascalでは，'{'，'}'は「そのものズバリ」の**BEGIN**と**END**です。

```
1: main ()
2: {
3:     loop:
4:     ジョイスティックまたはキーボードを読む
5:     操作に応じた反応をする
6:     loopの位置に戻る
7: }
```

これで一歩C言語に近づいた記述になりました。3行目のloop:は，C言語でもやっぱりloop:です。4〜5行目はC言語では「関数呼び出し」になります。関数呼び出しは関数名称 ();という形式になります。普通のC言語では，日本語は関数名称には使えません(本書の『付録ディスク』に収録した**GCC**では，日本語を使えるような環境にしてあります)。そこで英文字を使って書き直します。

```
1: main ()
2: {
3:     loop:
4:     read_joystick_or_keybord ();
5:     do_action ();
6:     loop の位置に戻る
7: }
```

もうこうなると，ほぼ90%はC言語プログラムです。6行目のloopの位置に戻るは，Cではそのものズバリgoto loop;と記述します(ここから先は混乱のない限り，「C言語」をCとだけ記述します)。

```
1: main ()
2: {
3:     loop:
4:     read_joystick_or_keybord ();
5:     do_action ();
6:     goto loop;
7: }
```

これはもう完全なCプログラムで，実際にコンパイルすることもできます。でも実行はできません。実際にコンパイルしてみると，次のようにエラーが出て，実行できるプログラムは作られないでしょう。

```
undefined symbol(s) in main.o
_read_joystick_or_keybord
_do_action
```

それもそのはず，read_joystick_or_keybord()とdo_action()は，実際には記述されていないからです。Cでは，コンパイラは「こいつはみたこともない関数だな」と思っても，「きっとどこかにあるんだろーな」とおおらかに許してしまいます。Pascalでは，こういったおおらかさはまったくなくて，read_joystick_or_keybordなんて知らないぞ!!といってエラーにします。このCのおおらかさが，逆に初心者にとっては障害になりやすいのです。たとえば，雑誌に掲載されたプログラムを一生懸命エディタで作成してコンパイルすると，

```
undefined symbol(s) in games.o
xxxxxxxxxxx
yyyyyyyyyyy
....
```

と最後にエラーが出て，実行可能なプログラムが生成されないことが多々あります。ちょっとでもCをかじったことがある人なら，即座にエラーの原因がわかるかもしれませんが，初めての人には，これだけでプログラミングを投げ出すのに十分な理由になります。**GCC**でもそうですが，最近のCコンパイラは，こういったおおらかさを許さないで警告メッセージを出力するような機能が備わっています[1)]。

話が大きくそれてしまいました。もう一度Cの関数に戻りましょう。関数という言葉は数学で用いられますね。次のような数学の問題では，

2次関数$f(x)$がある。$f(1)$は2，$f(2)$は5，$f(3)$は10である。関数$f(x)$を求めよ。

のように関数という言葉が使われています。Cでの関数も，これと形式は似ています。先ほどのread_joystick_or_keybord()を考えてみましょう。$f(x)$の'f'に相当するのが，read_joystick_or_keybordです。'('と')'に挟まれた部分に記述されるのが，数学の関数と同じ「引数」と呼ばれるものです。

read_joystick_or_keybord()は，引数がない形式の呼び出しです。引数がある形式でも記述できます。たとえば，

```
read_joystick_or_keybord (0)
```

といった記述です。

引数を使えば，read_joystick_or_keybord()の本体でこの引数を判断して，

[1)] **GCC**では，オプションに-Wallを付加すれば，このような場合に警告を発生するようになります。

```
read_joystick_or_keybord (0);   キーボードだけ読み取る
read_joystick_or_keybord (1);   ジョイスティックだけ読み取る
read_joystick_or_keybord (2);   キーボード，ジョイスティック双方を読み取る
```

というように，機能を変えることができます。引数を変えることで機能が変化するのは，read_joystick_or_keybord()が行っていることで，呼び出し側では，「0 を渡せば，“キーボードだけ読み取る” という動作を行ってくれる」という了解を前提にしています。read_joystick_or_keybord()を，そういう機能を実現するようにプログラムしておかないと意味がないのです。

数学の関数と C の関数が決定的に異なっている点は，数学の関数は「何かしら数値を返す」ことを必ず期待していますが，C 言語では，値が返ってくることを期待する場合とそうでない場合がある点です。普通，read_joystick_or_keybord()のような関数は，「ジョイスティックの A ボタンが押された」とかいうような，何かの情報を返すことを期待してプログラムされるでしょうから，これは明らかに数学でいう関数に近いものです。ところが，do_action()のような関数は，特に何らかの情報の返却を期待するわけではなく，画面の書き換えとか，do_action()を呼び出すことで行われる操作を期待しています。Pascal では，前者のような結果の返却を期待するものを function (英語での関数)，後者のような「呼び出すことによって行われる操作」を期待するものを procedure (英語での手続き) といって，厳密に区別しています。C 言語では，Pascal の「手続き」に相当する関数によって引き起こされる変化を，「関数の副作用」と呼ぶことがしばしばあります。というか，プログラムで使われる関数のほとんどがこの副作用を期待するもので，数学的な関数は全体の一部でしかないのが普通です。この手続きと関数との区別のなさからくる弊害や，前に述べた「おおらかな」仕様は，特にプログラムに慣れていない初心者を混乱させるでしょう。最近の C 言語では，この点が大幅に改良されています。これらに関しては，変数について理解してから改めて書いたほうがよいので，後に回すことにします。

次に，非常に非常に誤解が多いのが，C 言語では最初からコンパイラが知っている関数は main()だけであるという点です。入門書で多用されるprintf()のような関数は，コンパイラの知らない関数です。BASIC では，グラフィック命令である画面に線を描画するLINE命令や，キーボードから文字を読み取るINKEY$といった命令が最初から組み込まれていますが，C 言語では，こういった「組み込み機能」は備わっていません。printf()といった類の「便利のための関数群」は，コンパイラの仕様の外にあるライブラリとして，コンパイラベンダが提供する関数なのです。逆にいえば，C コンパイラがあれば，こういった類のライブラリは自分で構築できてしまうのです。「X68k Programming Series」の「**LIBC**」がその実例です。C 言語では，個別機種ごとに異なってしまうようなライブラリ関数をコンパイラの仕様から追い出すことで，他の言語にはみられない移植性をもったのですが，逆にベンダごとに異なったライブラリが提供されたため，混乱した時期がありました。ANSI 規定によって，この混乱は収束に向かいつつありますが，それでも機種間の移植は (他の言語に比べればやさしいでしょうが) 難しいのが実情です。

この節では，次の点を理解しておいてください。

1. **C では，プログラムは関数の集まりで記述される。**
2. **関数には，「手続き」的なものと「数学関数」的なものがある。**
3. **C コンパイラが知っている関数は，`main()`だけである。**

ここで1つだけ注意しておきます。C はすべて「関数」を単位にして，プログラムしますが，言語の種類からいえば，「手続き型」の言語であって，「関数型」ではありません。

■ 2.3 変数

変数とは何でしょう?? 前節で述べた，「数学関数」の機能をもった関数の返した値を格納しておいたり，繰り返しの回数を数えたりするための機能が変数です。普通，変数はメモリに格納された値であることが多いのです。

2.3.1 ■ 変数の実体

List 2.1 は，整数の変数`val`に，16 進数で`$aabb`という値を設定するプログラムです。本当に変数`val`に値が入っているかをみてみましょう。まずは，このプログラムをコンパイルします。

List 2.1 t0.c

```
1: int val = 0xaabb;
2: main()
3: {
4: }
```

◆コンパイルしてみる

```
A:/>gcc t0.c -O-g
```

変数`val`をのぞくには db.x を使います。ここでは直接メモリをのぞきますので，**SHARP** 純正のデバッガを使いますが，**GDB** のようなソースコードデバッガでは，このようなことをしなくても，ソースを参照しながら変数の名前で値の参照や変更ができます。特にメモリのアドレスを意識する必要もありません。ですから，ここの説明を実際にわざわざ実行してみる必要はありません。頭の中で理解しておけば十分です。さて，デバッガでは，変数`val`はその頭に'._'をつければ，次の実行画面のように **dl** コマンドで参照できます。なぜ参照できるのかは現在のテーマではありません。とにかく，こうすれば参照できるものなんだと覚えてください。

◆val をのぞいてみる

```
A:/>db t0.x

X68k Debugger v2.00 Copyright 1987,88,89,90 SHARP/Hudson
loading t0.x
PC=000AAC3C USP=00088E32 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AAB10 000ABED8 00089842 0007D110  000AAC3C 00000000 00000000 00088E32

__main:
lea $000ABED8,A7 ;000ABED8(62)
-dl ._val

000AB6E8  0000AABB 00000000 00000000 00000000                ..サ............
000AB6F8  00000000 00000000 00000000 00000000                ................
000AB708  00000000 00000000 00000000 00000000                ................
000AB718  00000000 00000000 00000000 00000000                ................
000AB728  00000000 00000000 00000000 00000000                ................
000AB738  00000000 00000000 00000000 00000000                ................
000AB748  00000000 00000000 00000000 00000000                ................
000AB758  00000000 00000000 00000000 00000000                ................
```

dl ._val で何やら数字が出力されました。先頭の$000AB6E8はvalに割り当てられたメモリのアドレスです(メモリってなに? アドレスってなに? → 1章に戻ってください)。**int** というのは，**X68000/X68030** の C 言語では，4バイト，1ロングワードの変数ですよという意味です。**dl ._val** の結果をみてみましょう。

```
000AB6E8  0000AABB ←AABBにちゃんとなっている
```

「詐欺みたい?」な書き方ですが，これでいいのです。では，変数valの値を変更してみましょう。

List 2.2 t1.c

```
1: int val = 0xaabb;
2: main ()
3: {
4:   val = 0xccdd;
5: }
```

関数main()の'{'と'}'の間にval = 0xccdd;が入りました。これは，変数valに16進数で$ccddを入れなさいという命令を指示したものです。「変数valに16進数で$ccddを入れる」という立派なプログラムです。それでは，デバッガでみてみましょう。

◆val を変更してみる

```
A:/>db t1.x

X68k Debugger v2.00 Copyright 1987,88,89,90 SHARP/Hudson
loading t1.x
PC=000AAC46 USP=00088E32 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000AAB10 000ABEE2 00089842 0007D110  000AAC46 00000000 00000000 00088E32

__main:
lea $000ABEE2,A7 ;000ABEE2(00)
-dl ._val

000AB6F2  0000AABB 00000000 00000000 00000000                ..サ............
000AB702  00000000 00000000 00000000 00000000                ................
000AB712  00000000 00000000 00000000 00000000                ................
000AB722  00000000 00000000 00000000 00000000                ................
000AB732  00000000 00000000 00000000 00000000                ................
000AB742  00000000 00000000 00000000 00000000                ................
000AB752  00000000 00000000 00000000 00000000                ................
000AB762  00000000 00000000 00000000 00000000                ................

-g
program terminated normally
-dl ._val

000AB6F2  0000CCDD 00000000 00000000 00000000                ..杢............
000AB702  00000000 00000000 00000001 08090000                ................
000AB712  00000000 00000000 00000000 00000000                ................
000AB722  0001080A 00000000 01000000 00000000                ................
000AB732  00000000 00000001 09080000 00000200                ................
000AB742  00000000 00000000 00000000 00010908                ................
000AB752  00000000 03000000 00000000 00000000                ................
000AB762  00000001 080A0000 00000400 00000000                ................
```

最初の **dl ._val** では，値がたしかに0000AABBになっていますね。次の **g** コマンドは，プログラムを実行しなさいというコマンドです。デバッガが，“program terminated normally”といってます。これはちゃんと実行できたよという意味ですが，実際にはただ単に「バスエラー」とかいう致命的なことは起きなかったよ，というだけのことです。さて，valをみてみましょう。**dl** コマンドでvalをみると，$0000CCDDになっています。ちゃんと値が変更されていますね。完璧です。プログラムは意図したとおりに動いているのが確認できました。

ここまでで説明していないC言語の特徴を補足しておきます。C言語では，プログラムを実行するということは，main()を実行することです。どんな巨大なプログラムでも，最初はmain()を実行します。ですから，main()がないCのプログラムは(普通は)存在しません。List 2.1の

```
main()
{
}
```

は,「何もしない」main()です。List 2.2 の

```
main()
{
  val = 0xccdd;
}
```

は,「変数valに 16 進数で$CCDDを入れる」main()です。デバッガの **g** コマンドでこのmain()を実行したので，変数valは$0000CCDDになったのです。変数とは，プログラムで値を変更できるメモリに他なりません。ですが，C 言語では，すべての変数をメモリに割り当てているわけではありません。この理由をお話しするには,「変数のスコープ」という概念を理解していただく必要があります。

また，変数には必ず変数のタイプがあります。int a;と書けば 32 ビットの符号つき整数の宣言，short a;なら 16 ビットの符号つき整数の宣言，char a;ならば 8 ビットの符号つき整数の宣言になります。**X68000/X68030** の **GCC** で使える基本的な変数のタイプは，Table 2.1 のとおりです。

●Table 2.1　基本的な変数のタイプ

変数のタイプ	値
signed char	8 ビット符号つき整数 (−128〜＋127)
unsigned char	8 ビット符号なし整数 (0〜255)
char	8 ビット符号つき整数 (符号なしにもできる)
signed short	16 ビット符号つき整数 (−32768〜＋32767)
unsigned short	16 ビット符号なし整数
short	16 ビット符号つき整数
signed int	32 ビット符号つき整数 (16 ビットにもできる)
unsigned int	32 ビット符号なし整数 (16 ビットにもできる)
int	32 ビット符号つき整数 (16 ビットにもできる)
signed long	32 ビット符号つき整数
unsigned long	32 ビット符号なし整数
long	32 ビット符号つき整数
signed long long	64 ビット符号つき整数
unsigned long long	64 ビット符号なし整数
long long	64 ビット符号つき整数
float	32 ビット浮動小数点数
double	64 ビット浮動小数点数
long double	64 ビット浮動小数点数

Table 2.1 は，C で直接扱える変数のタイプのすべてです。プログラマは，この基本的な変数のタイプを増やすことはできません。

2.3.2 ■ 変数のスコープ

「変数のスコープ」を理解するには，まず「ブロック」について理解することが先決です。関数の本体の始まりを示す‘{’から‘}’が，Cでは最大の「ブロック」になります。

```
{
    変数宣言部;
    実際の処理;
}
```

これが1つのブロックです。**変数宣言部**は(厳密ではありませんが),

```
変数タイプ　変数識別子0;
変数タイプ　変数識別子0,変数識別子1,[変数識別子2];
```

の2種類の宣言を任意の数並べることができます。変数宣言の後，処理を記述して‘}’で閉じてやればブロックの完成です。ブロックの中にも，ブロックを記述できます。このような構造を「ネストする」とよくいいます。

```
{
   変数宣言部0;
   処理0;
   {
     変数宣言部1;
     処理1;
   }
}
```

このように，ブロックの中にブロックを置けます。‘{’と‘}’で囲まれたブロックは，最大のブロックである関数の始まり‘{’と関数の終わり‘}’の間であれば，いくつでも内部にブロックを作ることができます。

実際にコンパイルできる例がList 2.3です。ここで`foo()`，`foo1()`といった名前が出てきますが，これはA君，B子さんといった一般的な例示を示しているだけです。特に大きな意味はありません。

List 2.3

```
1: foo()
2: {
3:   int a;
4:   a = 0;
5:   foo0(a);
6:   {
7:     int b;
```

```
 8:     b = 1;
 9:     foo1(b);
10:   }
11:   {
12:     int c;
13:     c = 2;
14:     foo2(c);
15:     {
16:       int d;
17:       d = 3;
18:       foo3(d);
19:     }
20:   }
21: }
```

ブロックについて理解していただけたでしょうか?? ブロックが理解できれば,「変数のスコープ」もわかります。コンパイラは，変数の参照や変更があった場合，その変数の名前をブロックの内側から外側に向かって捜し，最初に見つかった宣言を有効にします。

```
{
   int nantara;
   ...
   ...
   {
      int kantara;
      ...
      {
         ...
         nantara = 0;
         ...
      }
   }
}
```

コンパイラは，nantara = 0;を読んだ時点で，nantaraという名前の変数を捜してみます。まず，nantara = 0;が存在するブロック内での宣言からnantaraを捜します。ここで見つからなければ，その外のブロックを捜します。それで見つからなければ，さらに外…で，関数の外まで出てnantaraが見つからない場合は，最後にエラーになります。とにかく，最初にその変数の宣言が見つかったブロックでの宣言が有効となります。同じレベルのブロックに同じ名前が存在しない限りは，個々のブロックにいくつ同じ名前の変数があってもかまいません。内側ブロックの同じ名前の変数の宣言は，外側の変数をみえないようにします。

```
{
   int nantara;  /* (1) */
   ....
   {
      int kantara;
      nantara = 10;  /* このnantaraは(1)のnantara */
      ....
      {
         int nantara;   /* (2) */
         ...
         {
            int kantara;
            nantara = 5; /* このnantaraは(2)のnantara */
         }
      }
   }
}
```

上の例では，最初のnantara = 10;では，コンパイラは(1)のnantaraを見つけてそれを操作するのだと認識し，後のnantara = 5;では(2)のnantaraを操作するのだと認識します。説明に新しく加わった'/*'と'*/'で囲まれた部分があります。この部分をコンパイラは無視(' '(スペース)に置き換えられる)します。これはプログラムに注釈を入れるのに用います。

ある変数nameがどの範囲までみえているかが重要です。コンパイラが宣言された名前nameを見つけることができる範囲を，「変数のスコープ」と呼びます。このような変数のスコープという概念は，比較的最近の言語にしかみられない機能ですが，なぜこのような規則が使われるのでしょうか?? 一度でもこうした機能のない言語，たとえばBASICやCOBOLでプログラムしたことがあれば身にしみてわかるのですが，スコープとは変数の管理の混乱を防ぐための機能なのです。

人間はいい加減なもので，だいたいディスプレイで見えている範囲の変数しか把握できないものです。特に初心者のBASICプログラムにみられるのですが，I，II，III，IIII，IIIIIという変数が，あっちこっちにたくさんバラまかれています。こうしたプログラムになる理由について考えてみましょう。まず，ちょっとした処理を行うために一時的な変数を使いたい … といったときに(特にBASICでは宣言しなくても変数を使えるので)，安易に'I'を使います。だんだんプログラムが複雑になってくると，「あれぇ? この'I'はここで使っているからもう使えないや。え〜い，'II'を作るか」で'II'が誕生します。で，これをえんえん繰り返して'III'，'IIII'ができてしまい，最後にはもう完全に管理できなくなってしまいます。最近のプログラム言語では，このような，宣言なしに使え，しかも，プログラム全体でその変数がみえてしまうような言語は，ほとんど使用されなくなっています。使いたい変数は使いたい範囲で限定して，しかもちゃんと宣言して使う，これが最

近のプログラム言語の傾向なのです。Cについても，この考えはしっかり導入されています。以上が，この節で書いた「ブロック」と「変数のスコープ」がCに存在する理由です。

C言語では，「最大ブロック」(関数の'{'と'}'で囲まれた範囲)の外に変数を宣言することができます。この宣言による変数のスコープは2種類あります。

```
       int a;       /* このaはどこからでもみえる */
static int a;       /* このaはこの変数自身が存在するファイル
                       の中でのみみえる */
```

いきなりこのように書いても，非常にわかりにくいでしょうね。これは，C言語のもう1つの特徴でもある分割コンパイル機能のためにあります。普通，中規模以上のプログラムになると，C言語ではソースプログラムを機能ごとに分割して記述し，最後にそれぞれのプログラムを1つにくっつけて大きなプログラムにします。たとえば，次のように3つに分けて記述します。

game0.c　ジョイスティック処理
game1.c　スプライト関係の処理
game2.c　セーブロード処理

これをコンパイラにかけると，

game0.o
game1.o
game2.o

という3つのファイルが作られます。これをリンカと呼ぶプログラムで1つにまとめるのです。ここでちょいと頭をひねればわかりますが，この3つのソースプログラムで，共通の変数，そうですね，たとえば獲得点数を計算する変数を考えてみましょう。この変数を，仮に`int tokuten`とでもしておきましょう。この`tokuten`は，game0.c，game1.c，game2.cの3つのソースプログラムからみえる必要があるのです。「どこからでもみえる」というのは，まさにこの`tokuten`のような変数に必要な性質なのです。C言語では，「変数は宣言しなければ使えない」のですから，`int tokuten`はgame0.c，game1.c，game2.c全部に宣言しておく必要があります。宣言は全部で3つありますが，`int tokuten`自体が3つあるわけではなくて，リンクされる段階で1つにされます。とにかくリンクされる段階で，「どこからでもみえる変数」は，同じ名前であれば1つにされると覚えておいてください。

これに対して，`static int a;`のような宣言は，コンパイルされる各ファイルの中でしかみえない変数の宣言です。game0.cに`static int number;`があって，game1.cにも`static int number;`があるとしましょう。この2つの変数は，リンクされてもまったく独立した変数として扱われていて，game0.cのプログラムにはgame1.cの`number`はみえないし，その逆も成立するのです。後で混乱をしないために，もう一度改めて強調してお

きます。この規則は，Cでの「ブロック」の最大単位である，関数始めの‘{’と関数終わりの‘}’の外で宣言された変数について適用されます。後で出てきますが，関数始めの‘{’と関数終わりの‘}’の内部で宣言するときに使う，‘**static**’がついた宣言とは意味がまったく異なります。同じ‘**static**’の文字を使うので，本当によく混乱を招くのですが，とにかく‘**static**’の文字は，関数始めの‘{’と関数終わりの‘}’の中にある宣言と，外にある宣言とはまったく別だということを忘れないでください。

2.3.3 ■ 変数の寿命

変数には，もう1つ重要な要素として「寿命」があります。変数の寿命は，原則的に「スコープ」と同じになります。変数は「みえている」間だけ生きています。関数始めの‘{’と関数終わりの‘}’の中で宣言された変数は，ある例外を除いてスコープの範囲で生きています。最初に宣言された時点での値は不定です。どんな値が入っているかは特定できません。変数は，宣言と同時に値を入れてあげるのがバグを入れない方法の1つです。一度コーディングしてから，わざわざ変数の宣言を関数の頭の‘{’の直後に移動してコンパイルする必要はさらさらありません。必要な範囲でブロックを作って，その中で使えばいいのです。保守性能はこちらのほうがずっと上です。変数は，使いたいときに，使いたい範囲で宣言し，値を入れてから使う，これがコンパイラの最適化をも助ける有効な方法なのです。

例外が，前節の最後で強調した，関数始めの‘{’と関数終わりの‘}’の中で宣言した‘**static**’がついた変数です。この変数のスコープはそのブロック内ですが，寿命はそのプログラムが動いている間有効です。しかも，最初に0が入っていることが保証されています。ただし，0が入っていることを積極的に利用してはいけません。また，他の関数からこの変数はみえないので，不用意に値が変更されることもありません。

2.3.4 ■ 変数の記憶クラス

Cでは，「変数をどこに置くか」を指定することができます。たとえば，頻繁に利用される変数は，比較的低速なメモリに置くよりは，高速なレジスタに割り当てておくほうがプログラムの実行が速くなります。とはいえ，これは現在全盛の最適化Cコンパイラでは，実質無意味になっています。コンパイラが自分で変数ごとのスコープや参照回数を計算して，どの変数をレジスタにし，どの変数をメモリに割り当てるかを勝手に決めてしまうからです。ただし，関数始めの‘{’と関数終わりの‘}’の外で宣言された変数と，関数始めの‘{’と関数終わりの‘}’の中で‘**static**’をつけて宣言された変数は，レジスタに割り当てられることはありません。コンパイラが自動的に変数の割り付けを行うようになりましたので，最近では，**auto**や**register**といったCのキーワードはあまり使われなくなっています[2)]。

[2)] **auto**，**register**は変数の記憶クラスを示すCのキーワードです。

■ 2.4 式と文

先の節では，変数の宣言について注目したため，実際の処理を行う部分は**実際の処理;**などとごまかして説明をしていました。今度はこの処理についての説明です。C 言語では，式の後ろに ';' をつけたものが処理の 1 単位になります。まず式を理解しましょう。

2.4.1 ■ Cでの式とは

C での**式**は，数学で述べるところの式とはかなりニュアンスが違っています。実例をあげてみましょう。

```
int a;  /* これは宣言 */

test()
{
  a       /* a は式 !! */
  ;       /* ; がくっついて文 ! */
}
```

式は，変数の名前を書いたものであったり，数値そのものであったり，関数呼び出しであったりします。

```
int a;
foo()
{
  a;
  1;
  foo();
  foo;
}
```

嘘みたいなプログラムですが，これはコンパイルエラーにはならない立派な (?) プログラムです。C では「評価できるもの」はすべて式なのです。とりあえず「評価できるもの」を覚えておきましょう。

1. **数値**
2. **変数名称**
3. **関数**
4. **関数名称**

この 4 つを覚えておけば，たぶん不自由はしないでしょう。数学でいう式を考えてみましょう。たとえば，

$x + y$

です。これはC言語でも式です。'+' は数学でも演算子と呼びますが，C でもやはり演算子と呼びます。何度も書きますが，式に ';' をつけると文になります。x + yが式ですから

```
int x,y;
foo()
{
  x + y;
}
```

は，コンパイルエラーにならない立派なCのプログラムです。では，x + yは何をしているのでしょうか?? Cでは，これを式を評価しているといいます。変数xにyを加算して,「いくらになった?」と評価しているのです。ただ，その評価結果はどこにも発表されません。ただ捨てられるだけです。捨てないで利用する方法があります。よく使われるのが，代入演算子 '=' です。C言語では，単なる代入すら演算なのです。

```
int x;
foo()
{
  x = 3;
}
```

これはxに3を入れるプログラムですが，厳密にコンパイラがどう考えるかというと，以下のようになります。

1. **x = 3を式として評価します。**
2. **値 3 を得ます。**
3. **xが値 3 になります。**

'=' は左辺に右辺を代入して評価する演算子です。プログラムらしく書いてみましょう。

```
int x,y,z;
foo()
{
  z = x + y;
}
```

これを細かくみてみます。コンパイラは，z = x + yを式として評価しようとします。演算子には，優先順位がありますので，それに従って評価します。

1. **'+' は '=' より優先順位が高いので，まずx + yを評価，つまりx + yを実際に計算します。これを *ans* とでもしておきましょう。**

2. 次に，z = *ans* を式として評価します。

3. 値 *ans* を得ます。

4. zが値 *ans* になります。

あくまで式を評価した結果として，zにx + yの値が入るのです。文は，ある意味では式の値を評価して捨てているのです。実際に，zはx + yの値にはなりますが，式z = x + y自体の評価は捨てられています。

ここまで理解できたら，まず **if** 文から考えてみましょう。**if** 文は，C では次のような書き方をします。

```
if (式)
  文 0
else
  文 1
```

if は**式**を評価して，その結果が 0(整数の 0) でなければ**文 0** を実行し，結果が 0 だったら **else** の次にある**文 1** を実行します。また，**else** は省略できます。この場合には，評価した結果が整数の 0 でなければ**文 0** が実行され，それ以外では何もしません。

式の部分には，C でいう式であれば何でも入れることができます。例をあげて考えてみましょう。

```
int x,y,z;
foo()
{
  if (z = x + y)
    x;
  else
    y;
}
```

まず，コンパイラは式 z = x + yを先ほど書いた手順で評価し，値 *ans* を得ます。副作用として，zが *ans* の値になります。先ほどは，式 z = x + yの後ろに ';' があったので，この評価された値 *ans* は捨てられるだけでしたが，今回はちょっと事情が違います。**if** で評価されている式なので，結果 *ans* の値がもし 0 でなかったら文 x;を，0 だったら文 y;を実行するという，人間が普通考えるところの評価分岐をするのです。

if (式) では，式が 0 か，そうでないかしか判定できません。これではプログラムなんてできませんよね。そこで，C には比較演算子というのが用意されています (演算子の詳細については Appendix 2「演算子の一覧表」(P.259) を参照してください)。

exp0 == exp1　式 exp0 と式 exp1 の値が等しいとき 1

exp0 != exp1　式 exp0 と式 exp1 の値が等しくないとき 1

exp0 > exp1 式 exp0 が式 exp1 の値より大きいときに 1
exp0 >= exp1 式 exp0 が式 exp1 の値と等しいか，より大きいときに 1
exp0 < exp1 式 exp0 が式 exp1 の値より小さいときに 1
exp0 <= exp1 式 exp0 が式 exp1 の値と等しいか，より小さいときに 1

C 言語では，比較すら演算子で行い，その結果は式なのです。条件を満たす場合には整数で 1 の値に，そうでない場合には整数で 0 になります。式なのですから，代入演算子=を使って，実際に 1，0 の値を得ることもできます。

```
int x,y,z;
foo()
{
  x = (y == z);
}
```

式y == zが()で囲ってあるのは，代入演算子=のほうが比較演算子==より優先順位が高いためです。このプログラムでは，yとzの値が等しいときにxは 1 になり，そうでないときには 0 になります。()を忘れると全然意味が違ってきます。

```
int x,y,z;
foo()
{
  x = y == z;
}
```

この場合には，まずx = yが先に評価され，結果はyの値になります。同時に，xの値もyの値になります。次に，その評価されたyの値とzの値とを比較し，0 または 1 の値を評価しますが，この評価された値はただ捨てられます。演算子の優先順位は，覚えるよりも()を使って明確にすることをお勧めします。C 言語に慣れてくるほど，この式を駆使した記述をするようになってきます。そのとき，うろ覚えの優先順位でプログラムを記述すると，後でデバッグで泣くハメになります。

式には，変数と同じくタイプがあります。Table 2.1 に，C 言語の基本的な変数のタイプの一覧を示しましたね。式にもそれと同じだけのタイプが存在しています。C では，変数名称単体でも式になります。この場合の式のタイプは，その変数自身のタイプになります。もし，式が演算子をもっている場合には，かなり複雑なタイプ変換が行われてその結果のタイプになるのですが，この複雑なタイプ変換を経ても，通常，常識を逸脱したような変換は起こり得ないので，詳しい話はここでは割愛させていただきます。詳しく勉強なさりたい方は，「プログラミング言語 C」(共立出版) を参照してください。

次に関数呼び出しです。C では，関数呼び出し単体は式です。式にはタイプがありますから，関数呼び出しがどのタイプの式なのかを，コンパイラは知っておく必要があります。言語仕様上では，関数呼び出しは，「関数アドレスタイプの識別子」に演算子()をつけたも

のになっています。関数呼び出し自体が演算子を使った式なのです。たとえば,

```
foo ()
```

は関数アドレスタイプの式fooに演算子()を施したものです。関数アドレスタイプという言葉は，筆者が理解のために勝手に作った言葉です。C でこのような言葉が使われることはないのですが,「関数呼び出しが式である」ということの理解を助けるために作ったものです。関数アドレスタイプとは，その識別子が単体で，C 言語での関数に相当する処理を行うプログラムのアドレスを示すという意味です。アセンブラレベルでは,「ラベル」に相当します。第 1 章で，メモリにはデータやプログラムが混在して記憶されることを説明しました。この関数アドレスタイプの式は，そのメモリの中にあるなんらかの処理を行うプログラムのアドレスを示す式なのです。C 言語では，この関数アドレスタイプの式 (関数呼び出しの式) を評価するときには，決まった手順で，その識別子が教えるアドレスを呼び出して，決まった手順で返される値を式の評価の値にします。

普通の C の入門書では，こんな難解な書き方はしないでしょう。ですが，C 言語の特質は，この非常に柔軟なプログラムのアドレスの扱い方にあるので，あえてこのような難解な書き方をしてみたのです。このような形式で関数呼び出しを理解しておかないと，ほぼ必ず「関数へのポインタ」の理解が容易ではなくなるからです。C 言語では，関数アドレスタイプの式でありさえすれば，()をつけて関数として呼び出すことができるのです。後の章で述べますが,「キャスト」という「タイプの変換」を施すことで，任意の式をこの関数アドレスタイプの式に変換できます。つまり，C 言語では「任意のメモリのアドレスを関数として呼び出すことができる」のです。このようなめちゃくちゃな特質をもった言語は，C 言語くらいのものです。

「難しい話は後で考えるから，C について知りたい」という人は，とにかく関数呼び出しは「関数名称」に演算子()を施した式であって，式の値は決まった手順でコンパイラが勝手に評価してくれるものだと覚えておいてください。おおよそまちがいではない理解ですから，この方法で大多数の C のプログラムは理解できると思います。

また，関数は，数学の関数と同様に,「パラメータ」を受け取ることができます。ここで，2 つのことをコンパイラに教えてやる必要があることがわかります。1 つは関数の型，もう 1 つは関数のパラメータの数と型です。この 2 つを教える宣言を，関数のプロトタイプ宣言といいます。実例を以下に･･･。

```
/* パラメータのタイプだけ示す方式 */
int function_A(int,int);
/* パラメータのタイプと変数名を示す方式 */
int function_B(int a,int b);
```

どちらの方式でもかまいません。また，パラメータの変数名 (これを仮引数と呼ぶことがあります) を示す方式では，実際の関数の定義と変数名が一致している必要はありません。たとえば,

```
int function_B(int a, int b);

int
function_B(int x, int y)
{
  ...;
}
```

のようになります。コンパイラは，プロトタイプ宣言ではパラメータの数と型だけしかチェックしませんから，このように記述してもなんら問題ありません。2.2「関数」(P.29) で書いたように，関数には「関数的なもの」と「手続き的なもの」があります。手続き的なものは，関数自体が返す値より，その呼び出しにともなう副作用を期待するのが通常です。このような関数は，型として **void** という型を宣言してやります。また，パラメータの数が 0 の関数宣言は，パラメータ型に **void** と宣言します。たとえば，

```
void procedure_A(int,int);

int procedure_B(void);
```

のような宣言です。

関数の宣言は，関数の型と引数の型をコンパイラに通知しておくだけのものです。もし宣言がない関数呼び出しにコンパイラが出会ったときには，コンパイラは関数の型を **int** と解釈し，引数の数と型チェックはまったく行いません。これを「暗黙の宣言」と呼びますが，引数の型と数はよくまちがえるので，できればプロトタイプ宣言を行っておくべきです。

GCC は新しいタイプの C コンパイラなので，このようなプロトタイプ宣言と，引数のタイプと識別子を関数定義の()の中で行うことができます。一部の古い仕様のコンパイラでは，このような宣言と定義の方法はエラーになります。本書で扱うプログラムは，すべて新しい形式の宣言と定義を使っています。ここで，宣言と定義について少し説明をしておきます。

```
/* これは関数の宣言です */
int foo(int);

/* これは関数の定義です */
int
foo(int a)
{
  return a;
}
```

宣言というのは，関数の型と引数の型，および数をあらかじめコンパイラに通知してお

く処理です。先に述べたように，必ず行わなければならない処理とは限りません。**定義**というのは，実際に関数の処理を記述することです。宣言がなく，いきなり定義を行った場合には，それは宣言であると同時に定義になります。

2.4.2 ■ 文

式が理解できたら，文の理解は容易です。式に ';' をつけたものが**文**です。C 言語では，「空の文」は ';' 単体です。空の文なんて必要なさそうに思えますが，式の評価による副作用で処理を行う C の特質上，かなり頻繁に空の文は用いられます。

次に，文とブロックの関係を説明します。C では文法上，文が置ける場所にはブロックを置くことができます。2.4.1「C での式とは」(P.42) で **if** 文が出てきましたね。

```
if (式)
  文0
else
  文1
```

このように **if** 文は定義されていますので，**文 0** または**文 1** の部分には，単一の式に ';' をつけたものしか置くことができません。ですが，いくつかの文を '{' と '}' でくくってブロックを作ることで，その全体を 1 つの文にすることができます。ブロックの先頭では，新たに変数の宣言もできますので，非常に柔軟にプログラムを記述することができます。たとえば，ある条件が成立したときだけに必要な変数は，このブロックの先頭で宣言して使えばいいのです。**GCC** では，式と文の構造が拡張されていて，式の中に文を置くことができるといった非常に便利な拡張が施されています。**GCC** でしか使えないといったデメリットもありますが，C 言語に慣れてくると有用で便利な機能です。

■ 2.5 制御構造

2.4「式と文」では，式の説明の例として **if** 文を説明しました。この節では，その他に C 言語に備えられた，プログラムの流れを変更する制御構造について説明してみます。これらのプログラムの流れを制御する仕組みは，プログラムを記述するには必須です。ですから，ゴチャゴチャと説明を並べたてるよりは，話を先に進めて，実際にプログラムを組み上げて勉強するほうが得策でしょうから，サッと流して読んでください。

2.5.1 ■ if (式) 文0 [else 文1]

再度，**if** 文です。2.4「式と文」で説明したように，**if** 文では，式の値が 0 かそうでないかだけを判定して条件分岐します。**if** 文は「文」ですから，文の範囲があります。しばしばインデント (字下げ) にだまされて，正しい制御構造になっていないことがありますから，十分に注意が必要です。**if** 文の範囲は，**if** から始まって，**else** が省略された場合は条件成

立の実行文の終わりまで，**else** がある場合は **else** の実行文の終わりまでになります。複数の **if** が入れ子になっている場合には，**else** はプログラムの先頭方向に戻っていちばん近い **if** に対応します。

最近のエディタは賢いので，かなりの部分までプログラムの制御構造を認識してオートインデントしてくれますが，下手をすると，このオートインデントがバグの原因になりますので，**if**～**else**～の入れ子には十分に注意してください。

2.5.2 ■ while (式) 文

式が 0 でない間は文の実行を繰り返します。式が 0 でない定数式の場合には無限ループになり，式が必ず 0 の定数式の場合には文は 1 回も実行されません。この **while** 文では，しばしば条件式の副作用のみを期待して，実行される文が空文であることがよくあります。

2.5.3 ■ for (式 0; 式 1; 式 2) 文

最初に一度**式 0** を評価した後，**式 1** が 0 でない間は文を実行し，文を実行した後，**式 2** を評価することを繰り返します。**式 0**，**式 1**，**式 2** はどれも個別に省略可能です。条件式である**式 1** を省略した場合には，つねに**式 1** が 0 でないとみなされます。BASIC や Pascal の **FOR** 文に類似していますが，C 言語の **for** 文はめちゃくちゃに柔軟です。この **for** 文の使い方は「習うより慣れろ」の典型で，ときには非常に難解なプログラムとして登場することがあります。

2.5.4 ■ do 文 while (式) ;

まず文を実行して，その後式を評価します。評価した式が 0 でない間，文を繰り返し実行します。この **do**～**while(式)** が文になるためには，**while** の後に ';' が必須です。この ';' があることによって，ある種のマクロを記述するのが容易になります。

2.5.5 ■ switch (整数式) { case 定数: 文 ....}

まず整数式を評価します。その評価された値に対応した **case 定数:** の位置に分岐します。ちょっと難解なので，実例を示します。

```
switch (value)
    {
      case 0: 文0;
      case 1: 文1;
      case 2: 文2;
      case 3: 文3;
    }
```

この場合には，valueの値に応じて

0 の場合　文 0，文 1，文 2，文 3 を実行

1 の場合　**文 1**，**文 2**，**文 3** を実行
2 の場合　**文 2**，**文 3** を実行
3 の場合　**文 3** を実行
それ以外　何もしない

になります。あくまで **case** の該当する位置に分岐するだけです。分岐した後にいくつ **case** があっても，無視してそのままプログラムは実行されます。これは，**switch** の **case 定数:** という部分が単なるラベルとして認識されるという規約になっているためで，ラベル単体ではプログラムの流れになんら影響を及ぼさないからです。通常は 2.5.6「break;」で説明する **break** や，**case** 値で指定された以外の場合を意味する **default** とともに用いられます。

```
switch (value)
    {
      case 0: 文0;
              break;
      case 1: 文1;
              break;
      case 2: 文2;
              break;
      case 3: 文3;
              break;
      default:
              文4
              break;
    }
```

この場合には，valueの値に応じて

0 の場合　**文 0** を実行
1 の場合　**文 1** を実行
2 の場合　**文 2** を実行
3 の場合　**文 3** を実行
それ以外　**文 4** を実行

になります。しばしば，**break** を「故意に置かない」形式の **switch** 文が書かれます。この場合には，「なぜ故意に置かない」か，その理由をコメントで記述しておかないと，後で混乱の原因になります。

2.5.6 ▪ break;

任意の繰り返しや，**switch** 文から抜け出すのに使います。プログラムは，**break** 文に

よって抜け出した繰り返しの次の文，**switch** 文の次に移動します。

2.5.7 ■ continue;

任意の繰り返しの開始位置へプログラムを移動します。あまり使われることがない **continue** ですが，使いすぎると読みづらいプログラムになります。

■ 2.6 構造をもった変数

2.3.1「変数の実体」(P.33) で書いたように，C では本当に基本的なタイプの変数しか直接扱うことができません。その代わり，C には**構造をもった変数**を宣言し，定義する仕組みが備わっており，この性質が柔軟なプログラミングを可能にしています。そこで，この構造をもった変数について眺めてみましょう。

2.6.1 ■ 基本的な構造体

ゲームのキャラクタについて考えてみましょう。たとえば，グラディウスのようなシューティングゲームでは，あるスプライトのキャラクタを管理するには，簡単に考えても

1. **画面横方向の座標**
2. **画面縦方向の座標**
3. **キャラクタの性質**

のようないくつかの要素が考えられます。これらをいちいち個別の変数に割り当てていたのでは，管理する作業だけで死んでしまいます。そこで，これを 1 つの変数にまとめて管理することを考えます。C 言語では，**struct** という言葉を使って，いくつかの種類の変数を 1 まとめにして，管理する方法を提供しています。

```
struct {
   int pos_x;   /* 画面横方向の座標 */
   int pos_y;   /* 画面縦方向の座標 */
   int attr;    /* キャラクタの性質 */
} sp_chr;
```

これが C 言語の構造体変数の基本的な宣言の方法です。**struct** {}の{}に囲まれた範囲で，必要な変数を並べて宣言します。これらの並べられた変数を，「構造体のメンバ」と呼びます。

```
struct {
  メンバ宣言0;
  メンバ宣言1;
  メンバ宣言2;
```

```
} 変数名称;
```

これが一般的な構造体変数の記述形式です。実際に各メンバを参照するには,

```
変数名称.メンバ名称 = ...;
```

という書式で記述します。先ほどのゲームの例では,

```
sp_chr.pos_x = 0;
```

といった書式になります。構造体の内部に，さらに構造体をもつこともできます。ゲームの例では，キャラクタの性質として，たとえばキャラクタの種類や，当たり判定の有無といった性質が考えられます。このような座標とは独立した概念を１つの構造にしておいて，全体を入れ子の構造体で使うと便利です。

```
struct {
  int pos_x;  /* x座標 */
  int pos_y;  /* y座標 */
  struct {
      int chr_type;  /* キャラクタの種類 */
      int flag;      /* 当たり判定の有無 */
  } chr_attr;
} sp_chr;
```

入れ子になった構造体のメンバには，‘.’を複数使って

```
if (sp_chr.chr_attr.flag)
/* 当たり判定があるならば … */
  {
    ....;
  }
```

といった形式で参照します。

2.6.2 ■ 構造体タグとtypedef

ゲームのキャラクタについて個々の性質を考えてみますと，いくつかの共通点と相違点が混在していることに気がつくでしょう。たとえば，画面上のキャラクタについてはすべて座標で管理する必要があるでしょう。ですが，一部のキャラクタについては当たり判定は必要ないとか，一方はグラフィック画面で，他方はスプライトといったように，基本的な性質が異なるものもあります。こういった種々の情報を効率よく管理するための方法があります。たとえば，座標を１つの独立した構造体として宣言して，各個別のキャラクタは，その座標の構造体をメンバとしてもっている構造体として管理する，といった考え方です。

こういった管理を行う際は，構造体ごとに名前をつけておきたくなります。これが**構造体タグ**です。まず，座標を管理する構造体を考えます。

```
struct POS
{
  int pos_x;
  int pos_y;
};
```

上の場合，変数名称が省略され，**struct** の後に構造体タグ名称POSが追加されています。この書式は,「struct POS foo;といった宣言が，POSという名前のついた構造体の構造をもつ変数だよ」という意味づけをするために使われます。一度こういった宣言をしておけば，いちいち構造体の中身を並べて記述しなくても，

```
struct
{
   struct POS position;
   int chr_no;
} chr1;

struct
{
   struct POS position;
   int chr_no;
   int attr;
} chr2;
```

といった形式で，同じ構造をもったメンバが存在する異なった別の構造体を容易に作ることができます。そもそもこういった構造をもった形の変数は，一見異なってみえる各物体のもつ性質を細かく分類して，共通した性質をくくりだして，管理するためにあります。プログラムを組み上げるときには，まず自分がどのような事項を管理する必要があるかを洗いだして分類共通化し，必要な変数の形式を考えることから始めるのがよい方法です。

自分自身をメンバにするような構造体は宣言できません。たとえば，

```
struct ERR
{
   struct ERR er;
   int x;
} err;
```

は違反です。また，ある構造体の中で宣言する変数名称が，別の構造体のメンバ名称と同じであっても問題ありません。メンバ名称と同じ変数名称や関数名称があっても，エラーにはなりません。問題なく使えます。

また，**typedef** という言葉を使うことで，ある変数のタイプに名前をつけて，あたかもCにそういったタイプの変数が最初から存在するかのように使うことができます。先ほどのstruct POSを，**typedef** を使って書き直してみましょう。

```
typedef struct
{
  int pos_x;
  int pos_y;
} POS;
```

これは

```
struct
{
  int pos_x;
  int pos_y;
};
```

といった形をもつ構造体を，POSという名前をもつ型として宣言しているのです。この宣言をしておけば，このような構造をもっている変数は

```
POS foo;
```

といった形式で宣言して使うことができます。これは構造体の内部でも同じです。

```
struct
{
   POS position;
   int chr_no;
} chr1;

struct
{
   POS position;
   int chr_no;
   int attr;
} chr2;
```

これは先ほどの例を **typedef** で書き直した実例です。構造をもった形の変数は，「ポインタ」を理解することで，さらに応用範囲を広くすることができます。構造体のより応用的な使い方の前に，C言語を理解する上での最大の難関であるポインタを理解しましょう。

■ 2.7 ポインタ

さぁ，いよいよCでの最大の難関，**ポインタ**です。「ポインタを制する者，Cを制す」(本当かなぁ？) ですから，気合を入れて読んでください。といっても，ここまでの話がわかっ

ている人には簡単なことです。

2.7.1 ■ ポインタとはアドレスを入れておく変数である

タイトルに示したように，ポインタはメモリのアドレスを入れておく変数です。アドレスそのものではありません。ポインタでよく誤解されるのがこの点です。初期化されていないポインタ変数を使ってしまう失敗は，ポインタ変数が，あるメモリ上のアドレスを保持しておく変数であるという点を忘れているために生じます。C 言語では，2.3.3「変数の寿命」(P.41) で述べたように，関数のブロック内部で宣言された変数は値が不定です。このような値が不定の変数がポインタ変数であった場合，それを初期化しないで使うと，比較的安全性が高い **X68000/X68030** ではバスエラー程度の被害ですみますが，**MS–DOS** のような安全性の低い条件では何が起こっても不思議ではありません。

では，実際にポインタの性質についてみてみましょう。ポインタ変数は

```
変数タイプ *変数名称;
```

といった書式で宣言されます。たとえば，

```
int *integer_ptr;
```

です。これは **int** 型の変数のアドレスを保持する変数`integer_ptr`を宣言しています。ポインタ変数は，宣言しただけではどこのアドレスを指しているのかまったく不明です。必ず何かのアドレスで初期化する必要があるのです。

2.7.2 ■ 基本タイプのポインタ

C の変数のタイプは，Table 2.1「基本的な変数のタイプ」(P.36) で述べた種類のものしかありませんが，まずその種類の数だけ「????型へのポインタ」が存在します。

```
signed   char *s_char_ptr;
unsigned char *u_char_ptr;
         char *p_char_ptr;
....
```

これらはすべて，あるメモリ上のアドレスを保持する変数で，そのサイズは機種に依存します。特に 8086 系列の CPU には，メモリモデルという頭が溶けてしまうようなややこしい概念が存在するので，ポインタにいくつかの種類があったりします。ですが，**X68000/X68030** では，ポインタは必ず 32 ビットの符号なし整数として扱われますので，話が単純明快です。`int *int_ptr;`でも，`char *char_ptr;`でも，サイズは 32 ビットです。

ポインタに対する操作は

1. ポインタそのものを変更する操作

の2種類があります。

まず，**ポインタそのものを変更する操作**について説明しましょう。まずは単純な代入からみていきます。Cのライブラリ関数にmalloc()という関数があります。この関数は，現在使われていないメモリから必要な量のメモリをぶんどってきて，その先頭のアドレス，つまりポインタを返す関数です。

```
foo()
{
  int *int_ptr;
  int_ptr = malloc (512);
}
```

この例では512バイト分だけメモリをぶんどってきて，そのポインタをint_ptrに代入しています。これがポインタそのものを操作する例です。malloc()がぶんどってきたメモリは，ちゃんと周りの方々の承認を得たメモリですから，プログラムがどう使おうと自由です。ポインタ変数を宣言しただけでは，ただ単にどこかのアドレスを示すための場所4バイトが確保されただけで，自由に変更が許されているのはその4バイトだけです。

ポインタを使ってこのように確保したメモリを操作するには，2つの方法があります。1つは演算子*を使う方法，もう1つは演算子[]を使う方法です。まずは，*を用いた方法からです。

```
foo()
{
  int *int_ptr;
  int_ptr = malloc (512);
  *int_ptr = 0;
}
```

int 型へのポインタint_ptrに演算子*を施すと，ポインタ変数int_ptrが保持しているアドレスの **int** 変数を表す式になります。その変数を代入演算子=を用いて0にしているのが，このプログラムです。しばしば混同されるのが，宣言のint *int_ptrと参照の*int_ptrで使われる*の意味です。前者は「**int** 型へのポインタ」を示しており，意味上はその前にある **int** へ結合しています。後者はあくまで*演算子であり，int_ptrに結合しています。この違いを明確にしたいため，しばしば

```
foo()
{
  int* int_ptr;
  int_ptr = malloc (512);
  *int_ptr = 0;
}
```

のように，宣言では*を前の方向にずらして記述する方もおられますが，あまり一般的ではないようです。これは単純な代入なので，つまずくことはないでしょう。

次に，もう１つのポインタに対する操作である**ポインタに対する演算**について説明しましょう。ポインタに対する演算は，加減算しか許されていません。ポインタはアドレスを保持する変数なので，アドレスに対する乗算や除算は普通は考えられないでしょう。ポインタをＣで計算するときには，ポインタが何を指しているかによって，ポインタの値そのものに対する加算量や減算量が変化します。実例でみてみましょう。

```
char *char_ptr;
short *short_ptr;
int *integer_ptr;

test()
{
   char_ptr = char_ptr + 1;
   short_ptr = short_ptr + 1;
   int_ptr = int_ptr + 1;
}
```

この例では，**char**，**short**，**int** の各型へのポインタを１ずつ増やしています。

```
char_ptrが$a0000だった
short_ptrが$a0100だった
int_ptrが$a0200だった
```

場合は，関数test()を動かすと

```
char_ptrは$a0001
short_ptrは$a0102
int_ptrは$a0204
```

になります。要するに，ポインタが指しているデータの型の大きさだけ，アドレスが移動することになるのです。この大きさをスケールと呼ぶことがあります。**X68030** では，このスケールの大きさが 2，4，8 の場合，高速に扱える機械語の命令が存在しています。これが理解できたら，あるポインタからindex番目に存在するデータをアクセスするプログラムを考えてみましょう。宣言は次のようになります。

```
int pointer_val(int *int_ptr, int index);
```

引数として，**int** へのポインタint_ptrと，何番目かを表すint indexを受け取って，そのint_ptrが示しているアドレスからindex番目の **int** の値を返す関数です。それでは実際に定義してみましょう。

```
int
pointer_val(int *int_ptr, int index)
{
  return *(int_ptr + index);
}
```

ただ単にint_ptrにindexを加算して，演算子*を施せばいいのです。さて，ここで()を忘れるとどうなるでしょうか?

```
int
pointer_val(int *int_ptr, int index)
{
  return *int_ptr + index;
}
```

演算子の優先順位では，*が+より高いので，int_ptrが示すアドレスから **int** で値が取り出されて，それにindexを加えた値を返してしまいます。これは明らかにプログラムの意図とは違っていますね。普通，あるポインタから n 番目の要素を取り出したい場合には，*演算子より[]演算子を用います。この場合，関数の定義は

```
int
pointer_val(int *int_ptr, int index)
{
  return int_ptr[index];
}
```

のようになります。[]演算子はop0[op1]の形で用いられ，次のように解釈されます。

1. **op0，op1の一方はある型のアドレス形式の式であること。また他方は整数型の式であること。**
2. **1. を満たす場合には，式の値はop0を先頭アドレスとみなし，op1の値だけ進んだ位置にあるデータの値になる。**

C では，int *int_ptrとint indexの宣言を想定した場合，

```
int_ptr[index] ⟷ index[int_ptr] ⟷ *((int_ptr) + (index))
          等価                 等価
```

となっています。ここまで説明しておけば，C でのプログラミングにおいて最大の悲劇を引き起こす,「配列とポインタの混同」について解説することができます。

2.7.3 ■ 配列とポインタ

普通の C の本では，配列の話のほうが先に登場します。ですが，この本ではあえてポイ

ンタのほうを先に説明しました。これにはわけがあります。C では，配列というのはあまり役に立つデータの形式ではないからです。一般的な FORTRAN や COBOL や BASIC といった言語には，C のようなある意味では粗暴なポインタといった概念は存在しません。このような言語では，あるデータがメモリ上に並んでいる「配列」という概念は便利ですが，ポインタでの操作が発達している C では，プログラムに慣れるにつれて，配列よりポインタを使うほうが圧倒的に増えてきます。ですから，通常の解説書とは順序を入れ替えて，まずポインタを理解していただいたのです。

C では，**配列**は次のように宣言します。

```
int array[10];
```

これは要素数 10 個の **int** の配列`array`の宣言です。C では`[]`の中に要素の数を入れて宣言します。コンパイラは，この宣言で **int** のメモリを 10 個分連続したアドレスに用意します。データ数は演算子`[]`を使って指定します。先頭は`array[0]`です。`array[1]`ではないので注意してください。配列の要素番号の指定は 0 から始まりますので，この宣言ではいちばん最後の要素は`array[9]`です。

では，配列`array`から`index`番目のデータを取り出す関数を記述してみましょう。

```
int array[10];
int
array_val(int index)
{
  return array[index];
}
```

簡単ですね。ところで，`array`が`int *array;`と宣言されていた場合はどうなるでしょうか?

```
int *array;
int
array_val(int index)
{
  return array[index];
}
```

なんと関数の定義部分はまったく同じです。区別ができません。これが配列とポインタの混同を招く最大のポイントです。2.3.2「変数のスコープ」(P.37) で，C ではプログラムをしばしば複数のソースに分けてコンパイルすると書きました。ここで 2 つのソースを考えます。

test0.c は，

```
int array[10];
```

test1.c は,

```
extern int *array;
int
array_val(int index)
{
  return array[index];
}
```

となっているとします。

この 2 つのプログラムは，コンパイル/リンクして実行ファイルを作る際にはまったくエラーになりません。ところが，動作は予想したものとは完全に異なってしまいます。これが，配列とポインタの混同による悲劇のその 1 です。通常，プログラムの動作がおかしいときには，関数の動きばかりに目を奪われがちです。この手の宣言についてのまちがいは，デバッガでよほど注意深く追いかけないと気がつきません。このまちがいを防ぐには，コーディングのときに注意深くデータの宣言を行うしか方法がありません。

次に多次元の配列です。これは配列の配列と考えてかまわないのですが，これまたポインタでのアクセスとしばしば混同されます。まず，普通の多次元の配列の宣言方法からです。

```
int mult_array[10][10];
```

単純に[]で複数要素数を並べるだけです。この場合には，**int** の大きさが 4 バイトですから，$4 \times 10 \times 10$ で，400 バイトの連続したメモリをコンパイラが確保します。この配列を 1 で埋めるプログラムは，以下のようになります。

```
int mult_array[10][10];
void
fill_array_one (void)
{
  int i,j;
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      mult_array[i][j] = 1;
}
```

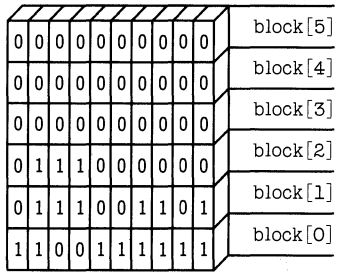
このコーディングは，BASIC でも FORTRAN でも普通にみられる方法です。C では，ここでポインタと配列の混同が起こることがあります。それは,「ポインタの配列」を使った場合です。以下のコーディングを見てください。

```
int *mult_array[10];
void
fill_array_one (void)
```

```
{
  int i,j;
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      mult_array[i][j] = 1;
}
```

これは，エラーにも警告にもならないプログラムです。またまた，配列の場合とまったく同じですね。決定的に違うのは，前のプログラムでは配列を用いているので，メモリはコンパイラが確保してくれますが，このプログラムでは，このままでは「バスエラー」になってしまう点です。なぜでしょうか? 答えは簡単です。このプログラムの`mult_array`はポインタの配列です。ポインタはメモリのアドレスですから，正しく有効なメモリのアドレスを指し示していなければなりません。「だったら配列を使えばいいでしょう。そのほうがコンパイラがメモリを確保してくれるから楽なのでは?」と思われるでしょうが，実はポインタの配列のほうが融通がきいて，C らしい処理ができるのです。

例として，テトリスもどきのゲームを考えてみましょう。Fig. 2.1 を見てください。あくまでモドキとして考えるので，難しいことは考えないでください。



●Fig. 2.1　テトリスもどきのデータ配列

テトリスは，画面の上からさまざまな形のブロックが降ってきて，それで横のラインを全部埋めたら，そのラインが消えてなくなるといった単純なゲームですよね。これは，多次元の配列で考えることができます。下の配列を見てください。

```
int block[6][10];
```

これは縦6，横10のテトリスもどきにおける画面のようすを管理する配列です。ブロックが存在する部分を'1'，存在しない部分を'0'として考えたら，あるブロック番号`block_no`について，消去できるラインかどうかは以下のように判定できるでしょう。

```
int
fill_all_one (int block_no)
{
  int i;
  for (i = 0; i < 10; i++)
```

```
    if (block[block_no][i] == 0)
      return 0;
    /* 全部埋まっていたので消去できる！ */
    return 1;
}
```

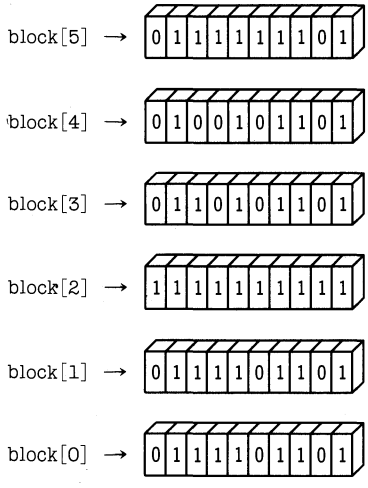
判定は簡単ですね。で，あるライン (block_noで表されています) が消去可能であると判定されたら消去を行うわけですが，この消去作業で配列を使う場合とポインタの配列を使う場合とでは，「わかりやすいアルゴリズム」を用いた場合，速度が全然違うのです。

配列を用いた場合には，あるラインが消去されたら，そのラインから上のすべてのブロックを順次コピーして移動するのが，単純でわかりやすいアルゴリズムです。ところが，この方法では，特にいちばん下のラインが消去されると，非常に多数のメモリをごっそり移動することになってしまいます。これでは，あまりに効率が悪くておもしろくありませんよね。配列を使う場合には，このメモリの大量移動を避けるためのアルゴリズムを工夫するしか方法がなく，結果としてわかりにくいプログラムになってしまいます。わかりにくいプログラムは，デバッグもメンテナンスも厄介です。

それでは，ポインタの配列を用いる場合には，なぜ高速にできるのでしょうか? この場合は，メモリの移動を実際には行わないですむからです。まず，blockをポインタの配列として宣言します。

```
int *block[20];
```

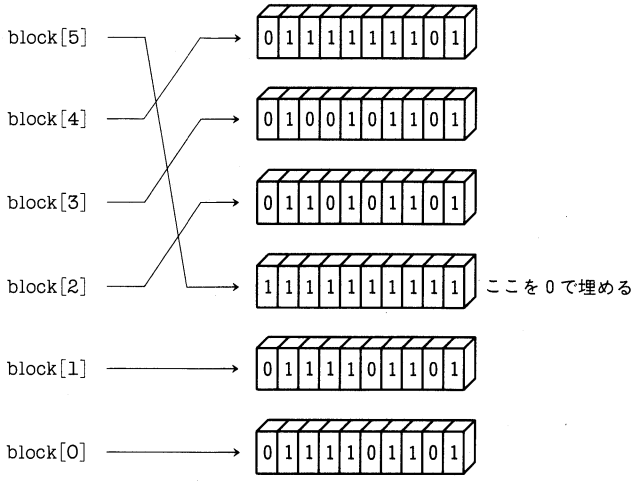
これだけでは，ポインタを 20 個並べたメモリを確保しただけですから，前に説明したように，値を 10 個入れておくメモリをmalloc()で確保しておかないと，「バスエラー」になってしまいます。ちゃんと確保さえしておけば，ラインが消去できるかどうかを判定する関数は，配列を使う場合とまったく同じ記述で判定できます。



●Fig. 2.2　ポインタの配列のイメージ

ポインタを，メモリのアドレスを示す矢印と考えると，ポインタの配列のイメージは，Fig. 2.2 のようになっていることになります。

配列の要素が，そのブロックの状態を示したメモリを指し示しています。Fig. 2.2 の状態では，`block[2]`の示す要素が消去可能です。この場合には，順次矢印の指している位置をつけ替えるだけで，多量のメモリの移動を行わずに新しい状態にできるのがすぐにわかるでしょう。Fig. 2.3 で示すように，順次ポインタが示しているアドレスをつけ替えていって，最後に`block[5]`(前に`block[2]`が示していたアドレスのメモリ) を 0 で埋めておけば，すべてうまくいくでしょう。この場合には，最大でも 5×4 で 20 バイトしかメモリの移動は起こりません。



●Fig. 2.3　ポインタのつけ替え

どうですか? おわかりいただけたでしょうか? こういった考え方を多用するのが，C のプログラムの大きな特徴です。もう 1 つ，ポインタの配列を使った場合にはメリットがあります。それはブロックの横方向に,「再度，コンパイルをしないと直せない」といった制限がない点です。配列を使った場合には，配列の大きさはコンパイル時点で決まってしまうので，このテトリスもどきの例でも，横方向を広げた場合には全体をコンパイルしなおさないと動かないプログラムになってしまいますが，ポインタの配列を使えば，横方向のサイズを変数にしておくだけで，任意の幅のテトリスもどきを作ることができます。さらに,「ポインタへのポインタ」という概念を覚えたら，縦方向の制限も除去することができるのです。この「ポインタへのポインタ」は，2.7.5「構造体へのポインタ」の後で説明をします。

他のプログラマが記述したプログラムで，ある変数名称が配列なのか，ポインタなのか，区別できないことがしばしばあります。こういったプログラムを解析するには，その変数名称に 0 を代入する処理を加えてコンパイルしてみるのです。その変数名称がポインタならば，エラーにならないでコンパイルができます。配列なら，コンパイルエラーになるでしょう。この違いが配列とポインタの違いでもあるのです。ポインタは代入式の左側に置

けますが，配列は単独では左側に置くことができません。

2.7.4 ■ 文字列とポインタ

BASIC でプログラムした経験のある人が，混乱するのが C の文字列です。C では**文字列**は，「**char** の配列」として扱われます。

```
#include <stdio.h>
main ()
{
  printf("Hello X68000\n");
}
```

これは，C の教科書のいちばん最初に出てくるプログラムですが，プログラムとしては単純でも，意味上はかなり複雑な処理になっています。'"' で囲まれた文字列は，**char** の配列として扱われますが，このプログラムでは，"Hello X68000\n"の部分が該当します。これが関数の引数や式に現れた場合には，自動的に，配列からその文字列の先頭へのchar *に変換が発生します。この場合は関数の引数ですから，コンパイラが"Hello X68000\n"の文字列をメモリ上に確保して，そのメモリを示すポインタを生成します。'\n' は，エスケープシーケンスと呼ばれる，キーボードからは直接入力できない文字コードを表すための表現方法です。C で使える主なエスケープシーケンスは，Table 2.2 のとおりです。

●Table 2.2　エスケープシーケンス

文字	意味
\n	改行
\r	復帰
\t	タブ
\\	バックスラッシュ

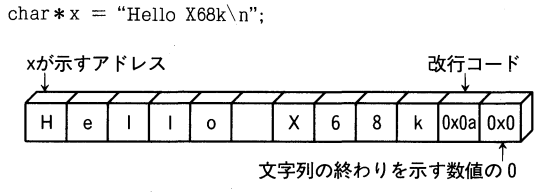
文字列が配列であることは，以下のプログラムが正しくコンパイルされることで証明できます。

```
int
hex_number_to_ascii (int hex)
{
    return "0123456789ABCDEF"[hex];
}
```

"0123456789ABCDEF"の部分は配列です。それに配列の要素を指定する演算子[]をつけるのは，コンパイラにとってはなんら不自然ではないのです。これは，BASIC を習った人間にとってはきわめて不自然にみえても，C では正しいプログラムなのです。このプログラムは，与えられた整数を 16 進数を表す文字コードで返すプログラムです。逆に，BASIC では自然にみえる以下のプログラムは，C では完全にエラーになります。

```
void
test (void)
{
  char *p = "ABCD"; /* BASICのP$と勘違い */
  char *q = "0123"; /* BASICのQ$と勘違い */
  char *x;
  /* x = "ABCD0123" を期待している */
  x = p + q;
}
```

文字列を扱うことは，Cでは配列を操作することに他ならないのです。BASICのように単純には扱えません。文字列はコンパイルのときに，READ ONLYな属性のメモリ空間にFig. 2.4のように格納されます。



●Fig. 2.4 Cにおける文字列のメモリイメージ

文字列はREAD ONLYである点に注意してください。文字列を示しているポインタの指しているメモリに書き込みを行うことで，どのような不都合が起きるのかは特定できません。

BASICプログラマを若干卒業した方が書いたCのプログラムでは，以下のようなミスがよく見られます。

```
void
test1 ()
{
  char *x = "X68000";
  char *y;
  /* strcpy を勉強できたつもり */
  strcpy (y, x);
}
```

この例では，ポインタyが正しいメモリアドレスを示していません。この場合には，何が起こっても不思議ではありません。「'y'が初期化されてないよ」と教えてあげると，

```
void
test2 ()
{
  char *x = "X68000";
```

```
  char *y = "";
  /* strcpy を勉強できたつもり */
  /* y も初期化したつもり */
  strcpy (y, x);
}
```

と修正します。ここまでBASICに毒されていると，その頭を改造するにはずいぶん骨が折れます。ポインタでメモリを操作する場合には，必ず必要なメモリを確保して，そのアドレスでポインタを初期化して使うようにしなければいけません。この場合では，正しいメモリを確保していないのです。

2.7.5 ■ 構造体へのポインタ

ポインタは，メモリのアドレスを格納する変数なので，そのアドレスが構造をもった変数を指す場合もあります。これが**構造体へのポインタ**です。構造体へのポインタを使いこなせるようになったら，もうCを自由自在に操ることができるでしょう。構造体へのポインタは，主に自分自身と同一の構造体を指し示すポインタとして用いられます。ここでは「友達の輪」を例に考えてみましょう。AさんとBさんは友達です。Bさんは･･･とつながっていく構造が友達の輪です。管理する要素として，名前，住所，電話番号を考えることにしましょう。構造体の説明で，自分自身を構造に含む構造体は作れないことは述べました。が，自分自身へのポインタを含む構造体は作れます。

```
typedef struct friend_loop {
  char *name;                /* 名前 */
  char *address;             /* 住所 */
  char *tel_no;              /* 電話番号 */
  struct friend_loop *next;  /* 友達へのポインタ */
} FRIEND;
```

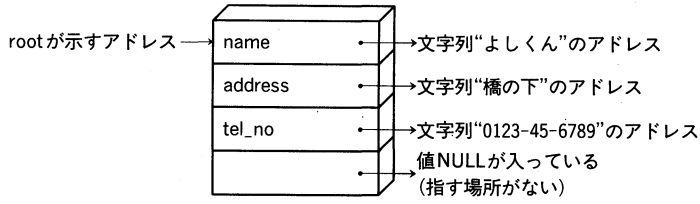
このように「友達の輪」構造体を定義します。出発点が必要なので，これは自分自身にしておきます。出発点は根っこということで，rootとでもしておきます。

```
FRIEND *root;
```

これだけでは，ポインタを作っただけなので，何もできません。まず自分自身を「輪」に加えます。

```
void
make_root ()
{
  /* まずメモリを確保します */
  root = malloc (sizeof (FRIEND));
}
```

例によってmalloc()でメモリをぶんどります。新しいCの用語sizeof()が登場しました。これは，ある変数型が何バイトのメモリを必要とするかを与える言葉です。**typedef**でFRIENDという型名を定義してありますので，sizeof()を適用して大きさを得，そのサイズ分メモリを確保します。次に自分を登録する作業です。make_root()を実行すると構造体FRIEND rootはFig. 2.5のようになります。



●Fig. 2.5　構造体 FRIEND

```
void
make_root ()
{
  /* まずメモリを確保します */
  root = malloc (sizeof (FRIEND));

  /* 名前を登録 */
  root -> name = "よしくん";
  /* 住所を登録 */
  root -> address = "橋の下";
  /* 電話番号を登録 */
  root -> tel_no  = "0123-45-6789";

  /* 友達は現在いない */
  root -> next = 0;
}
```

新しく演算子'->'が出てきました。これは構造体のポインタに適用できる演算子で，構造体へのポインタXが指している構造体の，あるメンバを特定するときに使う演算子です。「＊＊が指しているメンバ○○」と読めば，プログラムが読みやすくなります。たとえば，

```
root -> name
```

は，「rootが指しているメンバname」です。いかにもポインタらしくていいと思います。

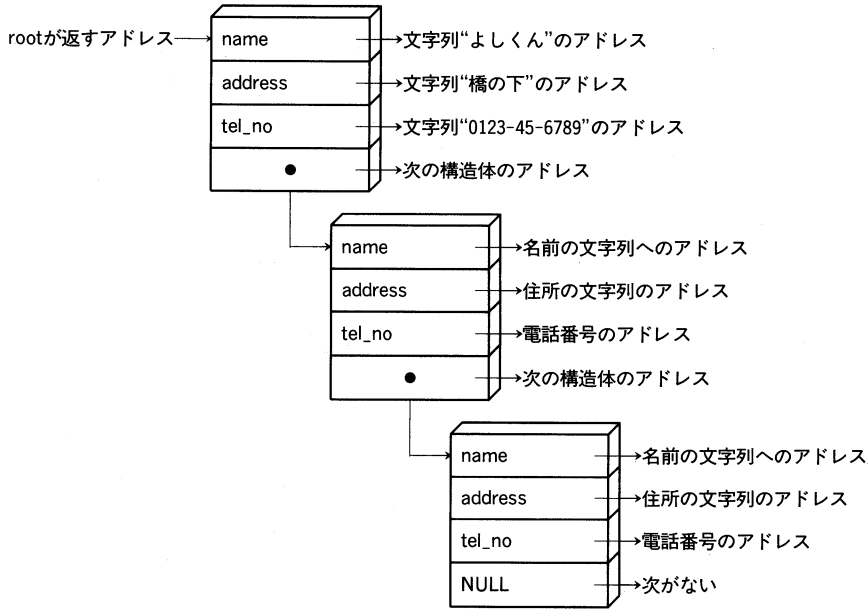
```
root -> name = "よしくん";
```

は「rootが指しているメンバnameが示すメモリのアドレス」には，文字列 "**よしくん**" が入っていることになります。後は同じように，住所や電話番号を登録しておきます。

最後に「友達は現在いない」の部分です。ポインタ変数が，現在何も有効なアドレスを

示していない場合には，0を入れておきます。コンパイラは，ポインタに対する整数0の代入や，整数0との比較には警告を発生しません。なぜなら，これはCでは「指しているものがない」という有効な値だからです。一部のコンパイラでは，ポインタへの0の代入や比較について警告を発生するかもしれませんが，それはコンパイラのほうが変なのです。

今度は，友達を加える関数を用意します。友達は，次のFig. 2.6のように構造体FRIENDに追加されます。



●Fig. 2.6　構造体 FRIEND のツリー

```
void
add_friend(char *name, char *address, char *tel_no)
{
  FRIEND *p;
  /* 友達がいない人までたどる */
  for (p = root; p -> next; p = p -> next)
    ;
}
```

まずは，根っこから友達がいない人までたどっていきます。

```
  /* 友達がいない人までたどる */
  for (p = root; p -> next; p = p -> next)
    ;
```

この書き方は，ポインタのくさりをたどる常套手段として使われる方法です。**for** 文の条件式の部分に，`p -> next`とだけ書かれています。`p -> next`が0でない間は，「その次は? その次は?」と順次たどっていくのです。「この書き方は綺麗ではない」という理由で，

厳密に書くことを強要する教科書もありますが，Cは実用言語であるという立場にたつ筆者は，こういったCらしい省略された必要十分な記述方法を好みます。最後までたどったら，後は最初にrootを作ったときと同じように，各メンバを登録すれば終わりです。

```
void
add_friend(char *name, char *address, char *tel_no)
{
  FRIEND *new_p;
  FRIEND *p;
  /* 友達がいない人までたどる */
  for (p = root; p -> next; p = p -> next)
    ;

  /* メモリを確保して */
  new_p = malloc (sizeof (FRIEND));
  /* 友達の輪にいれてあげる */
  p -> next = new_p;

  new_p -> name = name;
  new_p -> address = address;
  new_p -> tel_no = tel_no;

  /* 次はいない */
  new_p -> next = 0;
}
```

ここまでできたら，全体を作成できます。実際にコンパイルできる形に書き換えてみたものが以下のプログラムです。なお，main()は，プログラム中の任意の場所に1つだけ置くことができます。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct friend_loop {
  char *name;                /* 名前 */
  char *address;             /* 住所 */
  char *tel_no;              /* 電話番号 */
  struct friend_loop *next;  /* 友達へのポインタ */
} FRIEND;

FRIEND *root;
```

```
void
make_root ()
{
  /* まずメモリを確保します */
  root = malloc (sizeof (FRIEND));

  /* 名前を登録 */
  root -> name = "よしくん";
  /* 住所を登録 */
  root -> address = "橋の下";
  /* 電話番号を登録 */
  root -> tel_no  = "0123-45-6789";

  /* 友達は現在いない */
  root -> next = 0;
}

void
add_friend (char *name, char *address, char *tel_no)
{
  FRIEND *new_p;
  FRIEND *p;
  /* 友達がいない人までたどる */
  for (p = root; p -> next; p = p -> next)
    ;

  /* メモリを確保して */
  new_p = malloc (sizeof (FRIEND));
  /* 友達の輪にいれてあげる */
  p -> next = new_p;

  new_p -> name = name;
  new_p -> address = address;
  new_p -> tel_no = tel_no;

  /* 次はいない */
  new_p -> next = 0;
}

void
main ()
```

```
{
  FRIEND *x;
  make_root ();
  add_friend ("くんちゃん", "雲の上", "123-4565");
  add_friend ("おっきん", "海の中", "456-4579");
  add_friend ("まこちゃん", "山の頂", "456-4579");
  for (x = root; x -> next; x = x -> next)
    printf ("%sの友達は%sです\n",x -> name, x -> next -> name);
}
```

いかがでしょうか?? ポインタについて理解を深めていただけたでしょうか?? 「ポインタというのは，メモリのアドレスである」ということさえわかれば，つまずきそうなのは文字列とポインタの関係くらいのものです。怖がらないでどんどん先に進んでください。

2.7.6 ■ ポインタへのポインタ

ポインタについての最後の説明です。ポインタは，あらゆる型について存在します。ですから，**ポインタへのポインタ**も存在しています。ここまでの説明で，すでにポインタへのポインタが出てきていたのに気がついていますか? 前の2.7.5「構造体へのポインタ」で出てきたFRIEND構造体の各メンバはポインタでした。ですから，FRIEND構造体へのポインタは，ポインタへのポインタになるのです。

ただ，普通にC言語で出てくるポインタへのポインタは，

```
int **ptr_ptr_int;
```

といった形式のものが一般的です。これが，この章で何度も出てきた，配列とポインタの混同につながっているので，難解なものの代表になっています。ポインタへのポインタは，その名前のように，ポインタ (すなわち，あるメモリのアドレスを示す変数) が示すアドレスの中身が，またポインタであるような変数です。

```
char **ptr_ptr_char;
```

この場合，*ptr_ptr_char(ptr_ptr_charが示しているアドレスの中身) は，**char** へのポインタなのです。また，ptr_ptr_char[0]，ptr_ptr_char[1]，ptr_ptr_char[2]･･･も，それぞれ**char** へのポインタなのです。これは，まるでポインタの配列と同じですよね。2.7.3「配列とポインタ」(P.58) でお話ししたように，この場合も，char **と char *[10] とではアクセス手法に違いはなく，実際にアクセスするためのメモリを確保する方法が，コンパイラまかせか，自前かの違いがあるだけです。ただし，char **fooのfooは式の左側に置くことができ，char *bar[10]のbarは式の左側に置くことはできません。

ここまで説明すれば，2.7.3「配列とポインタ」で出てきた「テトリスもどき」を，任意の大きさにすることができるようになります。

```
int **block;  /* テトリスもどきデータ */
int Vsize;    /* 縦方向サイズ */
int Hsize;    /* 横方向サイズ */

void
init_block(void)
{
  int i;

  /* まずVsize分のポインタを確保する */
  block = malloc (sizeof (int *) * Vsize);

  /* Hsize分intデータのメモリを確保する */
  for (i = 0; i < Vsize; i++)
    block[i] = malloc (sizeof (int) * Hsize);
}
```

このようにメモリを確保すれば，今まで固定的に扱ってきた縦サイズや横サイズを，それぞれVsize，Hsizeで参照するようにしておくことによって，任意の大きさの(画面に表示できるかの是非はともかく)テトリスもどきを作ることができるようになります。Cでは，ハードウェアの制限に起因する固定サイズ以外は，できるだけ可変なサイズでプログラムしておいて，配列のような固定サイズデータをなるべく使わないようにしておくと，後で応用がききます。

2.7.7 ■ ポインタとメモリとI/O

ポインタは，メモリのアドレスを扱うための変数タイプです。第1章で書いたように，**X68000/X68030**では，すべてのハードウェア操作をメモリ空間で行う，「メモリマップドI/O」のハードウェア構成になっています。つまり，ポインタを使えば「**X68000/X68030**で可能なあらゆる操作」を行うことができるわけです。ということは，誤ったポインタの操作によって，ディスクを壊したりするような思いもかけない事態をも起こすことができるということです。**X68000/X68030**で採用されているCPU，68000シリーズは，このようなプログラムのバグに起因する不適切な操作を防ぐ機構が備わっています。あの「バスエラーが発生しました」の白いメッセージ窓は，この不適切な操作が行われた場合に発生する，システム保護のためのメッセージなのです。ですが，本書のテーマであるゲームプログラミングでは，この保護が邪魔になることもあります。

そのため，この保護をなくしてしまう「スーパーバイザモード」と呼ばれる状態で，ゲームなどを動作させることが多いのですが，この場合には，上記のような保護機能が低下することを頭に入れておいてください。本書では，一部のデモプログラムを除いて，できるだけユーザモードで動作するように記述してあります。

CHAPTER 3

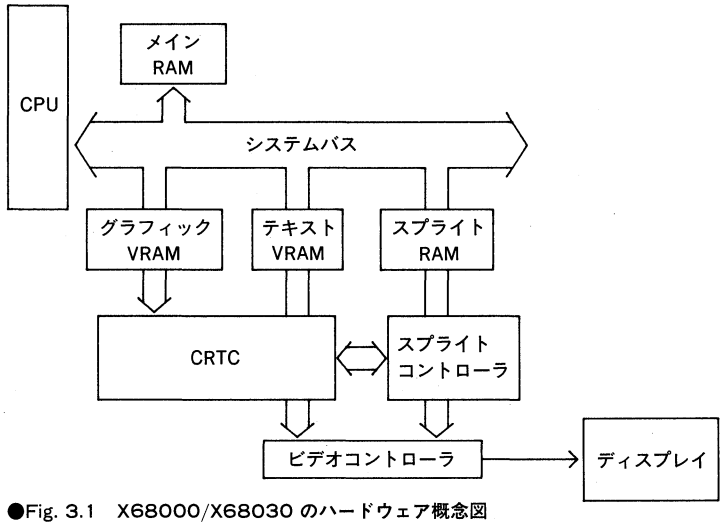
第3章

ゲームプログラミングの基礎知識

本章では，ゲームプログラミングを行うのに最低限必要なハードウェアとソフトウェアの知識について説明します。

■ 3.1 ゲームプログラミングのためのハードウェア知識

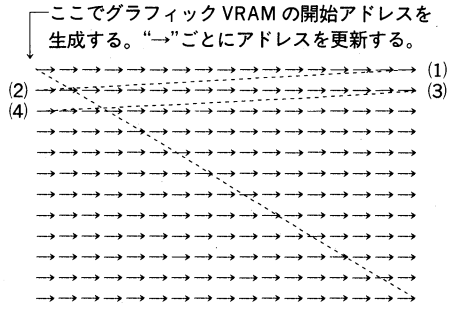
ゲームをプログラミングするためには，ハードウェアの知識が必要です。ここでは，プログラミングを進める上で最低限必要となる基本的な知識を説明します。



●Fig. 3.1　X68000/X68030 のハードウェア概念図

X68000/X68030 の CPU が何かを画面に表示するまでには，実に多くのハードウェアが働いています。Fig. 3.1 はその概念図です。まず，CPU が，表示したいデータをグラフィック VRAM に書き込みます。CRTC は，決められた時間にこのグラフィック VRAM を読み出して，そのデータに従ってディスプレイに表示を行います。ディスプレイには常時表示されているように見えますが，実際には，おおむね 1/60 秒ごとに 1 回の割合で書き換えが起こっています。この周期を垂直同期周波数といいます。

なぜ垂直同期というかというと，ディスプレイは画面の上左端から表示を開始して，画面下右端で表示を終了する仕組みになっていて，この縦方向を描く周期がこの垂直同期周波数に一致するからです。水平方向にも同様に何度も左から右になぞるように表示していますが，これも同じように水平同期周波数と呼びます。**X68000/X68030** のディスプレイには 15KHz，24KHz，31KHz などと表示されているものがありますが，これがその水平同期周波数です。CRTC の動きを詳しくみてみましょう。



●Fig. 3.2　CRTC の動作

CRTC は，画面上左端を表示し始める直前に，グラフィック VRAM のどの位置から表示を開始するか決定します。次に，水平方向に 1 ドット描くごとに，アドレスを順次更新していきます。画面の右端まで描画が終わったら，1 ドット下の左端アドレスを生成して，1 ドット下の部分を同様に描画します。これを規定回数繰り返したら，少々休憩して，再度画面の上左端からこの手順を繰り返し実行します。**X68000/X68030** では，表示する画面はグラフィック，テキスト，スプライトと 3 画面ありますが，おおまかなハードウェアの動きはこのようなものです。各画面のデータを読み出すためのアドレス生成やメモリアクセスは，CRTC やスプライトコントローラがうまくやってくれるので，ソフトウェアからあれこれ操作する必要はありません。

ここまでの説明で気がついたと思いますが，CRTC が画面を一生懸命描画している間に，CPU がグラフィック VRAM やテキスト VRAM，スプライト RAM を操作したら，どうなるのでしょうか?? この場合には，CPU の操作のタイミングによって反応が変わってきます。CRTC がすでに描画した座標だったら，次の表示期間がやってくるまで画面には反映されませんし，逆にまだ描画する前だったら，タイミング的にはシビアではありますが，CPU の操作の結果が画面に即座に反映することもあります。CPU がこのような操作を行うと，画面を見ている人には，「なめらかでない，書き換え」が見えます。

ここで，大切な結論が 1 つ出てきます。「表示している間は書き換えをしてはならない」ということです。書き換えをすることによって，なんらかの効果を得たい場合は，もちろんこの原則にあてはまりませんが，縦スクロールや横スクロール，キャラクタの発生や消去は，絶対に表示している期間は行わないのが原則です。それでは，いつ書き換えを行うのでしょうか?? 実は，何も表示していない期間が存在します。描画が画面下右端から画面上左端に戻るまでの期間，これを垂直帰線期間と呼びますが，この時間帯は，いくら書き換えを行っても不自然にはならない貴重な時間なのです。ファミコンやスーパーファミコン，アーケードゲームなどの「なめらかな動き」は，表示スピードが速いからではなく，なめらかに動くように書き換えをしているからそう見えるのです。

「表示していない期間」に書き換えを行うためには，当然書き換えを行う CPU にその期間を知る手段が用意されていないと話になりません。こうした手段のないハードウェアでは，綺麗な書き換えを行うのは不可能です。また，ゲーム画面でさまざまな効果的な演出を行うには，この非表示期間だけでなく，「現在画面のどの位置を描画しているか」を知

る手段が用意されていることも重要な要素です。幸いなことに，**X68000/X68030** では，この両方がハードウェアから通知されるように設計されています。**X68000/X68030** は，その「スプライト」機能から「はではでなゲームマシン」としての評価が高いのですが，ハードウェアの設計自体も「ゲームでほしい情報」が適切に通知できるような優れた設計になっています。

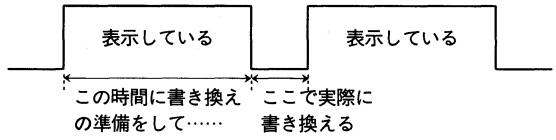
グラフィック VRAM の仕組みについても説明が必要でしょう。**X68000/X68030** のグラフィック VRAM は，某国民機種とはまったく異なった構成になっています。某国民機種では，パソコン創世期の時代から使われている，1 ビットが画面上の 1 ドットに対応した VRAM になっています。1 ビットが 1 ドットに対応した VRAM で多色のグラフィックを使う場合には，「プレーン」と呼ばれる複数の独立した VRAM を用意し，プレーンごとに設定されたドット情報を総合して色を決定します。この方法は，各プレーンを同一アドレスに割り当てておいて，別のハードウェアでプレーンを切り替えるといったことができるので，8 ビットパソコンや 8086 のような，1M バイトしかアドレス空間がないパソコンで主流になっています。このような VRAM の構成では，ある 1 ドットの色を変更するのに，場合によってはすべてのプレーンの VRAM をアクセスする必要があります。このために生じる速度の低下をハードウェアで解決しようとして，さまざまな LSI がくっついて肥大しているのが某国民機種の姿でもあります。

一方，**X68000/X68030** では，最初から 16M バイトのアドレス空間がありますので，贅沢に 2M バイトの空間を VRAM の領域として割り当て，1 ドット 1 ワードの VRAM を備えています。画面の 1 ドットの色を変更するには，該当するアドレスの 1 ワードを変更すればよいというシンプルな構成になっています。また，スクロールは，ハードウェアで高速に行うことができるようになっています。さらに，**X68000/X68030** には，このグラフィック画面とは独立して，某国民機種と同じ方式のグラフィック画面がテキスト画面として備わっています。本書では，テキスト画面を扱うことは行っていませんが，**X68000/X68030** には，ハードウェア上は某国民機種のグラフィック画面とほぼ同等で，プレーン切り替えの速度低下を抑制するハードウェアも備わっています。ゲームではこの画面もしばしば用いますが，やはり 1 ビットが画面上の 1 ドットに対応したデータは扱いにくいのと，スクロールが円筒スクロールで若干工夫が必要なため，扱わないことにしました。VRAM のハードウェアについては，本書の性格がハードウェア解説書ではなく，C プログラミングの本ということで，詳しくは扱いません。ですが，基本的な事項なので，「Inside X68000」(ソフトバンク) や「X68000 データブック」(小学館) などの参考書を適宜参照しながら読み進めていただくことを希望します。

■ 3.2 ゲームの基本アルゴリズム

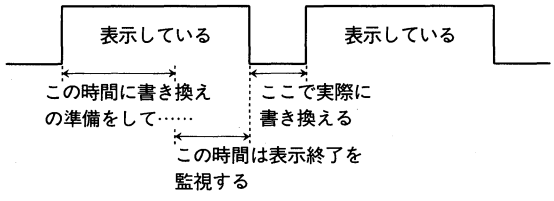
時間を横軸にして表示期間と非表示期間について考えると，Fig. 3.3 のようになります。ブラウン管に実際に表示している期間は，CPU はひたすら次の書き換えに備えて準備をし

ています。表示が終わった瞬間に，CPU は実際に書き換えを行います。再び「表示期間」になると，CPU は準備状態に入り，「非表示期間」になるとまた書き換えを行う…ということを繰り返します。とにかく，見た目に「なめらかに動く」印象を与えるには，ディスプレイに表示している時間帯に「表示物」を操作してはいけないということです。



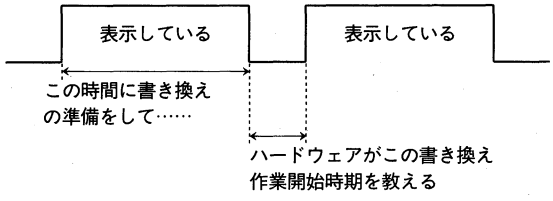
●Fig. 3.3　表示期間と非表示期間

ここで問題になるのが，表示期間終了を知る手法です。1 つ目の方法は Fig. 3.4 に示す方法です。いたって単純明快な方法で，表示するデータを準備できるまで仕事をし，やおら表示期間が終わったのを確認して実際に書き換えを行うというものです。人間のする分担作業による仕事にたとえれば，自分の担当分の仕事を終えるまでは，周りのことなどいっさい見ずに仕事をして，自分の担当分が終わったら，別の仕事をする人をじーっと眺めているといったイメージになります。別の人の作業が終わったというのが，表示期間終了に該当します。



●Fig. 3.4　監視による非表示期間の検出方法

もう 1 つの方法が，割り込みによる方法です。これは Fig. 3.5 のように，書き換え準備を常時行っていて，表示期間が終わったらハードウェアがそれを通知するので，その通知を受け取って書き換え作業を行います。先ほどの流れ作業の仕事にたとえれば，次の作業をする人が「準備ができたぞ，あんたの作業は終わったか?」とつついてくれるといったイメージです。



●Fig. 3.5　割り込みによる非表示期間の検出方法

Fig. 3.4，Fig. 3.5 どちらもそうですが，自分の作業が終了する前に，相手の作業が終わってしまったら，相手は待つことになります。「相手の作業が終わる」というのは，表示期間が終わってしまうことに相当しますので，次の表示期間終了まで書き換えが遅れるこ

とになります。これがゲームでよくみられる**処理落ち**です。**X68000/X68030** でスプライトを使ったゲームでは，(一部のタコゲームを除いて) **処理落ち**が起きない状態にある場合は **X68000**，**X68000 XVI**，**X68030** のどれで動かしても，見た目は同じ速度で動くのです。何が異なるのかといえば，**処理落ち**が起こる限界点の高さが違っているだけです。画面に表示するデータの加工が終わったら，CPU は単にディスプレイが非表示期間になるのを待っているだけです。この待ち時間が **X68000**，**X68000 XVI**，**X68030** では異なっているのです。**X–BASIC** で作ったプログラムを BASIC to C したゲームは，目に見えて動作が速くなりますが，そういった方法で動作速度を稼いだり，ゲームバランスを調整したりすると，機種ごとの動作速度が変わってしまう事態に陥ります。プロトタイピングにはよい方法でしょうが (キャラクタの表示テストくらいでしょうけど)，本格的にゲームを作成するにはあまりお勧めできる方法ではありません。最近のゲームは，**X68030** のような処理速度に余裕のあるマシンでは単位時間あたりの書き換え数を増やしたりして，高級機メリットを出しているものもあります。

なお，**X68000/X68030** では，「非表示期間検出」を行うのに監視を行う方法，割り込みによる方法，のどちらでも実現できます。

■ 3.3　割り込みとは(例外処理)

いきなり「割り込み」という言葉が登場して，混乱をなさったかもしれませんね。**割り込み**というのは，ある時点で CPU が行っている処理を，なんらかの原因で一時中断して，別の処理を行うような処理をいいます。なんらかの原因というのは，ハードウェアに起因するものと，ソフトウェアに起因するものの 2 種類があります。CPU 68000/680EC30 では，このような割り込み処理を「例外処理」と普通呼びます。

デバッグ中によく遭遇する「バスエラー」や「アドレスエラー」もこの例外処理で，ソフトウェアに起因する割り込みです。これらは **X68000/X68030** の OS，**Human68k** では致命的なエラーなので，それまで行っていた処理は再開されることなく強制的に中断されます。

一見何もしていないように見える状態，COMMAND.X がカーソルをチカチカさせて待っている状態でも，実は CPU はたくさんの割り込み処理を裏で密かに行っています。たとえば，マウスの処理や，キーボードの処理などは，すべて割り込みで常時行われています。割り込み処理を行う CPU は，「ドラマを見ながら全自動洗濯機で洗濯をしている奥さん」状態と同じです。全自動洗濯機は自分で洗濯の仕事をこなしますが，洗濯物を洗濯機に投入したり，洗い終わって干したりするのは奥さんの役目です。これは，ちょうど CRTC が VRAM をディスプレイに表示するのは自動で行い，画面の書き換えは CPU が行うのと同じです。洗濯機にはブザーがついていて，洗濯が終わったらそれを鳴らして奥さんに通知します。奥さんはそれを聞いて，次の仕事を洗濯機にまかせる処理を行うわけです。これとまったく同じで，先ほど説明した垂直帰線期間がくると，CRTC は CPU に対してそ

れを通知する機能をもっているのです。奥さんが「テレビに夢中」で，ブザーを無視することもあるでしょう。同じようにCPUでも，この通知を無視したり，最初から通知がこない状態にすることもできるようになっています。奥さんは，実は洗濯と同時に料理もしているかもしれません。こちらはお料理タイマで時間を計測しているとします。お料理タイマと洗濯機のブザーが同時に鳴る事態も起こるでしょう。この場合，両方同時にはできませんから，どちらか優先順位が高いほうの処理をするでしょう。これとまったく同様に，同時に起こった割り込みには「優先順位」があり，CPUはこの優先順位に従って処理を順次行います。さらに，この優先順位のレベルを指定して，「この優先順位は受けつけ，これより低い割り込みは無視」といった割り込みレベルの指定もできます。

これら「割り込みのマスク」や「割り込みの優先順位設定」は，今すぐ必要な知識ではありません。ですが，知っておいて損はないので，頭の片隅にでも置いておいてください。今，知っておいてほしいのは，現在行っている処理を一時中断して，別の処理をハードウェアからの信号で行うことができるという点です。

■ 3.4 スクロールからやってみましょう

いよいよ本格的にCのプログラミングを始めます。まずは，比較的簡単な画面のスクロールからです。**X68000/X68030**では，スクロールはハードウェアでサポートされていますから，普通のパソコンのゲームプログラミングではいちばん難しい処理とされる画面スクロールが，最も簡単な基礎プログラミングになります。

3.4.1 ■ グラフィックのスクロール

X68000/X68030は512KバイトのVRAM，同じく512KバイトのテキストVRAMをもっています。これらの通常の画面に加えて，**X68000/X68030**ではスプライト機能の補助機能として，BG画面と呼ばれる画面を最大で2画面とることができます。

VRAMもテキストVRAMもビットマップ方式で，パソコンとしてはかなり贅沢な構成になっています。このハードウェアを「ゲームの作成」という観点から考えると，実際に使える画面モードは限られてきます。いわゆる6万色表示モードは，自然画グラフィックの表示には向いていますが，ゲームで使うにはグラフィック画面が1面しかとれないことと，扱うデータ量が肥大化するためにあまり向いていません。実際によく使われているのは256色2画面モードか，16色4画面モードでしょう。また，表示ドット数は256×256ドットがほとんどです。512×512モードでは，画面全体を綺麗にスクロールさせるのに少々工夫が必要です。表示色については，たとえ16色モードであっても，6万色中の16色ですから，実際の表現力はこれで必要十分です。

最近のゲームでは，「多重スクロール」は常識になっていますが，**X68000/X68030**のハード構成では，この多重スクロールを非常に容易に実現できます。Fig. 3.6はグラフィック関係のスクロールに関するI/Oの抜粋です。

screen 0 X																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E80018	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@	@
screen 0 Y																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E8001A	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@	@
screen 1 X																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E8001C	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@
screen 1 Y																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E8001E	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@
screen 2 X																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E80020	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@
screen 2 Y																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E80022	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@
screen 3 X																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E80024	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@
screen 3 Y																
	b15	b14	b13	b12	b11	b10	b09	b08	b07	b06	b05	b04	b03	b02	b01	b00
$E80026	*	*	*	*	*	*	*	@	@	@	@	@	@	@	@	@

これらのレジスタは WRITE ONLY です。*のビットは無効です。

●Fig. 3.6　X68000/X68030 のグラフィックスクロールレジスタ (CRTC)

実際的な C のソースで，Fig. 3.6 のレジスタをアクセスすることを考えてみます。List 3.1 が実際のコーディング例です。**GCC** は ANSI コンパイラですので，スクロール量を保持する変数dataを，ポインタで直接スクロールレジスタアドレスに代入しています。68000/680EC30 では，1.5「メモリの話」(P.15) で説明したように，8086 系列にみられるような独立した I/O 空間は存在しないので，グラフィックスクロールレジスタを List 3.1 のようにしてアクセスできます。

たとえば，List 3.1 の関数scroll()を見てください。

```
29: *reg = data;
```

この 1 行で 8 個のレジスタに値を書き込むことができます。注意しなければならないのは，このレジスタは WRITE ONLY で，読み出しはできないことです。ですから，前にセットした値は，変数として別に保存しておかなければ，失われてしまいます。List 3.1 のように，I/O はメモリをポインタで扱うのと同じ手段で操作できますが，読み出しと書き込みで意味が違うものや，書き込みだけ，読み出しだけがそれぞれ可能といった，一般のメモリと異なる動作をしますので，ハードウェアを十分理解してコーディングする必要があります。通常のメモリであれば，

```
*reg = data;
```

として書き込んだデータは

```
data = *reg;
```

で取り出せますが，グラフィックスクロールレジスタはWRITE ONLYなので，dataに格納される値は不定です。

List 3.1

```
 1: #include <iocslib.h>
 2:
 3: /*
 4:   CRTCを構造体としてアクセスします
 5: */
 6:
 7: typedef struct {
 8:   short sc0_x_reg;
 9:   short sc0_y_reg;
10:   short sc1_x_reg;
11:   short sc1_y_reg;
12:   short sc2_x_reg;
13:   short sc2_y_reg;
14:   short sc3_x_reg;
15:   short sc3_y_reg;
16: } CRTC_REG;
17:
18: #define SET_SC0_X(CRTC,VAL) ((CRTC).sc0_x_reg = (VAL),
                                 (CRTC).sc1_x_reg = (VAL))
19: #define SET_SC0_Y(CRTC,VAL) ((CRTC).sc0_y_reg = (VAL),
                                 (CRTC).sc1_y_reg = (VAL))
20: #define SET_SC1_X(CRTC,VAL) ((CRTC).sc2_x_reg = (VAL),
                                 (CRTC).sc3_x_reg = (VAL))
21: #define SET_SC1_Y(CRTC,VAL) ((CRTC).sc2_y_reg = (VAL),
                                 (CRTC).sc3_y_reg = (VAL))
22:
23: CRTC_REG data;
24:
25: void
26: scroll ()
27: {
28:   register CRTC_REG *reg = (CRTC_REG *) 0xe80018;
29:   *reg = data;
30: }
31:
32: void
33: main()
34: {
35:   B_SUPER (0);
```

```
36:   SET_SC0_X (data, 10);
37:   SET_SC0_Y (data, 20);
38:   SET_SC1_X (data, 30);
39:   SET_SC1_Y (data, 40);
40:   scroll ();
41: }
```

いきなり，具体的なCのソースが出てきてしまいました。Cを少しでも扱ったことがあれば簡単なソースですが，「初めて」だと難しいでしょう。List 3.1では，今までまったく説明していないCのプリプロセッサの機能を使っています。プリプロセッサ命令はすべて'#'で始まります。

```
1: #include <iocslib.h>
```

これは，**X68000/X68030**のIOCSを呼び出すための，関数群のプロトタイプ宣言などが収められたファイルをここに挿入しなさいという命令です。`#include <`*filename*`>`のように'<'と'>'で囲んで*filename*を指定した場合には，コンパイラプリプロセッサの規定の範囲で，ファイルを捜し，それを挿入します。`#include "`*filename*`"`のように'"'で囲んだ場合には，カレントディレクトリからその*filename*を捜して，そこに挿入します。

```
18: #define  SET_SC0_X(CRTC,VAL) ((CRTC).sc0_x_reg = (VAL),
                                  (CRTC).sc1_x_reg = (VAL))
```

これは引数つきマクロと呼ばれるものです。一般的な書式は，

```
#define MACRO_NAME(ARG1,[ARG2,...])  ···
```

です(注意:'MACRO_NAME'と'('の間にはスペースやタブを入れることはできない)。'···'に何が入るかは，さまざまなバリエーションがあって特定できません。

たとえば，

```
#define TEST(VAL)     test(VAL)
```

の場合，これ以降に

```
TEST(x)
TEST(10)
TEST("as")
TEST(XX)
```

のような記述(これをマクロの呼び出しといいます)があると，プリプロセッサはそれぞれ

```
test(x)
test(10)
test("as")
test(XX)
```

と置き換えを行います。マクロの呼び出しは，一見普通のCの関数の呼び出しにみえますが，引数の数以外は特にチェックしません。この例の場合には，文字列を渡したり整数を渡したりしていますが，これらはプリプロセッサの段階では何の警告にもエラーにもなりません。

極端な例を書いてみます。たとえば，

```
#define TEST(VAL)
```

とだけ記述した場合にはどうなるのでしょうか?

```
TEST(x)
TEST(10)
TEST("as")
TEST(XX)
```

上のマクロの呼び出しは，引数もろともすべて消えてなくなります。

また，

```
#define TEST(VAL)  foo
```

とだけ記述した場合にはどうなるのでしょうか?

```
TEST(x)
TEST(10)
TEST("as")
TEST(XX)
```

これらは，すべて

```
foo
foo
foo
foo
```

となってしまいます。プリプロセッサは，単純に「置き換え」を行うプログラムだと考えてもいいのです。実際にList 3.1の置き換え結果をみてみましょう。

List 3.2

```
# 1 "test.c"
# 1 "d:\\include\\iocslib.h" 1

                    ⋮

# 1 "d:\\include\\cdecl.h" 1
```

```
                                    ⋮
# 1 "test.c" 2
                                    ⋮
typedef struct {
  short sc0_x_reg;
  short sc0_y_reg;
  short sc1_x_reg;
  short sc1_y_reg;
  short sc2_x_reg;
  short sc2_y_reg;
  short sc3_x_reg;
  short sc3_y_reg;
} CRTC_REG;
                                    ⋮
CRTC_REG data;

void
scroll ()
{
  register CRTC_REG *reg = (CRTC_REG *) 0xe80018;
  *reg = data;
}

void
main()
{
        _iocs_b_super  (0);
   ((data).sc0_x_reg = ( 10),(data).sc1_x_reg = ( 10)) ;
   ((data).sc0_y_reg = ( 20),(data).sc1_y_reg = ( 20)) ;
   ((data).sc2_x_reg = ( 30),(data).sc3_x_reg = ( 30)) ;
   ((data).sc2_y_reg = ( 40),(data).sc3_y_reg = ( 40)) ;
  scroll ();
}
```

たった数行のプログラムが，これほどにまで大きくなってコンパイラに渡されるのです。この例ではインクルードファイルはiocslib.hだけでしたが，普通，プログラムでのインクルードファイルの数はもっと多いので，小さなソースでも，コンパイラが実際に処理するソースの量は飛躍的に増大しています。プリプロセッサの話は，これでいったんおしまいにします。

List 3.2では，関数main()の中身が元のものとはずいぶん変わっていますね。プリプロセッサによって置き換えられた結果，このように変更されたのです。

```
((data).sc0_x_reg = ( 10),(data).sc1_x_reg = ( 10)) ;
```

は

```
36:   SET_SC0_X (data, 10);
```

が展開されてできた結果です。dataは構造体ですから，そのメンバのアクセスには‘.’演算子を使います。

```
(data).sc0_x_reg = ( 10)
```

の部分は，data.sc0_x_regに 10 を代入する式です。また，

```
(data).sc1_x_reg = ( 10)
```

も同様に，data.sc1_x_regに 10 を代入する式です。この 2 つの式を‘,’演算子で並べて，全体を冗長な()で囲んだ式に‘;’がついて文になっています。‘,’演算子は

```
exp0,exp1[,exp2...]
```

のように用います。意味は「exp0を評価して捨て，exp1を評価し･･･」と，順番に左から式を評価しては捨て，最終的に，式の値はいちばん右側で最後に評価された式の値になります。‘,’演算子は，1 つの式でいくつもの副作用を期待するときに便利です。ここで誤解を招かないように注意しておきます。関数の呼び出しのときに用いられる‘,’と，この演算子‘,’とは別のものです。前者の場合，式の評価の順序は規定されていません。

関数scroll()では，CRTC のスクロールレジスタに直接値を書き込んでいます。

```
28:   register CRTC_REG *reg = (CRTC_REG *) 0xe80018;
```

この文は，宣言と同時に値の初期化を行います。「レジスタ属性の構造体CRTC_REGへのポインタ変数regを，構造体CRTC_REGへのポインタ値 (つまりメモリアドレス)0xe80018で初期化する」という意味になります。

```
29:   *reg = data;
```

これは，regが示すアドレスの構造体CRTC_REGに，構造体CRTC_REG dataを代入しています。ここで，

*regは構造体CRTC_REG
dataは構造体CRTC_REG

ですから，これは「構造体の代入」です。昔のコンパイラでは，このような構造体の代入はできませんでした。ですが，**GCC** や **XC** のような新しいコンパイラでは，これは正しい代入です。

さて，これで実際にスクロールができるのですが，このままでは何も表示するデータが

ないので，おもしろくもなんともありません。ゲームのような，ある絵を表示しながらのスクロールの実例をみるために，もう少し勉強を進めましょう。

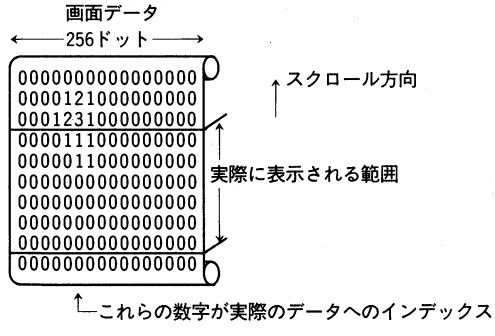
3.4.2 ■ マップによるグラフィックのスクロール

ゲームでは，一連の画面を，ちょうど巻物を窓からのぞくように画面に表示させて，進めていくのが一般的です。この画面データは，通常は適当な単位に分割して，データ量を減少させておきます。ここでは，私が昔作ったゲーム用のグラフィックエディタが 16×16 単位なので，この形式で話を進めていきます。逆にいえば，基本的に行うべきことは同じなのですが，実際のインプリメントでは多種多様なやり方があることになります。たとえば，メモリが許せば，あるステージ全体のデータをオンメモリで直接転送してもいいのです。

Fig. 3.7 を見てください。画面を 16 ドット単位に分割して縦方向の「巻物」を作るのが，ごく普通に考えられる縦スクロールのデータ形式です。ここで紹介する例では，画面のデータはたった 9 種類しかないので (原理的にはもっと増やせます)，これを 16 文字の文字列へのポインタ配列にして，次のように定義します。

```
char *map[] =
   { "0000000000000000",
     "0000001112000000",
     ....
   };
```

たとえば，map[0][0]は，上の例では '0' (これは数値ではなく文字コードです) になっていますので，パターン番号の 0 番を書き込みます。map[0][1]，map[0][2]と横方向に 16 個並べば，表示範囲は 256 ドットになります。このような，特定のパターンを VRAM に張り付けるような方法を「タイルを使った」とよく表現します。実際，本当にいろいろなパターンをもったタイルを，VRAM に張り付けるような感じで描いていきます。



●Fig. 3.7 マップデータのイメージ

List 3.3

```
1: #include <stdio.h>
```

```
 2: #include <iocslib.h>
 3: #include <interrupt.h>
 4:
 5: #ifndef SCR_UNIT
 6: #define SCR_UNIT 1
 7: #endif
 8:
 9: /* スクロールマップデータ */
10: static char *map_data[] = {
11:   "                ",
12:   "                ",
13:   "                ",
14:   "                ",
15:   "                ",
16:   "                ",
17:   "                ",
18:   "                ",
19:   "                ",
20:   "                ",
21:   "                ",
22:   "                ",
23:   "                ",
24:   "                ",
25:   "                ",
26:   "                "
27:   "                ",
28:   "                ",
29:   "      08888881  ",
30:   "      67    67  ",
31:   "      67    67  ",
32:   "      67    67  ",
33:   "      67    67  ",
34:   "      67    67  ",
35:   "      67    67  ",
36:   "      67    67  ",
37:   "      23    23  ",
38:   "                ",
39:   "  041           ",
40:   "  687  01  01    ",
41:   "  687  23  23    ",
42:   "  687           ",
43:   "  687    01     ",
44:   "  687    23     ",
45:   "  687           ",
46:   "  687           ",
47:   "  687  01   01  ",
```

```
48:   "  687  23   67  ",
49:   "  687       67  ",
50:   "  687       67  ",
51:   "  6884441   67  ",
52:   "  2555553   67  ",
53:   "            67  ",
54:   "       041  67  ",
55:   "       687  67  ",
56:   "       687  67  ",
57:   "       687  67  ",
58:   "       687  67  ",
59:   "       687  67  ",
60:   "       687  67  ",
61:   "       687  23  ",
62:   "       687      ",
63:   "  01   687  01  ",
64:   "  67   687  23  ",
65:   "  67   687      ",
66:   "  67   687  01  ",
67:   "  67   687  23  ",
68:   "  67   687      ",
69:   "  67   687      ",
70:   "  67   687      ",
71:   "  67   687      ",
72:   "  67   6884441  ",
73:   "  67   2555553  ",
74:   "  67            ",
75:   "  67   044441   ",
76:   "  23   688887   ",
77:   "      0888883   ",
78:   "     0888883    ",
79:   "    0888883     ",
80:   "   0888883      ",
81:   "   688883       ",
82:   "   25553        ",
83:   "                ",
84:   "  01        01  ",
85:   "  23   01   23  ",
86:   "       23       ",
87:   "  01        01  ",
88:   "  23   01   23  ",
89:   "       23       ",
90:   "  01        01  ",
91:   "  23   01   23  ",
92:   "       23       ",
93:   "  01        01  ",
```

```
 94:   "  23         23  ",
 95:   "                 ",
 96:   "                 ",
 97:   "   01  01  01    ",
 98:   "   67  67  67    ",
 99:   "   67  67  67    ",
100:   "   67  67  67    ",
101:   "   67  67  67    ",
102:   "   67  67  67    ",
103:   "   67  67  67    ",
104:   "   67  67  67    ",
105:   "   67  67  67    ",
106:   "   67  67  67    ",
107:   "   67  67  67    ",
108:   "   23  23  23    ",
109:   "                 ",
110:   "   04444441      ",
111:   "   68888887  01  ",
112:   "   68888887  23  ",
113:   "   68888887      ",
114:   "   68888887  01  ",
115:   "   25555553  23  ",
116:   "                 ",
117:   "       08888881  ",
118:   "       67    67  ",
119:   "       67    67  ",
120:   "       67    67  ",
121:   "       67    67  ",
122:   "       67    67  ",
123:   "       67    67  ",
124:   "       67    67  ",
125:   "       23    23  ",
126:   "                 ",
127:   "           01    ",
128:   "           67    ",
129:   "           67    ",
130:   "           67    ",
131:   "           67    ",
132:   "           67    ",
133:   "  04441    67    ",
134:   "  68887    67    ",
135:   "  68887    67    ",
136:   "  68887    67    ",
137:   "  68887    67    ",
138:   "  68887    67    ",
139:   "  68887    67    ",
```

```
140:   "  68887   67    ",
141:   "  68887   67    ",
142:   "  68887   67    ",
143:   "  68887   23    ",
144:   "  25553         ",
145:   "                ",
146:   "                ",
147:   "      08888881  ",
148:   "      67    67  ",
149:   "      67    67  ",
150:   "      67    67  ",
151:   "      67    67  ",
152:   "      67    67  ",
153:   "      67    67  ",
154:   "      67    67  ",
155:   "      23    23  ",
156:   "                ",
157:   "          01    ",
158:   "  01      23    ",
159:   "  23   01        ",
160:   "       23       ",
161:   "  01         01  ",
162:   "  23         23  ",
163:   "        01      ",
164:   "        23      ",
165:   " 01              ",
166:   " 23        01    ",
167:   "          23    ",
168:   "                ",
169:   "*"
170:   };
171:
172: /* グラフィックデータロードバッファ */
173: struct {
174:   int pal[256];
175:   unsigned char g_data[11][256];
176: } gdata;
177:
178: /* 現在の書き込みデータ管理用カウンタ */
179:
180: int map_line;
181:
182: /* VRAM 書き込みアドレス */
183:
184: unsigned short *vram = (unsigned short *) (0xc80000 - 0x400*16);
185:
```

```
186: /* 16*16 矩形を指定された VRAM アドレスに書き込み */
187:
188: static void write_vram (unsigned short *vr, unsigned char *dat)
189: {
190:   int i;
191:   for (i = 0; i < 16; i++)
192:     {
193:       unsigned short *p = vr;
194:       *p++ = *dat++;
195:       *p++ = *dat++;
196:       *p++ = *dat++;
197:       *p++ = *dat++;
198:       *p++ = *dat++;
199:       *p++ = *dat++;
200:       *p++ = *dat++;
201:       *p++ = *dat++;
202:       *p++ = *dat++;
203:       *p++ = *dat++;
204:       *p++ = *dat++;
205:       *p++ = *dat++;
206:       *p++ = *dat++;
207:       *p++ = *dat++;
208:       *p++ = *dat++;
209:       *p++ = *dat++;
210:       vr += 0x200;
211:     }
212: }
213:
214: /* 指定矩形VRAMをクリア */
215:
216: static void clr_vram (unsigned short *vr)
217: {
218:   int i;
219:   for (i = 0; i < 16; i++)
220:     {
221:       unsigned short *p = vr;
222:       *p++ = 0;
223:       *p++ = 0;
224:       *p++ = 0;
225:       *p++ = 0;
226:       *p++ = 0;
227:       *p++ = 0;
228:       *p++ = 0;
229:       *p++ = 0;
230:       *p++ = 0;
231:       *p++ = 0;
```

```
232:       *p++ = 0;
233:       *p++ = 0;
234:       *p++ = 0;
235:       *p++ = 0;
236:       *p++ = 0;
237:       *p++ = 0;
238:       vr += 0x200;
239:     }
240: }
241:
242: /* マップデータに基づいてグラフィックVRAMにデータを書き込む */
243: static
244: void write_graph (char *dat)
245: {
246:   unsigned short *vr = vram;
247:   while (*dat)
248:     {
249:       if (*dat == ' ')
250:         clr_vram (vr);
251:       else
252:         {
253:           int i = *dat - '0';
254:           write_vram (vr, gdata.g_data[i]);
255:         }
256:       dat++;
257:       vr += 16;
258:     }
259:   vram -= 0x200 * 16;
260:   if (vram < (unsigned short *) 0xc00000)
261:     vram =  (unsigned short *) (0xc80000 - 0x400*16);
262: }
263:
264: /* マップカウンタを初期化する */
265: /* ついでにscreen 2 も背景で埋めておく */
266: static
267: void init_map_line (void)
268: {
269:   int i,j;
270:   unsigned short *vr = (unsigned short *) 0xc80000;
271:   for (i = 0; map_data[i][0] != '*'; i++)
272:     ;
273:   map_line = i - 1;
274:   for (i = 0; i < 16; i++)
275:     {
276:       unsigned short *var = vr;
277:       for (j = 0; j< 16; j++)
```

```
278:         {
279:           write_vram (var, gdata.g_data[9]);
280:           var += 16;
281:         }
282:       vr += 0x200 * 16;
283:     }
284: }
285:
286: /* パレットを設定する */
287: static void
288: init_palet ()
289: {
290:   int i;
291:   int *dat = (int *)&gdata;
292:   for (i = 0; i < 256; i++)
293:     GPALET (i, *dat ++);
294: }
295:
296: /* データをロードして初期化する */
297: static void
298: load_data_and_init ()
299: {
300:   FILE *fp = fopen ("pattern.dat","rb");
301:   if (fp == NULL)
302:     {
303:       printf ("File can't open\n");
304:       exit (1);
305:     }
306:
307:   if (fread ((void *)&gdata, sizeof (char), sizeof (gdata), fp)
          != sizeof (gdata))
308:     {
309:       printf ("File read error\n");
310:       fclose (fp);
311:       exit (1);
312:     }
313:   fclose (fp);
314:   CRTMOD (10);
315:   G_CLR_ON ();
316:   B_CUROFF ();
317:   init_palet ();
318:   init_map_line ();
319: }
320:
321:
322: /* スクロールレジスタと同等の構造体 */
```

```
323:
324: typedef struct {
325:   short sc0_x_reg;
326:   short sc0_y_reg;
327:   short sc1_x_reg;
328:   short sc1_y_reg;
329:   short sc2_x_reg;
330:   short sc2_y_reg;
331:   short sc3_x_reg;
332:   short sc3_y_reg;
333: } CRTC_REG;
334:
335: /* スクロール用データ */
336: static CRTC_REG scroll_data;
337:
338: /* スクロール可能フラグ */
339: static volatile int scroll_flag;
340:
341: /* vsync カウンタ */
342: static volatile int vsync_counter;
343:
344: /* 割り込み処理ルーチン */
345: static void
346: scroll ()
347: {
348:   /* スクロールレジスタアドレス */
349:   CRTC_REG *crtc = (CRTC_REG *) 0xe80018;
350:
351: #ifdef HALF
352:   static int half_flag;
353:   if (half_flag > 0)
354:     {
355:       half_flag -= 1;
356:       goto ret;
357:     }
358:   else
359:     half_flag = HALF;
360: #endif
361:
362:   vsync_counter += 1;
363:   if (scroll_flag)
364:     {
365:       scroll_data.sc0_y_reg -= SCR_UNIT;
366:       scroll_data.sc1_y_reg -= SCR_UNIT;
367:       *crtc = scroll_data;
368:     }
```

```
369:
370: #ifdef HALF
371: ret:    ;
372: #endif
373:
374:   IRTE ();
375: }
376:
377: /* main start */
378: main()
379: {
380:   extern void scroll();
381:
382:   /* 本当は user に戻すべきだが exit すると戻るので
383:      明示して戻さない(手抜き) */
384:   B_SUPER (0);
385:
386:   /* データをロード */
387:   load_data_and_init ();
388:
389:   /* 割り込み許可 */
390:   VDISPST (scroll, 0, 1);
391:
392:   /* 最初のデータを書き込みしてスクロールを許可 */
393:   write_graph (map_data[map_line]);
394:   scroll_flag = 1;
395:
396: #ifdef OFFSET
397:   scroll_data.sc0_y_reg -= 64;
398:   scroll_data.sc1_y_reg -= 64;
399: #endif
400:   while (map_line > 0)
401:     {
402:       if (vsync_counter > (16 - 4)/ SCR_UNIT)
403:         {
404:           if (vsync_counter > (16 - 1)/ SCR_UNIT)
405:             scroll_flag = 0;
406:           map_line --;
407:           write_graph (map_data[map_line]);
408:           vsync_counter -= 16/SCR_UNIT;
409:           if (!scroll_flag)
410:             scroll_flag = 1;
411:         }
412:     }
413:   while (vsync_counter < 16/SCR_UNIT)
414:     ;
```

```
415:   VDISPST (0, 0, 0);
416:   B_CURON ();
417:   exit (0);
418: }
```

それではList 3.3を詳しくみてみましょう。先頭のマップデータの配列は，Fig. 3.7の前 (P.86) で説明したように，単なる文字列へのポインタの配列です。

```
172: /* グラフィックデータロードバッファ */
173: struct {
174:   int pal[256];
175:   unsigned char g_data[11][256];
176: } gdata;
```

これは，グラフィックデータのファイル構造そのものです。グラフィックデータファイルpattern.datは，先頭256ロングワードが256色のパレットデータで，先頭がパレット番号0番になっています。パレットに続くデータがグラフィックデータで，1タイルが16×16ドットになっています。**X68000/X68030**では，どんなグラフィックモードでもVRAMの1ワードが画面上の1ドットに対応していますが，このグラフィックデータは，256色モードではつねに0である上位8ビットを切り落としてバイトデータにしてあるので，1タイル256バイトになります。ファイル上では，それが10パターン連続して書き込まれています。

これをCの構造体にそのまま直すと，先頭がロングワードで256個ですから，

```
int pal[256];
```

です。**X68000/X68030**では，**int**はオプションで変更しなければ，32ビットロングワードですので，これでもいいのですが，他の機種で流用する場合には要注意です。とはいえ，**X68000/X68030**のグラフィックデータをそのまま使えるのは，**FM–TOWNS**ぐらいしか思いつきませんが，**FM–TOWNS**上の**GCC**や**High C**も**int**は32ビットのコンパイラですから，まぁ問題ないでしょう。

次がグラフィックデータで，バイトで256ですから，

```
g_data[256];
```

です。これが10個並んでいますが，データの大きさの都合上1個分増やす必要がありますので，11個にしてあります。

```
g_data[11][256];
```

となります。g_data[256][11]ではない点に注意してください。g_data[256][11]では，10バイトが256個並んでいる配列になります。このへんは，もう慣れるしか方法がありません。

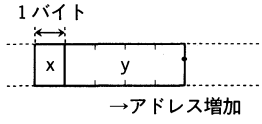
この2つがくっついて1つのグラフィックデータを表すので，全体を構造体にして

```
/* グラフィックデータロードバッファ */
struct {
  int pal[256];
  unsigned char g_data[11][256];
} gdata;
```

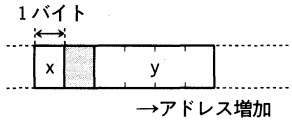
となるわけです。ここで1つ注意しておきます。構造体は，場合によっては，連続したアドレスに割り当てられないことがあります。たとえば，

```
struct {
  char x;
  int y;
} buf;
```

このような構造体では，xは1バイト，yは4バイトなので，メモリ上に



のように割り当てられるかのように思えますが，**X68000/X68030** では



のように，yとxの間に1バイトの隙間が作られます。これは，68000では奇数アドレスからワードデータやロングワードデータを読み書きできない制限があるためです。大多数のCPUでは，このように，ある場合には速度が遅くなってしまうとか，読み書きできないとかいった制限があるために，構造体を厳密に詰め合わせないで隙間を作ることが多いのです。このことをパディングといいます。

このため,「移植性」を重要視するプログラムでは，このようなファイル上のデータ構造をそのまま構造体にするような乱暴な行為はしません。ですが，ゲームプログラミングでは，移植性よりは「速度」と「簡便性」を重視しますので，本書ではあえて，速度と簡便性を重要視したプログラミングになっています。他の機種で動かすことを考えたようなプログラムでは，このような「機種依存」プログラミングは絶対に慎まないと，後で自分が痛い思いをします。

話は少しそれますが，1.5「メモリの話」(P.15) でエンディアンの話をしましたね。上の例のように，構造体をそのままファイルで読んだり書いたりした場合には，このエンディアンもモロに影響を受けます。同じ **short** の値でも，68000と8086では構造体のメモリの並びがまったく違うからです。**UNIX** の世界では，その昔，**SUN** シリーズで680?0系列のCPUがメインだったために，**UNIX** で「機種依存性がない」とされるTEXの出力など

は，ビッグエンディアンでデータが並んでいます。このため，**X68000/X68030** で TEX 関係のプログラムのデータを移植するのは，さほど苦労しません。ところが，**MS–DOS** で開発されたプログラムのデータは，これとはまったく逆で，68000 でデータを扱おうとすると大騒ぎになります。インテル VS モトローラの CPU 戦争の遺恨はこんなところにも残っているのです。

List 3.3 の説明を続けましょう。

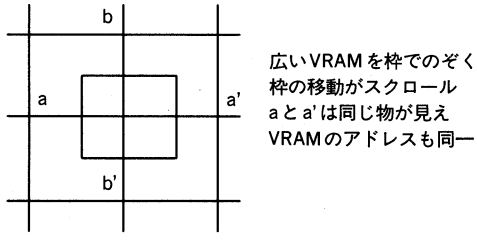
```
180: int map_line;
```

これは，現在何ブロックスクロールしたかを管理するカウンタです。**X68000/X68030** では，1 ドットごとにスクロール可能なのですが，実際に VRAM に書き込む単位は 16 ドットごとです。そこで，16 ドットスクロールするごとにこのカウンタ`map_line`を 1 ずつ増やして，次に VRAM に転送するデータを決定するのです。

次の部分の説明はちょっとややこしいです。

```
182: /* VRAM 書き込みアドレス */
183:
184: unsigned short *vram = (unsigned short *) (0xc80000 - 0x400*16);
```

Fig. 3.8 を見てください。



●Fig. 3.8　VRAM の表示概念

X68000/X68030 では，256 × 256 ドットモードでも VRAM は 512 × 512 ドットの大きさをもっています。全体で 4 画面分ある VRAM を 256 × 256 ドットの枠を通して見るようなイメージになります。ハードスクロールというのは，この見える範囲の枠を移動させることに相当します。

VRAM は 4 画面分ですので，上下左右にはみ出したらどうなるのでしょうか? はみ出した部分は，Fig. 3.8 のように，上下と左右がそれぞれつながったような感じで，常時見えるようになっています。これがいわゆる球面スクロールというものです。スクロールはこのように球面状になっていますが，だからといって VRAM が球面になっているわけではありません。256 色モードでは，512K バイトある VRAM が 2 分割されて，256K バイトで 1 画面を構成します。

最初のスクロール位置は原点ですから，1 ドット下にスクロールさせると，とたんに球面スクロールのワープ現象 (?) が起こって，4 画面ある VRAM のいちばん下が見えてきま

す。画面のいちばん上を描くのに，VRAM上ではいちばん下を書くわけです。VRAMのいちばん下は0xc80000で，1ラインあたり1024バイトですから，その16ライン上から書き始めることになります。

```
0xc80000 - 0x400 * 16
```

これが書き始めのアドレスです。このまま整数値をポインタに入れようとすると，コンパイラが「整数からポインタにしたぞ」と警告するので，「この値はポインタだよ」とコンパイラに教えてあげるのが，先頭についている(unsigned short *)です。で，最終的には

```
/* VRAM 書き込みアドレス */

unsigned short *vram = (unsigned short *) (0xc80000 - 0x400*16);
```

となるのです。

さて，いよいよ関数の解説です。まだCの実際のソースの解説を始めて間もないので，「とんでもなく詳しく」説明します。後半になるに従って細かい説明は減っていきます。途中でわからないソースに出会ったら，「前のソース」をたぶん忘れていることと思いますので，前に戻って読み直してください。最初は1語1句説明をつけます。この最初の関数は，コメントにあるように，16×16のタイルをVRAMに張り付ける関数です。

```
186: /* 16*16矩形を指定されたVRAMアドレスに書き込み */
187:
188: static                          /* 同一ファイルでしかみえない関数 */
     void                           /* 戻す値はない。手続き関数 */
     write_vram                     /* 関数の名前 */
     (                              /* 引数並び開始 */
      unsigned short *vr            /* 符号なし16ビット整数へのポインタvr */
                                    /* 実体はVRAMアドレスそのもの */
      ,                             /* 引数並びの‘,’ */
      unsigned char *dat            /* 符号なし8ビット整数へのポインタ */
                                    /* 書き込む16×16ドットタイルの
                                       先頭アドレス */
     )                              /* 引数並びの終わり */
189: {                              /* 関数の定義開始の ‘{’ */
190:   int i;                       /* ローカル変数 i の宣言 */
191:   for (i = 0; i < 16; i++)     /* iを0から1ずつ増やしながら16まで */
192:     {                          /* for文実行開始ブロックの ‘{’ */
193:       unsigned short *p = vr; /* ローカル変数pの宣言と初期化 */
194:       *p++ = *dat++;           /* *datを*pに入れてpとdatを1増やす */
195:       *p++ = *dat++;
196:       *p++ = *dat++;
```

```
197:        *p++ = *dat++;
198:        *p++ = *dat++;
199:        *p++ = *dat++;
200:        *p++ = *dat++;
201:        *p++ = *dat++;
202:        *p++ = *dat++;
203:        *p++ = *dat++;
204:        *p++ = *dat++;
205:        *p++ = *dat++;
206:        *p++ = *dat++;
207:        *p++ = *dat++;
208:        *p++ = *dat++;
209:        *p++ = *dat++;
210:        vr += 0x200;              /* 横を16ドット描いたから */
                                      /* 縦方向に1ドット動く */
211:    }                             /* for文実行ブロック終わりの '}' */
212: }                                /* 関数定義終わりの '}' */
```

1語1句説明してありますが，これはこのままちゃんとコンパイルできます。C では，意味が不明瞭になりさえしなければ，どのように記述してもコンパイラはなんら文句をいいません。

```
static void write_vram (unsigned short *vr,unsigned char *dat)
{int i; for (i=0;i<16;i++){unsigned short *p=vr;*p++=*dat++;
*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;
*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;
*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;*p++=*dat++;
vr+=0x200;}}
```

このように全体を記述しても文句はいいません。コンパイラにとってみれば，これは正しいソースなのですから。だからといって，このような読みにくい書き方をしたり，解説のような冗長すぎるコメントも考え物です。このようなプログラムの基本的な記述方法については,「プログラム書法」(共立出版) を読んでください。本書を終わりまで読めた方なら，十分理解できる書物です。

この関数では，新しく++演算子が登場しています。これ以外のコメント部分が理解できなかったら，前に戻って再度読んでください。それでは++演算子の説明です。この++には，対応して--があります。++はその見た目が示すように，++が施された式を1だけ増やします。--の場合には1減らす働きをします。また，

```
op0++;
++op0;
```

のように，後ろに置く場合と前に置く場合とがあります。置く位置によって，その式の評価の方法が変わります。後ろに置いた場合は，まずop0を評価します。その後，op0を１増やします(--の場合には「増やします」が「減らします」になります)。前に置いた場合には，まずop0を増やして，その後op0を評価します。ややこしいですか?

```
a = b++;
```

この場合には，まずaにbが代入され，その後bが１増やされます。式の値自体は，a = bですから，増やされる前のbの値です。

```
a = ++b;
```

この場合には，まずbが１増やされます。その後a = bが評価されます。ですから，式の値はbに１を加えた値になります。ちょっとパズル的ですが，以下のプログラムを見てください。

```
int
foo0(int a, int b)
{
  if (a = b++)
    return b;
  else
    return a;
}
```

この場合は，bの値によって返す値が違います。

```
int
foo1(int a, int b)
{
  if (a = ++b)
    return b;
  else
    return a;
}
```

この場合は，どちらでも同じ値が返ります。**GCC** は賢いコンパイラなので，この両者の違いをちゃんと見抜いて

```
_foo0:
        move.l 8(sp),d0
        move.l d0,d1
        addq.l #1,d1
```

```
        tst.l d0
        beq ?2
        move.l d1,d0
?2:
        rts

_foo1:
        move.l 8(sp),d0
        addq.l #1,d0
        rts
```

のようなコードを生成します。簡単なアセンブラコードですから，説明の必要はないでしょう(難しかったら，パス! してけっこうです)。

この演算子++がポインタと組み合わさったのが，

```
*p++ = *dat++;
```

です。*→ p ← ++で，どちらの演算子も対象はpである点に注意してください。コンパイラは，まず

```
*p = *dat
```

を評価します。その後，p++とdat++を評価します。よくまちがうのが，「ポインタ自体を増やすのか，ポインタが指しているデータを増やすのか?」です。

```
int *p = malloc (sizeof (int));
*p = 0;
/* pの指すメモリを増やしたい */
*p++;
```

これは誤りです。この場合には，ポインタp自体が増やされます。希望する動作を望む場合は，

```
int *p = malloc (sizeof (int));
*p = 0;
/* pの指すメモリを増やす */
(*p)++;
```

と記述します。これは後ろ側に++を置くので，わりあい発見できるのですが，

```
int *p = malloc (sizeof (int));
*p = 0;
...
...
```

```
...
/* OK increment it!!! */
*++p;
```

などと記述されていると，コメントとあいまって見事にだまされてしまいます。Cでは，アセンブラを知らない人間にとって，不自然な記述がときどき出てきます。これはもう慣れるしかないのです。アセンブラで，オートインクリメントやオートデクリメントのアドレッシングモードを使っていれば，++演算子や--演算子は非常に自然な記述で理解は容易なのですが，使った経験がないと，理解は困難でしょう。申し訳ないですが慣れてください。

また，ポインタに対する++は，ポインタを整数値としてみた場合，単に数値が1増えるのではなく，ポインタが指しているデータ1要素分だけ，値自体が増えます。

```
*p++ = *dat++;
```

では，pは **short** *ですから，*pのサイズは2バイトで，pの値は2増えます。datは **unsigned char** *ですから，*datのサイズは1バイトで，datの値は1増えます。ポインタ⟷整数の変換を頻繁に行う危ないプログラム (ゲームでは，ありがちなケースです) では，十分に注意してください。

List 3.3の説明に戻りましょう。関数static void clr_vram()はもう簡単にわかりますね。これはただ単にVRAMを透明データ0で埋めているだけです。

次の関数を完全に理解できたら，Cについてはかなり理解が深まったと思ってまちがいないでしょう。これはポインタを駆使した関数の例題です。ここで行っていることは，マップデータの配列文字列に従って，タイルを選択し，それをVRAMに張り付ける作業です。

```
/* マップデータに基づいてグラフィックVRAMにデータを書き込む */
static
void write_graph (
char *dat                          /* 引数dat は文字列を指している */
/*
    datの示すアドレス
       ↓
      +-+-+-+-+-+-+-+-+....+-+
      | |1|1|2|0| | | |....|e|     e は数値の 0
      +-+-+-+-+-+-+-+-+....+-+
*/
)
{
  /* 現在の書き始めVRAMアドレスを得ます */
  unsigned short *vr = vram;

  /* 文字列が続いている間実行 */
```

```
  while (*dat)
    {
      /* スペースコードは透明データなので */
      if (*dat == ' ')
      /* VRAMを透明タイルで埋める */
        clr_vram (vr);
      else
        {
          /* 文字コードから整数に変換する */
          int i = *dat - '0';
          /* タイルを指定してデータをVRAMに書く */
          write_vram (vr, gdata.g_data[i]);
        }
      /* 次の文字を指すようにポインタを増やす */
      dat++;
      /* 横に16ドット進む */
      vr += 16;
    }
  /* 縦に画面上方に16ドット進む */
  vram -= 0x200 * 16;
  /* 上にはみ出した? */
  if (vram < (unsigned short *) 0xc00000)
  /* はみ出たのでVRAMアドレスを補正する */
    vram =  (unsigned short *) (0xc80000 - 0x400*16);
}
```

詳しくコメントを入れましたが，これでも具体的なイメージがわいてこない場合もあるでしょう。いちばんわかりにくいのが **while** で行っている部分でしょう。

```
while (*dat)
```

これは，datが指しているアドレスのメモリが数値 0 でない間は，次の{}で囲まれた範囲を実行しなさいという意味になります。マップデータ文字列は

```
"␣␣␣␣␣␣67␣␣␣␣67␣␣"
```

のように，16 の文字列になっています。C では，文字列の終わりに必ず0x0がくっついています。これはバイトの値で 0 です。文字の '0' とは別です。**while** の処理{}の中でdatを 1 ずつ進めていけば，やがてこの値 0 の位置にきます。そのときは，この **while** の条件が成立しなくなって，次の文へ処理が移るのです。

```
/* スペースコードは透明データなので */
if (*dat == ' ')
```

```
/* VRAMを透明タイルで埋める */
  clr_vram (vr);
```

ここで新しい表記が出ています。'(シングルコーテーション) で囲まれた文字は，その文字のキャラクタコードを表す整数に変換されます。たとえば，'a'は97(10進数) です。このif (*dat == ' ')というのは，*datがスペースコードだったらという意味になります。スペースはVRAMを透明データで埋める指示なので，そのようにします。

次の **else** の処理

```
{
  /* 文字コードから整数に変換する */
  int i = *dat - '0';
```

は，文字コードをその文字が表す数値に変換しています。C に慣れていないと，ここでついatoi()とかsscanf()とかの変換関数を使いたくなります。ですが，このマップデータの場合には，文字の '0' から '9' までしか現れないので，文字コードが0123···9の順番で並んでいることを利用して，単純に文字コード '0' を表す数値を引き算することで，文字→数値の変換をすませています。これは頻繁に使われるテクニックの1つです。覚えておいて損はないでしょう。

次に，

```
  /* タイルを指定してデータをVRAMに書く */
  write_vram (vr, gdata.g_data[i]);
}
```

は，文字から数値への変換が終わったので，構造体gdataから該当データを選択します。これは簡単で，変換された数値を[]で囲んで

```
gdata.g_data[i]
```

とすればOKです。これをVRAMに転送するわけですから，write_vram()を呼び出してやればいいのです。このgdata.g_data[i]は **unsigned char** の配列ですから，関数に渡す式では **unsigned char** *に変換されます。VRAMに転送が終わったら，次の文字を得るために

```
/* 次の文字を指すようにポインタを増やす */
dat++;
```

を実行します。同時に，VRAMに書き込む位置が16ドット移動しますから，

```
  /* 横に16ドット進む */
  vr += 16;
}
```

を実行したら，while (*dat)に戻ります。**while**ループを抜け出したら，16タイルぶんの転送を終わっていますから，縦方向に16ドットぶん移動して，もしVRAM範囲をはみ出したら，VRAMアドレスを正しい位置に補正してやります。どうですか? わかるでしょうか?? いっぺんにいろいろなことが出てきたので，消化不良になるかもしれませんね。ここいらで一服して，ゲームでも動かして息を抜いてください(^_^)。

それでは，続きを始めましょう。関数init_map_line()です。ここまでの話を完全に理解したなら，もう何も説明することはありません。マップ文字列のいちばん最後が'*'で終わっていることを知っていれば，解読は簡単でしょう。

次は，init_palet()です。ここでは，新しく&演算子が登場しています。実は，この例では&演算子を使う必要はないのですが，勉強の意味で使ってみました。&演算子は，ある変数のアドレスを取り出す演算子です。int a;が宣言されていれば，&aはaのアドレスになり，その型は**int** *です。

init_palet()では，

```
291:    int *dat = (int *)&gdata;
```

と，&演算子が使われています。これは

```
int *dat = gdata.pal;
```

と実は同じなのです。pal[0]は，構造体gdataの先頭にある**int**のデータです。先頭ということは，構造体gdataのアドレスがこれに一致します。配列が[]なしで現れたときには，その配列の先頭を示すポインタに変換されます。ですから，

```
int *dat = gdata.pal;
```

は

```
int *dat = &gdata.pal[0];
```

とも同等なのです。これがわかってしまえば，この関数も他には新しい事項は出てきません。次の関数に進みましょう。

関数load_data_and_init()です。ついにファイルストリームの登場です。これまた本書の特徴なのですが，普通の入門書では，一通りの説明が終わったら，まずこのファイルストリームの話が出てきます。ファイル入出力関係も，初心者がつまずく候補の1つです。まず，突然現れる

```
300:    FILE *fp = fopen ("pattern.dat","rb");
```

に驚きます。「なんじゃこのFILE *って??」。FILE *というのは，stdio.hで定義されているファイルストリーム構造体へのポインタです。2.2「関数」(P.29)でも書きましたが，このようなファイル入出力のような「高度な処理」は，Cには最初から備わっていません。

stdio.h とかライブラリとかの助けを借りて，初めて処理できるのです。

一般にファイルの入出力は，その機械や OS の違いにより，異なった方法で行われます。ところが，C コンパイラは，その異なった機械や OS の上で動きます。機械ごとに異なったファイルの入出力処理をいちいち記述していたのでは，プログラマがいくらいても足りなくなります。そこで，ファイルストリーム構造体という，OS や機械の違いを吸収するための仕掛けを作っておいて，この構造体とライブラリ関数を通して行う処理は，(完全ではないが) 同じ記述でまかなえるように工夫してあるのです。

普通，`printf()`とか`fprintf()`といった関数は，必ずといっていいほど使われますが，これすら C に最初から備わった機能ではありません。`#include <stdio.h>`はオマジナイのごとく，必ず C のソースには付随してくるのです。このソースでも，しっかり`#include <stdio.h>`を使っています。ですから，`FILE *`とか`printf()`のようなデータ形式や関数が使えます。と，ここまで書いて気づいたのですが，このプログラムの頭の説明ですでに「ファイル」という言葉が出てきており，しかもその内容まで説明してしまってますね。ここではもう一度改めてファイルについて説明します。

ファイルとは何でしょうか? 通常，ファイルとはディスクに記録されたデータを指していることが多いのです。よく **dir** とタイプしてダラダラと表示される，アレがファイルです。ファイルには，さまざまなデータが記録されていますが，実は，プログラム自身もファイルに他なりません。**GCC** もディスク上でファイルとして保存されています。C ではキーボードも，ディスプレイもファイルとして扱うことができます。これらは，とてもファイルとは似ても似つかない構造をしていますが，プログラムからみたらファイルとして扱うこともできるのです。ですが，やはりこれらも機械や OS に依存しているので，一律には扱えません。そこで，C では最初から 3 つのファイルが使えるようになっており，基本的な入出力はそこを通して行うようになっています。これらの 3 つのファイルもやはり，`FILE *`として最初から用意されています。

これも，`#include <stdio.h>`のオマジナイで使えるようになります。

stdin ： 標準入力（普通はキーボード）
stdout： 標準出力（普通はディスプレイ）
stderr： 標準エラー出力（普通はディスプレイ）

GCC のコンパイラ本体は，入出力の指定がされていない場合は標準入力からプログラムを読んで，コンパイル結果を標準出力に，エラーメッセージを標準エラー出力に書き出します。**UNIX** から移植されたプログラム類は，たいていこういった仕様になっています。なぜこうなっているのか，その理由は，COMMAND.X によるリダイレクトと密接に関係しています。この 3 つの入出力は，COMMAND.X によって別のファイルに自由に割り当てることができるのです。通常は標準入力はキーボードですが，COMMAND.X のリダイレクト機能を使えば，ファイルをキーボードの代わりに割り当てることができるのです。この場合，プログラムは，あたかもキーボードがタイプされたかのように，そのファ

イルから文字やデータを読み取ることができるのです。残念なことに，**Human68k** ではCOMMAND.Xが標準エラー出力をリダイレクトできません。そのために，一部の(**GCC**を含めて) **UNIX** 上で動いていたプログラムは，使いづらい場合があります。

ところで，この標準入出力のように，最初から用意されていないファイルを扱うことができるようにすることを，「ファイルをオープンする」といいます。逆にいえば，3つの標準入出力は，最初からファイルがオープンされているわけです。

ファイルをオープンする関数がfopen()です。この関数には2つの引数があり，1つはファイルネームを示す文字列へのポインタ，もう1つはファイルオープンモードを示す文字列へのポインタです。ファイルのオープンモードにはいくつか種類がありますが，ゲームで普通行うのは「読み取りモード」と「書き込みモード」しかないでしょう。**Human68k** は **MS–DOS** そっくりの仕様のため，ファイルにテキストモードとバイナリモードという忌まわしいモードも指定する必要があります。**UNIX** や **OS–9** といった「まっとうなOS」では，このようなテキスト，バイナリといった区別は存在していません。まぁ，これは愚痴ってもしょうがないので，**type FILE** しても読めないファイルはバイナリだと決めつけても，大きなまちがいにはならないでしょう。

ファイルの説明が終わったところで，実際の関数load_data_and_init()の説明です。

```
300:   FILE *fp = fopen ("pattern.dat","rb");
```

これが実際にファイルをオープンしている部分です。ファイルネームはpattern.dat，オープンモードが"rb"です。'r' はread，'b' はbinaryですから，「読み出しバイナリモード」でファイルをオープンしています。関数fopen()は，ファイルが見つかったら，ファイル構造体へのポインタを返し，なんらかの原因でファイルがオープンできなかったら，数値0を返します。ポインタの説明のところで述べたように，数値0は「指す物がない」というポインタの特殊な値です。stdio.hには，この「指す物がない」ポインタの値を，NULLとプリプロセッサにマクロ定義させています。そこで，次の処理で

```
301:   if (fp == NULL)
302:     {
303:       printf ("File can't open\n");
304:       exit (1);
305:     }
```

のように，ファイルがオープンできなかったらメッセージを出して，即座にexit()します。関数exit()は，名前のとおりプログラムを終了させる関数です。呼ばれたら最後，戻ってはきません。ファイルが無事オープンできたら，実際に読み取りを行います。

```
307:   if (fread ((void *)&gdata, sizeof (char), sizeof (gdata), fp)
           != sizeof (gdata))
308:     {
309:       printf ("File read error\n");
```

```
310:        fclose (fp);
311:        exit (1);
312:      }
```

関数fread()は，オープンされたファイルから読み取ったデータを置くためのメモリアドレスを指定して(これを普通バッファと呼びます)，ファイルをリードする関数です。fread()がどのような引数をもっていて，どんな結果を返すのかなどについては「ライブラリマニュアル」に明記してあります。今までは，できる範囲で1つ1つ説明してきましたが，ここからは自分でマニュアルを参照してみてください。いちいち全部説明していると，勉強にならないだけでなく，本書がライブラリの説明書になってしまいますので(^_^;;;;)。はしょって説明すれば，ファイルを実際にリードしてみて，中身はともかく既定のサイズを読めない場合には，メッセージを出してexit()しています。

```
310:        fclose (fp);
```

これはオープンされているファイルをクローズします。開けたら閉めるのが常識です。ライブラリは馬鹿ではないので，閉め忘れてexit()した場合や，閉め忘れたままmain()を終えた場合でも，自分がオープンしたファイルを閉めてくれますが，これはライブラリが気をきかせているだけで，保証されているわけではありません。

```
314:    CRTMOD (10);
315:    G_CLR_ON ();
316:    B_CUROFF ();
```

この3つの処理は画面関係の初期化です。G_CLR_ON()で，もしもグラフィックにRAM-DISKを作っていた場合には内容が破壊されます。本来は，グラフィックが使えるかどうかをきちんとチェックすべきところです。

```
317:    init_palet ();
318:    init_map_line ();
```

この2つはもう説明の必要はないでしょう。自前の関数ですし，すでに説明は終わっています。これで，下請けの関数群は終わりです。いよいよスクロールを行う部分の処理です。

```
322: /* スクロールレジスタと同等の構造体 */
323:
324: typedef struct {
325:    short sc0_x_reg;
326:    short sc0_y_reg;
327:    short sc1_x_reg;
328:    short sc1_y_reg;
329:    short sc2_x_reg;
330:    short sc2_y_reg;
```

```
331:    short sc3_x_reg;
332:    short sc3_y_reg;
333: } CRTC_REG;
```

この宣言は，ハードウェアのスクロールレジスタとまったく同じ構造をもった構造体を作成しています。ハードウェアのスクロールレジスタは，WRITE ONLY なので，書き込んだデータは消失します。これでは管理ができないので，ハードウェアに書き込む値を管理する変数が必要です。それが，次のscroll_data変数です。

```
335: /* スクロール用データ */
336: static CRTC_REG scroll_data;
```

この変数の値を操作し，ハードウェアのスクロールレジスタに転送することで，実際の処理を行っています。

次のscroll_flag変数は，スクロールさせる関数にグラフィックの準備ができたことを通知します。

```
338: /* スクロール可能フラグ */
339: static volatile int scroll_flag;
```

スクロールを処理する関数は，割り込み処理になっています。割り込み処理は，ほうっておくと，グラフィックの書き込みが終わっていなくても，時間がくれば勝手にスクロール処理をしてしまいます。これでは困るので,「できたよ」「ほな，やりましょ」と打ち合わせして処理を進めなければいけません。scroll_flag変数は，そのネゴをするための変数です。

次の，

```
341: /* vsync カウンタ */
342: static volatile int vsync_counter;
```

は，割り込み処理が今まで何度通過したかをカウントするカウンタです。後で説明しますが，垂直帰線期間ごとに１ドットスクロールさせると，けっこう速いスクロールになります。そこで，スクロールをトグルにして「する」「しない」を交互に繰り返すことで，見た目のスクロール量を半分にします。このためのカウンタです。

関数scroll()は，スクロールを行う関数の本体です。この関数は，明示的な呼び出しがどこにもありません。リストのどこを見てもscroll();といった呼び出しはありませんが，この関数はちゃんと呼ばれます。これを呼び出すのは，ハードウェアから通知される垂直帰線期間を示す信号です。CPU はこの信号を受け取ると，あらかじめ登録してあるこのscroll()を自動的に呼び出します。そのときに CPU がどんな処理をしているかは特定できません。ですから，前に説明したように，本体の処理ときちんと同期をとるような変数が必要になるのです。ここで，少し前に戻ります。

```
338: /* スクロール可能フラグ */
339: static volatile int scroll_flag;
340:
341: /* vsync カウンタ */
342: static volatile int vsync_counter;
```

この2つの変数には**volatile**という見慣れない言葉がくっついています。この**volatile**は,「予期しない値の変動」が起こる可能性のある変数につけます。一般に，コンパイラは，割り込み処理など，本来のプログラムの流れからは予期できない方法によって変数の値が変更されないものとして，最適化を行います。たとえば，変数scroll_flagは，本来の処理で1を入れておけば，自分で変更をしない限りは1であるという前提です。ですが，実際にはこの変数は，割り込み処理により予期しない変更を受ける可能性があります。こういった変数には**volatile**という言葉をつけて,「予期しない変更がある変数だよ」とコンパイラに通知しておきます。こうしておけば，コンパイラは，この変数に予期しない変更があることを知って，変更がありえないといった仮定での最適化を行わなくなります。

XCのように，ほとんど最適化をしないコンパイラではこういった配慮は無用ですが，**GCC**ではこのような配慮をしておかないと簡単に無限ループに陥ります。例をあげれば,

```
scroll_flag = 1;
while (scroll_flag)
  ;
```

のような処理がその典型です。scroll_flagに1を入れていますから，次の**while**は黙って無限ループをするコードに最適化します。scroll_flagが割り込み処理で突然0に変わるかもしれないことを，コンパイラは知らないからです。こういった変数には必ず**volatile**が必要です。

それでは，実際の割り込み処理をみてみましょう。新しいプリプロセッサの制御が出てきています。まず,

```
351: #ifdef HALF
     ..
     ..
360: #endif
```

です。これはHALFが**#define**されていれば，#ifdef HALF～#endifまでに記述してある部分をそのまま残してコンパイルします。もしHALFが**#define**されていなければ，#ifdef HALF～#endifまでに記述してある部分をバッサリ削除してコンパイルします。これは条件コンパイルと呼ばれるもので，たとえばゲーム作成中は，当たり判定をなくしてデバッグしておき，ほぼ完成したら当たり判定を復活して，さらにデバッグするとかいった用途に使います。**GCC**では，コマンドラインから

```
A:>gcc map.c -DHALF -O -liocs
```

のように，**-D** オプションを使って，HALFを**#define** したのと同等の処理を行わせることができます。これは，機種間格差を吸収するのにも便利です。この List 3.3 では，スクロール量をコントロールするのにこの**#ifdef** 命令を使っています。

次の部分がスクロール量を調節している部分です。

```
351: #ifdef HALF
352:   static int half_flag;
353:   if (half_flag > 0)
354:     {
355:       half_flag -= 1;
356:       goto ret;
357:     }
358:   else
359:     half_flag = HALF;
360: #endif
```

ここで，新しい演算子-=が登場しています。対応する演算子として '+=', '*=', '/=' があります。これらはそれぞれ，

```
op0 += op1; → op0 = op0 + op1; (加法)
op0 -= op1; → op0 = op0 - op1; (減法)
op0 *= op1; → op0 = op0 * op1; (乗法)
op0 /= op1; → op0 = op0 / op1; (除法)
```

に対応しています。C には，なるべくタイピングの量を減らせるように，このような省略した記述方法がたくさんあります。

goto は 2.2「関数」(P.29) に出てきましたね。goto *Label*;で，*Label*が存在する位置(同じ関数内でないとダメです) にプログラムの制御が移動します。

```
362:   vsync_counter += 1;
363:   if (scroll_flag)
364:     {
365:       scroll_data.sc0_y_reg -= SCR_UNIT;
366:       scroll_data.sc1_y_reg -= SCR_UNIT;
367:       *crtc = scroll_data;
368:     }
369:
370: #ifdef HALF
371: ret:    ;
372: #endif
```

ここまではもう問題ないでしょう。すべて，これまでに説明した記述方法です。最後の

```
374:    IRTE ();
```

は，このマクロが記述されている関数が割り込み処理関数であることをコンパイラに通知しています。割り込み処理関数については，「X68k Programming Series #1 Develop.」および「X68k Programming Series #2 libc」を参照してください。割り込み処理関数の動作を制御するマクロ群の説明は，おそらく本書で説明するレベルを超えてしまっていると思いますので。

残りは関数main()です。これがいちばん最初に実行される関数です。一般的に，大きなプログラムになるに従って，main()は逆に小さくなる傾向になっています。main()には必要な下準備だけして，後はプログラムの本体関数を順番に呼び出すだけというのが，大規模なプログラムのやり方です。このプログラムはもちろん大規模ではないですが，main()自体は小さい処理になっています。では，main()を説明しましょう。

```
382:    /* 本当は user に戻すべきだが exit すると戻るので
383:       明示して戻さない(手抜き) */
384:    B_SUPER (0);
```

この部分では，プログラムの実行状態をユーザモードからスーパーバイザモードに切り替えています。VRAMやスクロールレジスタなどは，ユーザモードでは保護されていて，操作しようとしてもバスエラーになってしまいます。「不便だなぁ」と思われるでしょうが，この保護機能のお蔭で安心してプログラムが組めるのが，**X68000/X68030**の長所でもあるのです。そう簡単には暴走しないし，某国民機種のように，プログラムのバグによって，親切にも自発的にリセットする(爆笑)ようなことは，**X68000/X68030**ではまず起きません。

ですが，このプログラムではユーザが直接VRAMを操作しますので，バスエラーにならないように保護機能を外します。そのペナルティとして，「何が起こっても知らないよ」の状態になるのですが，安心してください。**X68000/X68030**では，スーパーバイザモードで暴走しても，滅多にディスク破壊などの致命的事態にはなりません。そうなる前に，「バスエラー」か「アドレスエラー」，「おかしな命令」などの保護にひっかかって停止します。その後，リセットを必要とする場合が多いのですが，これはスーパーバイザで動かしたペナルティだと思ってあきらめてください。

関数load_data_and_init()の呼び出しについては，説明は必要ないでしょう。

次に，

```
389:    /* 割り込み許可 */
390:    VDISPST (scroll, 0, 1);
```

です。これが関数scroll()の，割り込みによる呼び出しを設定している部分です。関数名称が'()'なしで引数に現れています。この場合，コンパイラはその関数へのポインタを生成して引数にします。ここで新しい概念である**関数へのポインタ**が出てきました。ポイン

タはメモリのアドレスの値そのものです。関数は，コンパイルされると，機械語になってメモリに置かれます。メモリに置かれるということは，そのメモリのアドレスが存在しますね。このアドレスを値にするのが，関数へのポインタです。関数VDISPST()の１つ目の引数には，垂直帰線期間に入ったら呼び出される，割り込み処理のアドレスを渡します。「割り込み処理」のアドレスであって，関数のアドレスではない点に注意してください。

関数scroll()は，**X68000/X68030 版 GCC** の拡張機能によって，**割り込み処理**としてコンパイルされています。通常の関数を関数VDISPST()の１つ目の引数として渡すと，100%確実に暴走します。**割り込み処理**は通常の関数とは違った形式のプログラムなのです。垂直帰線期間ごとに'Hello!'と表示させることなどを，マニュアルを読んで試みられる方もおられると思うので，注意点を書いておきました。話題が少しそれました。

関数へのポインタは，他にはどんな用途に使われるのでしょうか?? ゲームのプログラムで考えられるのが特殊処理です。たとえば，コナミのパロディウスでは，自機が「緑ベル」を拾うと大きくなって，無敵の状態になりますよね。これを実現する方法として，関数へのポインタを使う方法があります。自機の処理の呼び出しに関数へのポインタを使います。「自機の処理を行う」関数へのポインタには，通常,「通常処理」を行う関数のアドレスが登録してありますが，緑ベルを拾った瞬間に,「拡大無敵」の処理を行う関数のアドレスに登録し直します。これで「拡大無敵」になります。一定時間が経過したら，この「拡大無敵処理」の関数によって「自機の処理を行う」関数へのポインタを「通常処理」が指すように変更すれば，また元の状態に戻るわけです。このように，関数へのポインタを使えば，自己書き換えなどという姑息な手段を使わなくても，特殊な処理を行うことができます。本書では,「関数へのポインタ」は直接扱っていませんが，使えるような用意はしてあるプログラムを扱っています。挑戦してみてはいかがでしょうか? 関数main()では，他には特に変わったことはしていません。今までの知識で十分解読できるでしょう。

3.4.3 ■ 垂直帰線期間を無視してみる

3.4.2「マップによるグラフィックのスクロール」(P.86) で扱ったスクロールプログラムは，ゲームの常識に従い，垂直帰線期間でしかスクロールを行っていません。ですから,「書き換えが見える」ようなみっともないスクロールはやりません。が,「本当にそうなの??」といった疑問もわいてくるでしょう。そこで，3.4.2 で扱ったプログラムを，垂直帰線期間をまったく無視するように書き換えてみましょう。

やり方は簡単です。今まで割り込みが自動的に呼び出していた関数scroll()を，main()から自前で呼び出すように変更すればいいのです。

List 3.4

```
1: /* スクロールレジスタと同等の構造体 */
2:
3: typedef struct {
4:   short sc0_x_reg;
```

```
 5:   short sc0_y_reg;
 6:   short sc1_x_reg;
 7:   short sc1_y_reg;
 8:   short sc2_x_reg;
 9:   short sc2_y_reg;
10:   short sc3_x_reg;
11:   short sc3_y_reg;
12: } CRTC_REG;
13:
14: /* スクロール用データ */
15: static CRTC_REG scroll_data;
16:
17: /* スクロール可能フラグ */
18: static volatile int scroll_flag;
19:
20: /* vsync カウンタ */
21: static volatile int vsync_counter;
22:
23: /* 割り込み処理ルーチン */
24: static void
25: scroll ()
26: {
27:   /* スクロールレジスタアドレス */
28:   CRTC_REG *crtc = (CRTC_REG *) 0xe80018;
29:
30: #ifdef HALF
31:   static int half_flag;
32:   if (half_flag > 0)
33:     {
34:       half_flag -= 1;
35:       goto ret;
36:     }
37:   else
38:     half_flag = HALF;
39: #endif
40:
41:   vsync_counter += 1;
42:   if (scroll_flag)
43:     {
44:       scroll_data.sc0_y_reg -= SCR_UNIT;
45:       scroll_data.sc1_y_reg -= SCR_UNIT;
46:       *crtc = scroll_data;
47:     }
48:
49: #ifdef HALF
50: ret:    ;
```

```
#endif

}

/* main start */
main()
{
  extern void scroll();

  /* 本当は user に戻すべきだが exit すると戻るので
     明示して戻さない(手抜き) */
  B_SUPER (0);

  /* データをロード */
  load_data_and_init ();

  /* 最初のデータを書き込みしてスクロールを許可 */
  write_graph (map_data[map_line]);
  scroll_flag = 1;

#ifdef OFFSET
  scroll_data.sc0_y_reg -= 64;
  scroll_data.sc1_y_reg -= 64;
#endif
  while (map_line > 0)
    {
      int wait;
      wait = ONTIME();
      if (vsync_counter > (16 - 4)/ SCR_UNIT)
        {
          if (vsync_counter > (16 - 1)/ SCR_UNIT)
            scroll_flag = 0;
          map_line --;
          write_graph (map_data[map_line]);
          vsync_counter -= 16/SCR_UNIT;
          if (!scroll_flag)
            scroll_flag = 1;
        }
      scroll ();
      while (ONTIME() - wait < 3)
        ;
    }
  B_CURON ();
  exit (0);
}
```

ただ関数scroll()を呼び出すだけの変更では，全体の処理が速すぎて全然スクロールに見えません。かといって，ただ単に空のループで処理時間を遅延させるのでは，機種間での処理速度の違いが顕著に現れてしまいます。なんらかの方法によって，プログラムの進行を一定時間で行うように工夫する必要があります。そこで，**X68000/X68030** では，電源スイッチを投入してからの時間を絶えず計測していることを利用して，この時間をみながら一定の処理速度で動くように作成します。この時間計測も，実は割り込みで行われていますが，垂直帰線期間とは同期していませんので，スクロールがおかしくなるはずです。

実際に実行させてみるとわかりますが，それほど激しく画面が乱れるわけではありません。なぜなら，グラフィック画面のスクロールレジスタの垂直方向の値は，垂直帰線期間にCRTCから読み取られて，内部で有効になるからです。ですから，このサンプルのような縦スクロールでは，それほど顕著に乱れは発生しません。それでも，前節で説明した割り込みを用いたスクロールに比べれば，「なにかしら変」といった感じになっています。このサンプルでは，時間の測定を比較的厳密に行っていますから，機種ごとの処理速度の違いは吸収されていますが，もっと複雑な処理がたくさん加わってくると，処理速度の違いを吸収するのはたいへんな作業になってきます。

結果的にそれほどよいサンプルにはなりませんでしたが，とにかく表示期間中に表示物を操作するのは御法度であることを忘れないでください。

■ 3.5 スプライトを使ってみましょう

X68000/X68030 の最大のゲーム機能であるスプライトを，実際にCで使ってみましょう。その前に一度，スプライトとはナンゾヤ? から始めます。スプライトは「アニメーション」のセル原画によくたとえられますが，OHPのフィルムにたとえたほうがより実際に近いと思われます。OHPは会議のプレゼンテーションなどによく用いられますが，スクリーンはパソコンのCRTで，何枚かに重ねられるフィルムはスプライトにあたります。スクリーンに投影された絵や文字を移動するには，絵そのものを動かさなくても，フィルムをずらすことで簡単に実現できます。絵そのものをずらす方法は，スプライト機能をもたないパソコンでゲームを作成する場合に用いられる方法に相当します。当然，「書き換え」を行うのですから，処理時間もかかるし，重ね合わせなどを考慮するとたいへんな処理になります。一方，スプライト機能があれば，OHPの場合と同じく，フィルムに相当するスプライトを移動させるだけでよく，背景との合成はハードウェアが解決してくれます。このように，スプライトは多数のキャラクタを同時に動かすのに，たいへん便利な機能です。ハードウェア的にみると，OHPで重ね合わせするフィルムの最大枚数が最大表示キャラクタ数で，フィルムに描ける絵の種類の最大数(これはOHPでは無限ですね。もう1つ注意すると，スプライトではフィルムには1つしか絵を描くことができません)が最大定義数になります。また，フィルム(スプライト)に描くことのできる絵の大きさにも制限があります。ハードウェアの構成によっては，横方向に並べることのできる絵の数にも制限があ

ります。

●Table 3.1　スプライト機能のスペック（スプライト機能をもつパソコン，家庭用ゲームマシンのスペック）

	X68000/X68030	メガドライブ	PC-ENGINE	SUPER-FAMICOM	FM-TOWNS
最大表示個数	128	80	64	128	1024
最大定義個数	256	2048	??	??	896
横方向制限	32	20	8	32	なし
ドット構成	16 × 16	8 × 8 − 32 × 32	64 × 32(Max)	64 × 64(Max)	16 × 16
表示色 (MAX)	16/65536	64/512	16/512	256/32768(??)	32768
パレット	16	−	32(??)	??	
BG 機能	256 × 256, 2 面 512 × 512, 1 面	320 × 224, 2 面	256 × 216, 1 面	256 × 224, 4 面	なし

＊ 上記のスペックはあくまでも参考である。実際にプログラムする場合には，いろいろな制限があることが多いのが現実である。X68000/X68030 以外は筆者が雑誌などで調べたものである。

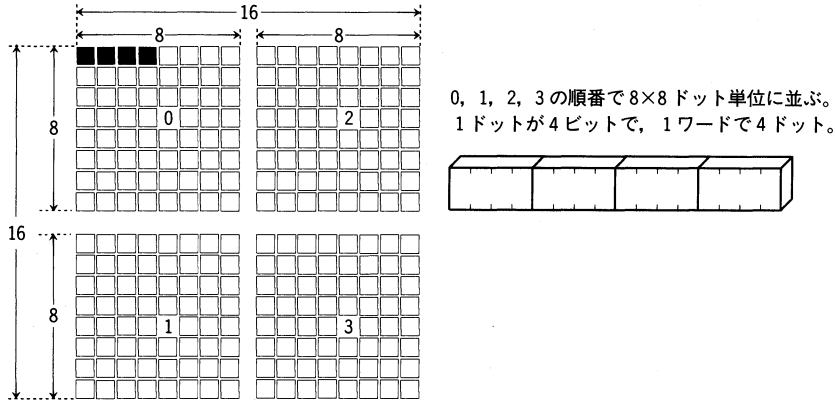
Table 3.1 がそのスペックの一覧です。性能と値段が一致してない! などと思ってはいけません (笑)。**FM-TOWNS** が一見して断トツの性能に見えますが，現実はそう甘くはありません。現在多数派を占める，多重スクロールタイプのゲームを移植したり作成したりする場合には，スプライトの使える画面モードが非常に制限されている **FM-TOWNS** ではたいへんです。イメージファイトの 4 面がよい例で，**X68000** 版では「楽々」処理している「シールド編目」の処理が，80386 パワーでも少々苦しいでしょう。かといって，**X68000/X68030** にギャラクシーフォースみたいなゲームが移植できるかといえば，そう簡単ではありません。横方向制限が **FM-TOWNS** にないのは，スプライト自体の方式がまったく異なっているためで，**FM-TOWNS** では VRAM を半分このスプライト画面に割り当てて，キャラクタデータをハードウェアで高速転送しています。このために，スプライト画面以外の背景画面が 1 面しかもてないようになっています。とにかく，カタログ上では表面に出てこない性能の差がたくさんありますので，字面だけで判断するのは危険です。ちなみに，ゲームセンターに置いてある一部の機械のスペックを Table 3.1 のものと比べると，まさに「驚愕」のスペックになるでしょう。**X68000/X68030** のスプライト表示能力は，家庭用としてみればまだ第 1 級の実力があります。今のところ，スプライトを使ったゲームをセルフ開発できるのは，**X68000/X68030** と **FM-TOWNS** だけということになります。**MSX** でもできますが，いささか開発環境に苦しいものがあるでしょう。

X68000/X68030 のスプライト機能を制御するハードウェアは，一部の機能を除いて $EB0000～$EBFFFF に配置されています。機能的には，前のたとえで述べたフィルムのずらす位置を指定するスプライトスクロールレジスタと，フィルムに描く絵を定義するためのプログラマブルキャラクタジェネレータの 2 種類のハードウェアに分類できます。

3.5.1 ■ プログラマブルキャラクタジェネレータ

まず，プログラマブルキャラクタジェネレータから説明しましょう。普通は，こんなに長い名前で呼ばずに PCG と省略しますので，これからは PCG と書くことにします。PCG は，ス

プライト画面に描くためのキャラクタを登録しておくハードウェアです。**X68000/X68030**では，16×16ドットのキャラクタを最大256パターン登録できます。描画に使える色は，1つのキャラクタにつき16色であり，65,536色から任意の16色を選んだ16種類のテーブル(スプライトパレット)の中から，任意の1つを指定することで選びます。PCGは，CPUからみれば，ワードかロングワード単位でしかアクセスできないRAMにみえます。PCG先頭アドレスから1キャラクタ単位で番号がふられています。この番号がPCG番号です。後で説明するスプライトスクロールレジスタで，どのキャラクタを表示するのかを選ぶのに，このPCG番号を指定するような仕組みになっています。キャラクタのドットとハードウェアの対応は，Fig. 3.9のようになっています。



●Fig. 3.9　キャラクタのドットとハードウェアの対応

バックグランド画面(後で説明します)のPCGと兼用になっていますので，ちょっとややこしいキャラクタのドット座標とハードウェアの対応になっています。Cの構造体(共用体)で表すと，

```
/* PCG レジスタの構造体 */
typedef  union  {
    unsigned short dummy;
    struct {
      unsigned pos0: 4;
      unsigned pos1: 4;
      unsigned pos2: 4;
      unsigned pos3: 4;
    } sp;
} SP_REG_U;
```

です。新しい概念が出てきました。**共用体**と**ビットフィールド**です。まずはビットフィールドから。ビットフィールドというのは，CPUの最小の処理単位であるバイトをさらに分割して，ビットごとに処理できるようにするためのデータ構造です。ビットフィールドは，Cの規格であるANSIにおいても非常に機種依存が強いデータ構造です。ビットフィール

ドは，ビットを単なるフラグとして用いるような論理的な使い方以外では，ほとんど互換性がないと考えてかまいません。**X68000/X68030** の **GCC** では，ビットごとに上位から順次埋めていきます。今，説明している PCG では，4 ビットごとに 1 ドットが対応し，上位が画面左側なので，ビットフィールドで表現すると，直感的なデータ構造と実体がピッタリ合致します。

具体的な記述は，

```
struct {
  unsigned pos0: 4;
  unsigned pos1: 4;
  unsigned pos2: 4;
  unsigned pos3: 4;
} sp;
```

のように，構造体のメンバ宣言が

```
unsigned 割り当てフィールド名前 : ビット数;
```

の形式になった構造体宣言です。ビットフィールドのメンバは，指定したビット数の符号なし整数として扱われます。たとえば，

```
sp.pos1 += 1;
```

は，sp全体で 16 ビットですが，その 16 ビットの上位 4 ビット，下位 8 ビットを変化させないで，pos1が占めている 4 ビットだけを符号なしで 1 加算します。ビットフィールドでは非常に細かいビット操作が可能ですが，ビットフィールドを直接機械語で扱えない 68000/680EC30 では，大きく遅いコードにコンパイルされます。ところが，PCG では，このビットごとに分割されたメンバに対してはほとんど代入しか行わないので，それほど遅くなるわけではありません。

このビットフィールドで PCG をすべて扱うことはできますが，「遅くて大きいコード」にコンパイルされますので，スピード上不利です。しかも，ドットごとに処理するようなことは非常にまれで，ハードウェア上，PCG は 16 ビット単位，**short int** で処理できます。C では，このように同一のデータを時と場合で異なった種類のデータとして扱う際に，共用体というデータ構造を用います。この場合は，横方向 4 ドットで 1 単位として扱えば楽です。

```
/* PCG レジスタの構造体 */
union  {
    unsigned short dummy;
    struct {
      unsigned pos0: 4;
      unsigned pos1: 4;
```

```
        unsigned pos2: 4;
        unsigned pos3: 4;
      } sp;
} pcg;
```

これがその PCG に対応した共用体の宣言です。共用体pcgは,「符号なしの **short int** 型の変数であるdummyと，ビットフィールド構造体spとが重なった構造をしている」と宣言しています。この例は少し複雑なので，もっと簡単な例を出してみましょう。

```
union {
   int val;
   int *ptr;
} int_point;
```

これは **int** と **int** *が重なった例です。「ある場合には中身は整数，ある場合は中身はポインタ」といったデータがほしい場合によく使われます。

```
int_point.val = 50;
```

これは **int** として扱った場合の代入です。

```
int_point.ptr = &foo;
```

これは，int fooが宣言してあった場合のポインタとしての代入です。構造体として扱う場合と異なるのは，int_point.valとint_point.ptrはまったく同じメモリに存在しているという点です。コンパイラは，そのメモリに現在ポインタとしての値が入っているか，整数としての値が入っているかは管理しません。プログラマが責任をもって中の値の意味を管理しなければいけません。

共用体で，大きさが異なったデータを同一化した場合のデータの詰め方は，機種に依存します。たとえば，

```
union {
  int int_val;
  char char_val;
} int_char;
```

の場合で，

```
int_char.char_val = 10;
```

と初期化した後，

```
int i;
i = int_char.int_val;
```

のような参照は完全に機種依存となっています。また，**X68000/X68030** では，特に指定をしない限り，**int** とポインタは同一サイズなので，整数とポインタの相互変換を安易に行ってしまいますが，**MS–DOS** のコンパイラでは，この 2 つはサイズが異なっています。**UNIX** 上のソースプログラムでは，この両者の区別があいまいなことが非常に多いので，**MS–DOS** に移植する場合は注意してください。

ビットフィールドと共用体は，PCG のようなハードウェアを直接 C に置き換えるのには便利な記述方法ですが，非常に機種依存性が強いので，十分に注意して使うようにしましょう。共用体や構造体のデータの詰め方も機種に依存します。**X68000/X68030** の **GCC** では，この構造体や共用体の詰め方までオプションで制御できますが，コンパイラが構造体をどのように扱うかを十分に理解してから使うようにしてください。

3.5.2 ■ スプライトスクロールレジスタ

次に，PCG に対してもう一方のハードウェアである，スプライトスクロールレジスタについて説明します。

スプライトスクロールレジスタは，スプライトの画面上の位置や，スプライトの表示方法，表示色などをハードウェアに通知するためのレジスタです。このレジスタは全部で 128 個あり，この数が同時表示可能数 128 としてカタログに記載されているわけです。このスプライトスクロールレジスタにも 0 から 127 までの番号があり，番号の小さいものが優先順位が高くなっています。**X68000/X68030** には，スプライト同士の衝突をハードウェアで通知する機能は備わっていません。ですから，スプライト同士の衝突判定は，すべてソフトウェアで行います。

スプライトスクロールレジスタには，たくさんの機能が備わっています。C でこのレジスタを表してみましょう。

```
/* スプライトスクロールレジスタ構造体 */
typedef struct {
  short int sp_x;                  /* 表示座標 横方向 */
  short int sp_y;                  /* 表示座標 縦方向 */
  union {
    short int dummy;               /* 共用体ダミー */
    struct {
      unsigned v_invert : 1;       /* 垂直方向反転ビット */
      unsigned h_invert : 1;       /* 水平方向反転ビット */
      unsigned dummy    : 2;       /* 使われていません */
      unsigned color    : 4;       /* スプライトパレット */
      unsigned sp_code  : 8;       /* PCGコード */
    } sp;
  } sp_ctrl;
  union {
    short int dummy;               /* 共用体ダミー */
```

```
    struct {
      unsigned dummy   : 13;        /* 使われていません */
      unsigned ext     :  1;        /* 拡張用，使われていません */
      unsigned pwr     :  2;        /* バックグランドとのプライオリティ */
    } sp;
  } sp_pwr;
} SP_CTRL;
```

かなり複雑な構造をしていますが，全体では4ワードのレジスタです。

```
short int sp_x;                    /* 表示座標 横方向 */
short int sp_y;                    /* 表示座標 縦方向 */
```

これは表示座標です。スプライトは仮想画面 1024×1024 の任意の位置を指定でき，実際に表示されるのは，256×256 か 512×512 です。実際の表示座標は 256×256 で $(16,16) - (271,271)$，512×512 で $(16,16) - (527,527)$ です。

```
unsigned v_invert : 1;         /* 垂直方向反転ビット */
unsigned h_invert : 1;         /* 水平方向反転ビット */
```

これらは，スプライトパターンの上下左右の反転を指定するビットです。同じ機能を**FM–TOWNS**では「回転機能」と称していますが，ゲームセンターのマシンで使われる「回転機能」はこんな単純なものではなく，任意の角度でパターンを回転できるものです。これは単なる「反転機能」です。

```
unsigned color    : 4;         /* スプライトパレット */
```

16組あるスプライトパレットのうち，どれを使うのかを指定するビットです。ここを操作することで，スプライトの表示色を一瞬にして変更することができます。

```
unsigned sp_code  : 8;         /* PCGコード */
```

3.5.1「プログラマブルキャラクタジェネレータ」(P.118) で説明したPCG番号を指定します。同じパターンのキャラクタは同じPCG番号を共有します。

```
unsigned pwr      :  2;        /* バックグランドとのプライオリティ */
```

バックグランド画面との優先順位を指定するビットです。バックグランド画面については後述します。ここをすべて0にしておくと，スプライトは表示されません。

なお，PCGとスプライトスクロールレジスタは，任意の時間にCPUが書き換えを行うことができますが，表示期間にアクセスすると，CPU側にウエイトが入ることがあります。基本的に，表示期間中はこれらのハードウェアにアクセスしないのが原則です。また，垂直帰線期間中に書き換えを行いますから，この間に余計なウエイトが入らないように設

定ができるようになっています。これについては次項で説明します。

3.5.3 ■ スプライトのデモプログラム

ここでは，実際にスプライトを動かしてみます。サンプルプログラムとはいえ，スプライトを綺麗に動かすために，ソースが相当に難しくなっています。メゲないで頑張ってください。

List 3.5

```
 1: /* Pattern DEF */
 2: unsigned char
 3: pat_X[] =
 4: {
 5:     4,4,4,4,4,4,4,1,4,4,4,4,4,4,0,0,
 6:     4,4,4,4,4,4,4,1,1,4,4,4,4,0,0,0,
 7:     0,4,4,4,4,4,4,4,1,4,4,4,4,0,0,0,
 8:     0,4,4,4,4,4,4,4,1,1,4,4,4,0,0,0,
 9:     0,0,4,4,4,4,4,4,4,1,4,4,0,0,0,0,
10:     0,0,4,4,4,4,4,4,4,1,1,4,0,0,0,0,
11:     0,0,0,4,4,4,4,4,4,4,1,0,0,0,0,0,
12:     0,0,0,4,4,4,4,4,4,4,1,0,0,0,0,0,
13:     0,0,0,1,4,4,4,4,4,4,4,0,0,0,0,0,
14:     0,0,4,1,4,4,4,4,4,4,4,0,0,0,0,0,
15:     0,4,4,1,1,4,4,4,4,4,4,4,0,0,0,0,
16:     0,4,4,4,1,4,4,4,4,4,4,4,0,0,0,0,
17:     0,4,4,4,1,1,4,4,4,4,4,4,4,0,0,0,
18:     4,4,4,4,4,1,4,4,4,4,4,4,4,0,0,0,
19:     4,4,4,4,4,1,1,4,4,4,4,4,4,4,0,0,
20:     4,4,4,4,4,4,1,4,4,4,4,4,4,4,0,0
21: };
22:
23: unsigned char
24: pat_6[] =
25: {
26:     0,0,0,0,13,13,13,13,13,13,13,0,0,0,0,0,
27:     0,0,0,13,12,12,12,12,12,12,12,13,0,0,0,0,
28:     0,0,0,13,12,11,11,11,11,11,12,12,13,0,0,0,
29:     0,0,13,12,11,0,0,0,0,0,11,12,13,0,0,0,
30:     0,0,13,12,11,0,0,0,0,0,0,12,12,0,0,0,
31:     0,0,13,12,11,0,0,0,0,0,0,0,0,0,0,0,
32:     0,0,13,12,12,12,12,12,12,12,11,0,0,0,0,0,
33:     0,0,13,12,13,13,13,13,13,13,12,11,0,0,0,0,
34:     0,0,13,13,12,11,11,11,11,11,13,12,11,0,0,0,
35:     0,0,13,12,11,0,0,0,0,0,0,13,12,11,0,0,
36:     0,0,13,12,11,0,0,0,0,0,0,13,12,11,0,0,
```

```
37:     0,0,13,12,11,0,0,0,0,0,0,13,12,11,0,0,
38:     0,0,13,12,11,0,0,0,0,0,0,13,12,11,0,0,
39:     0,0,13,12,12,11,0,0,0,0,0,13,12,11,0,0,
40:     0,0,0,13,13,12,12,12,12,12,13,12,11,0,0,0,
41:     0,0,0,0,0,13,13,13,13,13,12,11,0,0,0,0
42: };
43:
44: unsigned char
45: pat_8[] =
46: {
47:     0,0,0,0,13,13,13,13,13,13,13,12,11,0,0,0,
48:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
49:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
50:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
51:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
52:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
53:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
54:     0,0,0,0,13,13,13,13,13,13,13,12,11,0,0,0,
55:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
56:     0,0,13,12,11,11,0,0,0,0,0,0,13,12,11,0,
57:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
58:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
59:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
60:     0,0,13,12,11,11,0,0,0,0,0,0,13,12,11,0,
61:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
62:     0,0,0,0,13,13,13,13,13,13,13,12,11,0,0,0
63: };
64:
65: unsigned char
66: pat_0[] =
67: {
68:     0,0,0,0,13,13,13,13,13,13,13,12,11,0,0,0,
69:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
70:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
71:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
72:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
73:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
74:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
75:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
76:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
77:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
78:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
79:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
80:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
81:     0,0,13,12,11,0,0,0,0,0,0,0,13,12,11,0,
82:     0,0,0,13,12,12,12,12,12,12,12,13,12,11,0,0,
```

```
 83:     0,0,0,0,13,13,13,13,13,13,13,12,11,0,0,0
 84: };
 85:
 86: unsigned char
 87: pat_C[] =
 88: {
 89:     0,0,0,0,0,4,4,5,5,5,5,0,0,0,0,0,
 90:     0,0,0,0,3,4,4,4,4,4,5,5,0,0,0,0,
 91:     0,0,0,3,3,3,3,3,4,4,4,5,5,0,0,0,
 92:     0,0,3,3,3,3,3,3,3,4,4,4,5,5,0,0,
 93:     0,0,3,3,3,3,3,3,3,3,4,4,4,5,0,0,
 94:     0,3,3,2,2,2,3,3,3,3,3,4,4,4,5,0,
 95:     0,3,2,2,2,2,2,2,3,3,3,4,4,4,5,0,
 96:     0,2,2,2,2,2,2,2,2,3,3,3,4,4,4,0,
 97:     0,2,2,2,1,1,2,2,2,2,3,3,3,4,4,0,
 98:     0,2,2,1,1,1,1,2,2,2,3,3,3,4,4,0,
 99:     0,2,2,1,1,1,1,2,2,2,3,3,3,4,4,0,
100:     0,0,2,1,1,1,1,2,2,2,3,3,3,4,0,0,
101:     0,0,2,1,1,1,1,2,2,2,3,3,3,4,0,0,
102:     0,0,0,1,1,1,1,2,2,2,3,3,4,0,0,0,
103:     0,0,0,0,1,1,2,2,2,3,3,3,4,0,0,0,
104:     0,0,0,0,1,2,2,2,3,3,3,4,0,0,0,0
105: };
106:
107: int
108: sp_pal[] =
109: {
110:   0,21140,33824,44394,
111:   54964,63420,38,48,
112:   54,62,1024,1344,
113:   1664,1984,0,0
114: };
```

List 3.5 は，動かすスプライトのパターンデータです。これは，**X68000/X68030** に付属のスプライト作成ツールが出力する **X–BASIC** プログラムを C に直したものです。配列の初期化定義を使っています。

```
int foo[] = { 0,1,2,3 };
```

このように，配列fooを初期化して宣言することができます。このとき，配列の要素の数はコンパイラが数えてくれます。また，

```
int foo[10] = { 0,1,2,3 };
```

のように，要素数を指定して初期化することもできます。この場合，宣言した要素数と初

期化の要素の並びの数が一致しない場合は，先頭から順番に埋めていき，足りない部分は0で初期化されます。また，初期化要素の並びのほうが数が多い場合は，コンパイラは警告を出して，よぶんな初期化要素を無視します。List 3.5では，'X''6''8''0''C' のキャラクタを定義しています。最後の変数sp_pal[]は，スプライトパレットの設定データです。このパターンデータは，ハードウェアの構造に直接対応していないので，ハードウェアに転送する場合には少々変換が必要になります。

List 3.6

```
 1: #include <interrupt.h>
 2: #include <stdlib.h>
 3: #include <iocslib.h>
 4:
 5: /* パターン定義のヘッダ */
 6: #include "sp_test.h"
 7:
 8: /* 実際のスプライト数はこの値の6倍 */
 9: #define MAX_SP 20
10:
11: /* スプライト間のドット数 */
12: #define STEP   3
13:
14: /* 機種，コンパイラに完全に依存する記述 */
15: /* PCG レジスタの構造体 */
16: typedef  union  {
17:     unsigned short dummy;
18:     struct {
19:       unsigned pos0: 4;
20:       unsigned pos1: 4;
21:       unsigned pos2: 4;
22:       unsigned pos3: 4;
23:     } sp;
24: } SP_REG_U;
25:
26: /* スプライトスクロールレジスタ構造体 */
27: typedef struct {
28:   short int sp_x;
29:   short int sp_y;
30:   union {
31:     short int dummy;
32:     struct {
33:       unsigned v_invert : 1;
34:       unsigned h_invert : 1;
35:       unsigned dummy    : 2;
36:       unsigned color    : 4;
```

```
37:       unsigned sp_code  : 8;
38:     } sp;
39:   } sp_ctrl;
40:   union {
41:     short int dummy;
42:     struct {
43:       unsigned dummy   : 13;
44:       unsigned ext     :  1;
45:       unsigned pwr     :  2;
46:     } sp;
47:   } sp_pwr;
48: } SP_CTRL;
49:
50: /* スプライトを扱う配列 */
51: static SP_CTRL sp_disp[128];
52: static short int this_sp_is_active[128];
53: static short int life_of_this_sp[128];
54:
55: /* 既定移動パターンのテーブル */
56: static short *move_tbl_x;
57: static short *move_tbl_y;
58:
59: static void
60: make_move_tbl (void)
61: {
62:   extern double sin (double);
63:   int i;
64:   move_tbl_x = (short *) malloc (sizeof (short) * (256 + 33));
65:   move_tbl_y = (short *) malloc (sizeof (short) * (256 + 33));
66:   for (i = -16; i < 256 + 16; i++)
67:     {
68:       move_tbl_x[i] = i;
69:       move_tbl_y[i] = 128 +
70:         (int)(100.0 * sin (3.1415 * (((double) i)/64.0)));
71:     }
72: }
73:
74: /* スプライトパターンの定義 */
75: static void
76: def_sp (int pat_no, unsigned char *sp_dat)
77: {
78:    SP_REG_U *sp_reg0 = (SP_REG_U *) 0xeb8000
                          + (128/sizeof (SP_REG_U)) * pat_no;
79:    SP_REG_U *sp_reg1 = (SP_REG_U *) 0xeb8040
                          + (128/sizeof (SP_REG_U)) * pat_no;
80:    int i;
```

```
 81:    /* SP スクロールレジスタをCPUに開放 */
 82:    *(short *)0xeb0808 &= 0xfdfc;
 83:    for (i = 0; i < 16; i++)
 84:      {
 85:         SP_REG_U tem;
 86:         tem.sp.pos0 = *sp_dat ++;
 87:         tem.sp.pos1 = *sp_dat ++;
 88:         tem.sp.pos2 = *sp_dat ++;
 89:         tem.sp.pos3 = *sp_dat ++;
 90:         *sp_reg0 ++ = tem;
 91:         tem.sp.pos0 = *sp_dat ++;
 92:         tem.sp.pos1 = *sp_dat ++;
 93:         tem.sp.pos2 = *sp_dat ++;
 94:         tem.sp.pos3 = *sp_dat ++;
 95:         *sp_reg0 ++ = tem;
 96:
 97:         tem.sp.pos0 = *sp_dat ++;
 98:         tem.sp.pos1 = *sp_dat ++;
 99:         tem.sp.pos2 = *sp_dat ++;
100:         tem.sp.pos3 = *sp_dat ++;
101:         *sp_reg1 ++ = tem;
102:         tem.sp.pos0 = *sp_dat ++;
103:         tem.sp.pos1 = *sp_dat ++;
104:         tem.sp.pos2 = *sp_dat ++;
105:         tem.sp.pos3 = *sp_dat ++;
106:         *sp_reg1 ++ = tem;
107:      }
108:   /* SP 表示 */
109:   *(short *)0xeb0808 |= 0x0200;
110: }
111:
112:
113: /* inline アセンブラサンプルになります */
114:
115: static void
116: init_screen (void)
117: {
118:   {
119:     /* screen , 256*256 256 color,high freq */
120:     register int trapNo asm ("d0");
121:     register int mode   asm ("d1");
122:     trapNo = 0x10;
123:     mode   = 10;
124:     asm ("trap #15":: "d"(trapNo),"d"(mode):"d0");
125:   }
126:
```

```
127:   {
128:     /* カーソル off */
129:     register int trapNo asm ("d0");
130:     trapNo = 0x1f;
131:     asm ("trap #15":: "d"(trapNo):"d0");
132:   }
133:
134:   {
135:     /* スプライト面初期化 */
136:     register int trapNo asm ("d0");
137:     trapNo = 0xc0;
138:     asm ("trap #15":: "d"(trapNo):"d0");
139:   }
140:
141:   {
142:     /* PCG クリア */
143:     register int trapNo asm ("d0");
144:     register int pcg asm ("d1");
145:     for (pcg = 0; pcg < 256; pcg ++)
146:       {
147:         trapNo = 0xc3;
148:         asm ("trap #15"::"d"(trapNo), "d"(pcg): "d0");
149:       }
150:   }
151:
152:   {
153:     /* スプライトパレット設定 */
154:     register int trapNo    asm ("d0");
155:     register int pal_code  asm ("d1");
156:     register int block     asm ("d2");
157:     register int col_code  asm ("d3");
158:     int i;
159:     for (i = 0; i < 16; i++)
160:       {
161:         trapNo = 0xcf;
162:         pal_code = 0x80000000 | i;
163:         block = 1;
164:         col_code = sp_pal[i];
165:         asm ("trap #15"::"d"(trapNo),"d"(pal_code),"d"(block),
                    "d"(col_code));
166:       }
167:   }
168:
169:   {
170:     /* スプライト表示 on */
171:     register int trapNo asm ("d0");
```

```
172:     trapNo = 0xc1;
173:     asm ("trap #15":: "d"(trapNo):"d0");
174:   }
175:   /* PCG 定義 */
176:   def_sp (0, pat_X);
177:   def_sp (1, pat_6);
178:   def_sp (2, pat_8);
179:   def_sp (3, pat_0);
180:   def_sp (4, pat_C);
181: }
182:
183: static volatile int vsync_counter;
184: static volatile int sp_is_ready;
185:
186: static void
187: vsync_disp ()
188: {
189:   if (sp_is_ready)
190:     {
191:       int i;
192:       SP_CTRL *sp_scr_reg = (SP_CTRL *)0xeb0000;
193:
194:       /* SP スクロールレジスタをCPUに開放 */
195:       *(short *)0xeb0808 &= 0xfdfc;
196:
197:       /* スプライトスクロールレジスタ書き込み */
198:       for (i = 0; i < 128; i++)
199:         if (this_sp_is_active[i])
200:             *sp_scr_reg ++ = sp_disp[i];
201:
202:       /* SP 表示 */
203:       *(short *)0xeb0808 |= 0x0200;
204:
205:       vsync_counter++;
206:       sp_is_ready = 0;
207:     }
208:   IRTE ();
209: }
210:
211: void
212: main()
213: {
214:   B_SUPER (0);
215:   make_move_tbl ();
216:   init_screen ();
217:   vsync_counter = 0;
```

```
218:   sp_is_ready = 0;
219:
220:   {
221:     int i;
222:     for (i = 0; i < 6*MAX_SP; i++)
223:       {
224:         life_of_this_sp[i] = - STEP * (i + 1);
225:         sp_disp[i].sp_ctrl.sp.color = 1;
226:         sp_disp[i].sp_pwr.sp.pwr = 3;
227:       }
228:   }
229:
230:   {
231:     int i;
232:     for (i = 0; i < 6*MAX_SP; i += 6)
233:       {
234:         sp_disp[0 + i].sp_ctrl.sp.sp_code = 3; /* '0' */
235:         sp_disp[1 + i].sp_ctrl.sp.sp_code = 3; /* '0' */
236:         sp_disp[2 + i].sp_ctrl.sp.sp_code = 3; /* '0' */
237:         sp_disp[3 + i].sp_ctrl.sp.sp_code = 2; /* '8' */
238:         sp_disp[4 + i].sp_ctrl.sp.sp_code = 1; /* '6' */
239:         sp_disp[5 + i].sp_ctrl.sp.sp_code = 0; /* 'X' */
240:       }
241:   }
242:
243:   VDISPST (vsync_disp, 0, 1);
244:   while (1)
245:     {
246:       if (!sp_is_ready)
247:         {
248:           int i;
249:           for (i = 0; i < 6*MAX_SP; i++)
250:             {
251:               life_of_this_sp [i] ++;
252:               if (life_of_this_sp[i] >= 256 + 16)
253:                 life_of_this_sp[i] = -16;
254:               if (life_of_this_sp[i] >= -16)
255:                 {
256:                   sp_disp[i].sp_x = move_tbl_x[life_of_this_sp[i]];
257:                   sp_disp[i].sp_y = move_tbl_y[life_of_this_sp[i]];
258:                   this_sp_is_active[i] = 1;
259:                 }
260:             }
261:           sp_is_ready = 1;
262:         }
263:       while (sp_is_ready)
```

```
264:          ;
265:        while (BITSNS (0xe) & 1)
266:          ;
267:        if (BITSNS (0) & 2)
268:          break;
269:      }
270:    VDISPST (0, 0, 0);
271:    {
272:      /* screen 0,0 */
273:      register int trapNo asm ("d0");
274:      register int mode   asm ("d1");
275:      trapNo = 0x10;
276:      mode   = 16;
277:      asm ("trap #15":: "d"(trapNo),"d"(mode):"d0");
278:    }
279:
280:    {
281:      /* カーソル on */
282:      register int trapNo asm ("d0");
283:      trapNo = 0x1e;
284:      asm ("trap #15":: "d"(trapNo):"d0");
285:    }
286:    exit (0);
287: }
```

List 3.6 がスプライトのデモプログラム全体です。できるだけライブラリを使わないで，自前で処理するように記述してあるために，かなり内容が難しくなっていますが，基本的なアルゴリズムは，Fig. 3.3「表示期間と非表示期間」(P.77) のように，データを用意しては垂直帰線期間に書き換えを行うことを繰り返しているだけです。まずは変数の説明からです。

```
50: /* スプライトを扱う配列 */
51: static SP_CTRL sp_disp[128];
```

これがスプライトスクロールレジスタの作業用バッファです。表示期間中にこのバッファにデータを用意して，割り込みがそれをハードウェアに一気に転送します。このとき，毎回 128 個も転送するのはムダなので (スプライトをたくさん使う本格的なゲームなら，実際に表示しているかどうかチェックなどせずに，常時 128 個転送してもいいのです)，128 個あるスプライトのうち，どのスプライトを「使っているか，使っていないか」を判断するテーブルが，

```
52: static short int this_sp_is_active[128];
```

です。他にスプライトの座標を管理するためのテーブルとして，

```
53: static short int life_of_this_sp[128];
```

を用意しています。これは，時間の経過によってスプライトの位置を動かすためのカウンタとして使われています。

```
55: /* 既定移動パターンのテーブル */
56: static short *move_tbl_x;
57: static short *move_tbl_y;
```

この 2 つのポインタは，スプライトを *sin* カーブのように動かすための，*sin* カーブデータテーブルを作成するための変数です。次の関数`make_move_tbl()`で，*sin* カーブのテーブルを作成しています。`make_move_tbl()`は，C ライブラリ関数の`double sin()`を使って，画面上の x 座標に対する y座標の値のテーブルを作成しています。`move_tbl_x`, `move_tbl_y`に演算子`[]`をつけて，配列のように使っています。この意味がわからない方は，2.7.3「配列とポインタ」(P.58) まで戻って再度読み直してください。次の関数`def_sp()`は，PCG エリアにスプライトパターンを転送しています。配列で定義してあるスプライトパターンは，直接ハードウェアに転送できない形式になっているので，いったんハードウェアに転送できる形式である`SP_REG_U`型に変換してから，ハードウェアに転送を行っています。

```
81:     /* SP スクロールレジスタをCPUに開放 */
82:     *(short *)0xeb0808 &= 0xfdfc;
```

この部分が，3.5.2「スプライトスクロールレジスタ」(P.122) の最後で説明した，CPU に対してウエイトが入らないようにする処理です。ビット演算子`&=`が使われています。C では，ビット操作の演算子として`&`, `|`, `~`, `^`, `>>`, `<<`が使えます (ビット操作については 1.3.2「負の数を 2 進数で扱う」(P.8) で説明していますので，参照してください)。

●Table 3.2　ビット演算子

ビット演算子	意味
op0 & op1	ビットごとの AND
op0 | op1	ビットごとの OR
~op1	ビットごとの NOT
op0 ^ op1	ビットごとの XOR
op0 >> op1	op0をop1ビット右シフト
op0 << op1	op0をop1ビット左シフト

演算子`&=`は，`+=`, `-=`などと同様の処理を行う演算子です。シフト命令のシフト量に負の値を指定した場合，その結果は不定です。エラーチェックは行われません。シフト演算子に慣れてくると，2 のべき乗での割り算や掛け算をシフトで解決しようとすることもあります。たとえば，

```
array[i*4][j/2] = 0;
```

を

```
array[i<<2][j>>1] = 0;
```

と記述する方法です。これは，昔コンパイラがあまり賢くなくて，配列の要素のアドレスを決定するのに素直に掛け算命令を使うような時代の産物です。今の最適化コンパイラでは，このような妙な工夫をしなくても，2のべき乗の掛け算や割り算をシフト命令に置き換えるくらいのことは簡単にやってくれます。素直に割り算は割り算，掛け算は掛け算として記述すればよいのです。プログラムを読む人に，「なぜ? これはシフトなの??」と余計な疑問を抱かさないためにも，「計算のためのシフト記述」は避けるべきです。シフト命令は「本当にシフトする」場合にだけ使えば，意味がわかりやすいプログラムになります。

話題が少しそれました。

```
81:     /* SP スクロールレジスタをCPUに開放 */
82:     *(short *)0xeb0808 &= 0xfdfc;
```

これは68000のアセンブラでは，`and.w #$fdfc,$eb0808`を期待する記述です。`$fdfc`は2進数で1111110111111100ですから，メモリの`$eb0808`番地の1，2，10ビットを0にしています。1，2ビットにはバックグランドの表示画面を指定するのですが，ここではバックグランドは使っていませんので，0でクリアしています。10ビットを0にすると，スプライト関係の表示は禁止されて，PCGやスプライトスクロールレジスタを操作する場合にCPUに対してウエイトが入らなくなります。PCGエリアをCPUに開放させた後，スプライトパターンのデータ配列からPCGにデータを転送しています。このとき，同時にデータのコンバートも行っています。

次の`init_screen()`関数では，**GCC**の拡張である**asm**文を多用してハードウェアの初期化を行っています。この**asm**文については，ある程度のアセンブラの知識が必要です。本書はアセンブラについてはあまり詳しく扱わないので，ここでは，この部分の説明は割愛させていただきます。詳しい**asm**文の説明については，「X68k Programming Series #1 Develop.」の**GCC**の**asm**文についての説明を参照してください。これらはすべてIOCSコールですから，ライブラリ関数に置き換えが可能です。

関数`vsync_disp()`が垂直帰線期間割り込み処理の関数です。この関数で，配列`sp_disp`に用意されたスプライトスクロールレジスタのデータをハードウェアに転送しています。割り込み処理と通常処理との情報の受け渡しの同期は，変数`sp_is_ready`を使って行っています。`sp_is_ready`は，P.111で説明したように，**volatile**属性を付加しておかないと，通常の処理が無限ループに陥ることがあります。

関数`main()`は，一連のハードウェアの初期化を終えた後に，割り込みの登録やスプライトスクロールレジスタ関係のデータの初期化を行い，メインの処理であるループに入ります。この部分の処理を取り出して，詳しく説明します。

```
while (1)
  {
    /* 割り込みによってsp_is_readyは0にされる */
```

```
    /* これはハードへの転送が終わったことの通知である */
    if (!sp_is_ready)
      {
        /* スプライトの次の表示位置を決定する。毎回計算では
           処理が遅いので，make_sin_tbl()で作成したテーブルを
           スプライトの表示経過時間から配列参照で決定する */
        int i;
        for (i = 0; i < 6*MAX_SP; i++)
          {
            /* スプライトの経過時間を増やす */
            life_of_this_sp [i] ++;

            /* 画面の外にはみ出す場合は最初に戻す */
            if (life_of_this_sp[i] >= 256 + 16)
              life_of_this_sp[i] = -16;

            /* 表示されているスプライトについて座標を設定する */
            if (life_of_this_sp[i] >= -16)
              {
                sp_disp[i].sp_x = move_tbl_x[life_of_this_sp[i]];
                sp_disp[i].sp_y = move_tbl_y[life_of_this_sp[i]];
                this_sp_is_active[i] = 1;
              }
          }
        /* データが用意できたので割り込み関数に教える */
        sp_is_ready = 1;
      }
    /* 割り込みがデータを転送するまで待つ  */
    /* sp_is_readyがvolatileでないとここで
       無限ループになる */
    while (sp_is_ready)
      ;
    /* Shiftキーが押されたら離されるまで待つ */
    while (BITSNS (0xe) & 1)
      ;
    /* ESCキーが押されたらループを脱出 */
    if (BITSNS (0) & 2)
      break;
  }
```

ESC キーが押されてループを脱出したら，まず割り込みを禁止してから，画面関係を普通の高解像度モードに戻して`exit()`します。Shift キーで処理を停止させる部分は，ゲー

ムの一時中断処理のサンプルでもあります。このサンプルデモプログラムも，割り込みを使って画面の乱れを完全に抑えてあります。そこで，スクロールの場合のように，表示のタイミングを完全に無視して書き換えてみましょう。List 3.7がその変更の抜粋です。スプライトスクロールレジスタへの書き込みを，実際のディスプレイの表示タイミングとまったく無関係に行うように書き換えてあります。なお，必要のない処理が盲腸のように残っていますが，現在のテーマには直接関係がありませんので，気にしないでください。スプライトスクロールレジスタに転送するときに，CPU側にスプライト関係のレジスタを開放させる処理を加えると，画面の乱れがひどくなるので，プリプロセッサ命令を使ってその部分を取り除くようにしてあります。

```
#if 0
```

と

```
#endif
```

で囲まれた部分がその部分です。プリプロセッサは，#if 0をいつも「偽」として扱いますので，この#if 0から#endifまでの部分はソースから削除されます。この#if 0～#endif(つねに削除される)と，#if 1～#endif(つねに残される)は，デバッグやこのようなテストを行う場合にしばしば使われます。デバッグの際，「どうもここらへんの処理がおかしい」といった場合に，デバッグ用の処理を追加するときがあります。デバッグが終わってソースを綺麗にする場合も，このような追加処理は削除せずに，#if 0～#endifを使ってソースの上では見えるように残しておくのが後々を考えると便利です。

List 3.7

```
....
....
static void
sp_display ()
{
  if (sp_is_ready)
    {
      int i;
      SP_CTRL *sp_scr_reg = (SP_CTRL *)0xeb0000;
#if 0
      /* SP スクロールレジスタをCPUに開放 */
      *(short *)0xeb0808 &= 0xfdfc;
#endif
      /* スプライトスクロールレジスタ書き込み */
      for (i = 0; i < 128; i++)
        if (this_sp_is_active[i])
            *sp_scr_reg ++ = sp_disp[i];
#if 0
```

```
      /* SP 表示 */
      *(short *)0xeb0808 |= 0x0200;
#endif
      vsync_counter++;
      sp_is_ready = 0;
    }
}

void
main()
{
  .....
  .....
  .....
  while (1)
    {
      int wait;
      wait = ONTIME();
      if (!sp_is_ready)
        {
          int i;
          for (i = 0; i < 6*MAX_SP; i++)
            {
              life_of_this_sp [i] ++;
              if (life_of_this_sp[i] >= 256 + 16)
                life_of_this_sp[i] = -16;
              if (life_of_this_sp[i] >= -16)
                {
                  sp_disp[i].sp_x = move_tbl_x[life_of_this_sp[i]];
                  sp_disp[i].sp_y = move_tbl_y[life_of_this_sp[i]];
                  this_sp_is_active[i] = 1;
                }
            }
          sp_is_ready = 1;
        }
      sp_display ();
      while (ONTIME() - wait < 2)
        ;
      while (BITSNS (0xe) & 1)
        ;
      if (BITSNS (0) & 2)
        break;
    }
  ....
  ....
```

少し話がそれました。実際に，この「表示タイミングを無視する」プログラムでは，たしかに表示が不自然になります。それでは，このような割り込みを使わない方法は絶対にうまくいかないのでしょうか? 実はそうではありません。3.2「ゲームの基本アルゴリズム」(P.76) で書きましたが，**X68000/X68030** では，ディスプレイの表示タイミングは「割り込み」の他に，ハードウェアを監視することにより実現できます。そこで，このプログラムをさらに変更して，割り込みを使わない綺麗な表示を行ってみましょう。

ディスプレイの表示タイミングは，**X68000/X68030** に備えられたMFP(マルチファンクションペリフェラル) というLSIのレジスタを監視することで知ることができます。垂直帰線期間は`$e88001`番地のビット8でわかります。このビットが0のときは，垂直帰線期間，1のときは表示期間です。そこで，List 3.8のように，関数`sp_display()`を変更します。

List 3.8

```
....
....
....
....
static void
sp_display ()
{
   /* MFP レジスタアドレス */
   char volatile *mfp = (char *)0xe88001;

   /* 垂直帰線期間なら表示期間を待つ */
   while (!((*mfp) & 0x10))
     ;
   /* 垂直帰線期間を待つ */
   while ((*mfp) & 0x10)
     ;
  if (sp_is_ready)
    {
      int i;
      SP_CTRL *sp_scr_reg = (SP_CTRL *)0xeb0000;
      /* SP スクロールレジスタをCPUに開放 */
      *(short *)0xeb0808 &= 0xfdfc;
      /* スプライトスクロールレジスタ書き込み */
      for (i = 0; i < 128; i++)
        if (this_sp_is_active[i])
            *sp_scr_reg ++ = sp_disp[i];
      /* SP 表示 */
      *(short *)0xeb0808 |= 0x0200;
      vsync_counter++;
      sp_is_ready = 0;
```

```
    }
}

void
main()
{
  ....
  ....
  ....
  while (1)
    {
      if (!sp_is_ready)
        {
          int i;
          for (i = 0; i < 6*MAX_SP; i++)
            {
              life_of_this_sp [i] ++;
              if (life_of_this_sp[i] >= 256 + 16)
                life_of_this_sp[i] = -16;
              if (life_of_this_sp[i] >= -16)
                {
                  sp_disp[i].sp_x = move_tbl_x[life_of_this_sp[i]];
                  sp_disp[i].sp_y = move_tbl_y[life_of_this_sp[i]];
                  this_sp_is_active[i] = 1;
                }
            }
          sp_is_ready = 1;
        }
      sp_display ();
      while (BITSNS (0xe) & 1)
        ;
      if (BITSNS (0) & 2)
        break;
    }
  ....
  ....
  ....
```

まずはMFPのレジスタを監視するためのポインタ変数を用意します。

```
/* MFP レジスタアドレス */
char volatile *mfp = (char *)0xe88001;
```

これがそのポインタ変数ですが，**volatile**に注目してください。**volatile**が，「予期しない変更が起こる変数」につけるキーワードであることは，何度か説明しましたが，この場合には，このmfpという変数に予期しない変更が起こるわけではなく，*mfp，つまりこの

ポインタが示しているアドレスのメモリ (この場合はMFPのレジスタです) に予期しない変更が起こるのです。こういった場合には，次のように

```
char volatile *
```

の順番で記述します。では，

```
char *volatile foo;
```

は，どういった意味になるのでしょうか?? これは，ポインタ変数fooに予期しない変更が起こることをコンパイラに通知します。**volatile**の置かれる位置によって，意味がまったく違ってきますので，十分に注意してください。

また，

```
/* MFP レジスタアドレス */
char *mfp = (char *)0xe88001;
```

と，**volatile**を忘れると悲惨です。コンパイラは，$e88001の値が時間によって変わるとは夢にも思わないので，*mfpを一度取り出したらもう二度と参照しないようなコードを生成します。コンパイラは，値がコロコロ変わるようなメモリはないと仮定して，コードを最適化するからです。

次の処理も重要です。

```
/* 垂直帰線期間なら表示期間を待つ */
while (!((*mfp) & 0x10))
  ;
/* 垂直帰線期間を待つ */
while ((*mfp) & 0x10)
  ;
```

この例のように，ハードウェアを監視する場合は，時間の経過を頭に置いて処理を考える必要があります。垂直帰線期間を検出する処理を行うときにはまず，現在が垂直帰線期間でないかどうかを確認する必要があります。なぜなら，現在がすでに垂直帰線期間なら，次の瞬間には表示期間になっている可能性があるからです。このような場合には，いったん垂直帰線期間の終了を待ってから次の垂直帰線期間を検出するようにしないと，垂直帰線期間に行うべき処理が表示期間に食い込むことが起こります。特に，ノーマルな**X68000/X68030**で多数のキャラクタを処理するような重いゲームでは，表示期間内にすべての準備が整わないことが頻繁に起こり得ます。

ただ単に

```
/* 垂直帰線期間を待つ */
while ((*mfp) & 0x10)
  ;
```

としたのでは，破綻する場合があることを理解していただけたでしょうか? 処理が間に合ってさえいれば，この監視に入る時間は必ず表示期間でしょうから，

```
/* 垂直帰線期間なら表示期間を待つ */
while (!((*mfp) & 0x10))
  ;
```

の部分は即座に抜けてくるでしょう。もし処理が間に合わないときに，ここでループすると，次の表示期間は何もハードウェアに変更が起こらないので，いわゆる処理落ちが起こるわけです。割り込み処理による表示の場合には，割り込み処理と通常処理で同期をとっているので，やはり処理が間に合わないと，垂直帰線期間に何もハードウェアを操作しないので，処理落ちが起きます。この点でも「監視を用いた方法」と「割り込みを用いた方法」ではまったく違いはなく，どちらの方法を採用するのかはプログラマの趣味の問題です。現在の傾向としては，ほとんどのシューティングゲームは割り込みを用いた方法を使っています。本書でも，次章のサンプルゲームでは，この割り込みを用いた方法ですべてのプログラムを行うことにします。なお，`sp_display()`により，プログラムはつねにハードウェアに同期して実行するようになったので，関数`main()`は余計な時間監視は行わず，単にループするように変更してあります。

CHAPTER 4

第4章

Cによる実践ゲーム制作

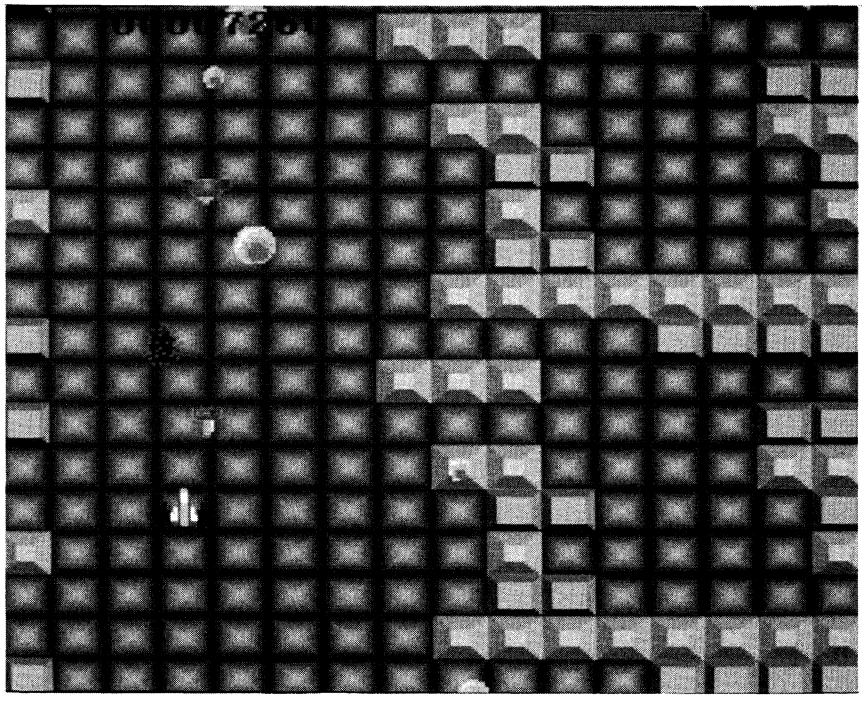
本章では実際にゲームを制作してみます。今までの知識だけで理解するには，かなり難しい内容もありますが，頑張って読んでみてください。本章で制作するゲームは次のような仕様になります。

1. グラフィック画面2面256色モード，バックグランド画面1面，縦スクロール。場面によってグラフィック画面のラスタースクロールを使う。
2. バックグランド画面はスコア等の表示に使う。
3. ジョイスティック使用。

実はこの仕様は最初からあったわけではなく，本書の大本である「Cマガジン」の連載記事の進行に従って結果的にこうなった，というかなりいい加減な仕様です。ですが，実際にゲームのプログラムを学習するには十分な内容になっていると思います。

■ 4.1 ゲームの内容

まず，本書で作成するゲームの具体的な内容について，簡単に説明しましょう。このゲームは，基礎的な縦スクロールのシューティングゲームです。ジョイスティックで自機を8方向に移動させて，画面上の敵キャラクタを破壊するという単純明快なものです。Fig. 4.1が実際のゲームの画面です。



●Fig. 4.1 ゲームの画面

グラフィック2面の多重スクロール，バックグランド画面によるスコア表示，半透明を

使った特殊効果，ラスタースクロール，パレット書き換えによるアニメーションと，いちおう一通りのゲームでの「芸」を実装しています。

このゲームはジョイスティックで操作を行います。操作方法は Table 4.1 のようになっています。

●Table 4.1　ゲームの操作方法

操作	機能
移動レバー	8 方向の移動
トリガ A	ゲームの中止
トリガ B	弾の発射
トリガ B(連続)	溜め撃ち
ESC キー	ゲームの一時停止

■ 4.2　ゲームソースの構成

このゲームのソースプログラムはすべて C で記述されています。アセンブラは，C ソースに埋め込んだ，インラインアセンブラの簡単なものしか使っていません。比較的簡単なソースなので，時間をタップリかけて読んでいただければそれほど難しくはないでしょう。

ゲームのソースは，各機能ごとに分割されて作成されています。『付録ディスク』をフロッピーディスクドライブに入れ，リセットキーを押すと，AUTOEXEC.BAT でこのゲームを実際に作成し，ゲームを起動します。初めて C に触る人でも，コンパイルが行われるようすを実際に目で見て確認することができます (本扉裏の「付録ディスクの使い方」をご覧ください)。

それでは，ソースの構成について説明しましょう。

Makefile
: GNU make 用の Makefile です。Makefile というのは，プログラムを構成するソースファイルの相互依存関係を記述しておいて，Make と呼ばれるプログラムを用いて自動でプログラムを生成するためのファイルです。Make や Makefile については，**XC Ver.2.0** のマニュアルや市販の参考書を読んでください。

mapdata.h
: 背景に使われるグラフィックの構成を記述したヘッダファイルです。map.c が **#include** しています。

sindata.h
: ラスタースクロール用の *sin* カーブテーブルが入ったヘッダです。これも map.c が**#include** しています。

上の 3 つのファイルはゲームプログラムの補助的なファイルで，本文では直接の説明をしていません。以下のファイル群がゲームを構成する主要なファイルです。

game.h　4.3「共通ヘッダの説明」(P.146以降で説明)

ゲームで使うデータの構造体の宣言や，外部変数の宣言，関数のプロトタイプ宣言等が収められています。全ソースが**#include**している共通なヘッダファイルです。

sprite.c　4.4「スプライト管理部分の作成」(P.163以降で説明)

スプライトの表示の管理と，垂直帰線期間割り込みの処理を行う関数が記述されています。

utiles.c　4.5「各種下請け処理の制作」(P.192以降で説明)

ゲームプログラムで使われる下請け関数や，初期化の処理を行う関数が記述されています。

mychr.c　4.6「自機の管理」(P.202以降で説明)

自機の表示や移動，弾の発射関係を処理する関数が記述されています。

enemy.c　4.7「敵キャラクタ管理」(P.211以降で説明)

敵キャラクタの発生や移動を管理する関数が記述されています。

clash.c　4.8「当たり判定とスプライトの消去」(P.215以降で説明)

当たり判定や，スプライトの消去処理を行う関数が記述されています。

map.c　4.9.1「グラフィック画面の処理」(P.223以降で説明)

グラフィック画面のスクロールや，ラスタースクロールのためのラスター割り込み処理関数が記述されています。

back.c　4.9.2「バックグランド画面の処理」(P.238以降で説明)

バックグランド画面の表示を処理する関数が記述されています。

game.c　4.10「メインルーチン」(P.242以降で説明)

ゲームプログラムのメインループをもつ最も上位の関数が記述されています。

市販のゲームには比較できないほどの小規模なゲームです。ソースの量からみれば，中規模アプリケーションに属するくらいの大きさになるでしょう。推測ですが，これくらいの規模のプログラムを全部自前で記述でき，アセンブラも使えるくらいの力量であれば，「0.8人前のゲームプログラマ」ということができるでしょうか(私が「一人前」かどうかは定かでありませんが･･･)。

■ 4.3　共通ヘッダの説明

List 4.1 は，制作するゲームの共通データ構造を定義しているヘッダです。普通，こういったヘッダは，ゲームの作成に従って完成されていく類の部分であり，最初からこのように完成された形になっているものではありません。ですから，最初からゲームを作成する場合には，このようなヘッダは空のファイルになっていることが普通です。このサンプルゲームではすでにゲーム全体が完成していますので，ここで全体のデータ構造が明確に

なるように詳しく説明をしておくことにします。

List 4.1 game.h

```
 1: /* C Magazine Sample ゲームヘッダ */
 2:
 3: /* コンフィグマクロ */
 4: /* XC (Ver.1 or Ver.2)のライブラリを使用する
 5:    場合はコメントを外してください */
 6: /* #define USE_XC_LIB */
 7:
 8: /* SP_REGST で引数の数がエラーになったら
 9:    コメントを外してください */
10: /* #define HAVE_SPREGST_BUGS */
11:
12: /* 溜め撃ちで自分の弾の威力を当たった数だけ減らしたい
13:    場合はコメントを外してください */
14: /* #define NON_LIKE_R_TYPE */
15:
16: /* 背景データをファイルで読む場合はコメントを外して
17:    ください */
18: /* #define MAP_IS_FILE */
19:
20: /* 背景描画を DMA で行わない場合はコメントを外してく
21:    ださい  */
22: /* #define SOFT_TRANS */
23:
24:
25:
26: #include <stdio.h>
27: #include <stdlib.h>
28: #include <string.h>
29: #include <io.h>
30:
31: #ifdef HAVE_SPREGIST_BUGS
32: #define SP_REGIST  XC2_BUG_PROTOS
33: #include <iocslib.h>
34: #undef SP_REGIST
35: int SP_REGST(int, int, int, int, int, int);
36: #else
37: #include <iocslib.h>
38: #endif
39:
40: #include <interrupt.h>
41: #include <setjmp.h>
42: #ifndef USE_XC_LIB
```

```
43: #include <signal.h>
44: #else
45: #include <doslib.h>
46: #endif
47:
48: #ifndef MAIN
49: #define EXTERN extern
50: #else
51: #define EXTERN
52: #endif
53:
54: #define DEBUG
55:
56: #ifndef NULL
57: #define NULL 0
58: #endif
59:
60: #define LOAD_DIR "..\\DATA\\"
61:
62: #define SP_CHAR_NO(SP)              ((SP) -> char_no)
63: #define SP_SP_PTR(SP)               ((SP) -> spr)
64: #define SP_POS_X(SP)                ((SP) -> spr -> sp_x)
65: #define SP_POS_Y(SP)                ((SP) -> spr -> sp_y)
66: #define SP_CODE(SP)                 (((SP) -> spr -> sp_ctrl).sp.sp_code)
67: #define SP_PALET(SP)                (((SP) -> spr -> sp_ctrl).sp.color)
68: #define SP_PRIORITY(SP)             (((SP) -> spr -> sp_pwr).sp.pwr)
69: #define SP_DEFINED(SP)              ((SP) -> pcg_defined)
70: #define SP_PCG_DATA_PTR(SP)         (&((SP) -> pcg_data[0]))
71: #define SP_ANIME(SP)                ((SP) -> anime)
72: #define SP_PCG_NO(SP,POS)           ((SP) -> pcg_no[POS])
73: #define SP_PCG_DATA(SP,POS)         ((SP) -> pcg_data[POS])
74: #define SP_H_INV(SP)                (((SP) -> spr -> sp_ctrl).sp.h_invert)
75: #define SP_V_INV(SP)                (((SP) -> spr -> sp_ctrl).sp.v_invert)
76:
77: #define SPD_BODY(SP,PART_X,PART_Y)  ((SP) -> body[(PART_X)][(PART_Y)])
78: #define SPD_TYPE(SP)                ((SP) -> type)
79: #define SPD_MOVE(SP)                (((SP) -> def_move).move_array)
80: #define SPD_MOVE_FUNC(SP)           (((SP) -> def_move).move_func)
81: #define SPD_LIFE(SP)                ((SP) -> life)
82: #define SPD_REGIST(SP)              ((SP) -> registance)
83: #define SPD_ANIME_PTR(SP)           ((SP) -> anime_def)
84: #define SPD_WORK0(SP)               ((SP) -> work0)
85: #define SPD_WORK1(SP)               ((SP) -> work1)
86: #define SPD_WORK2(SP)               ((SP) -> work2)
87: #define SPD_WORK3(SP)               ((SP) -> work3)
88: #define SPD_POS_X(SP)               SP_POS_X(SPD_BODY((SP), 0, 0))
```

```
 89: #define SPD_POS_Y(SP)                SP_POS_Y(SPD_BODY((SP), 0, 0))
 90:
 91: #define MOVE_X(MOV,CL)               ((MOV[CL]).dif_x)
 92: #define MOVE_Y(MOV,CL)               ((MOV[CL]).dif_y)
 93:
 94: #define  NO_USE 0
 95: #define  DISP   1
 96: #define  USE    2
 97:
 98: #define 絶対値(X,Y) ((X) - (Y) >= 0 ? (X) - (Y) :  -((X) - (Y)))
 99: #define 表示   1
100: #define 非表示 0
101: #define MAX_NUM 118
102: #define 最終面  0
103:
104: #define PAL_C_MIN 160
105: #define PAL_C_MAX 175
106:
107: #define H_BLOCK_NUM 18
108:
109: /* 機種, コンパイラに完全に依存する記述 */
110: /* PCG レジスタの構造体 */
111: typedef  union  {
112:     unsigned short dummy;
113:     struct {
114:       unsigned pos0: 4;
115:       unsigned pos1: 4;
116:       unsigned pos2: 4;
117:       unsigned pos3: 4;
118:     } sp;
119: } SP_REG_U;
120:
121: typedef SP_REG_U  PCG[64];
122:
123: typedef struct {
124:     unsigned v_invert : 1;
125:     unsigned h_invert : 1;
126:     unsigned dummy    : 2;
127:     unsigned color    : 4;
128:     unsigned sp_code  : 8;
129: } BG_REG;
130:
131: /* スプライトスクロールレジスタ構造体 */
132: typedef struct {
133:   short int sp_x;
134:   short int sp_y;
```

```
135:   union {
136:     short int dummy;
137:     BG_REG sp;
138:   } sp_ctrl;
139:   union {
140:     short int dummy;
141:     struct {
142:       unsigned dummy   : 13;
143:       unsigned ext     :  1;
144:       unsigned pwr     :  2;
145:     } sp;
146:   } sp_pwr;
147: } SP_CTRL;
148:
149: /* スプライトデータ構造体 */
150:
151:
152: typedef struct {
153:   SP_CTRL *spr;              /* スプライトスクロールレジスタ   */
154:   short   char_no;           /* キャラクタナンバ */
155:   short   pcg_defined;       /* PCGがすでに定義されている場合は1 */
156:   short   anime;             /* アニメーションすれば anime > 0 */
157:   short   pcg_no[8];         /* アニメ用PCG定義番号 */
158:   PCG     *pcg_data[8];      /* アニメ4パターン, PCGのデータへのポインタ */
159: } SPRITE;
160:
161: typedef SPRITE *sprite;
162:
163: enum sp_type {
164:   DOT16,
165:   DOT32_1_2,
166:   DOT32_2_1,
167:   DOT32_2_2,
168:   DOT48_1_3,
169:   DOT48_2_3,
170:   DOT48_3_3,
171:   DOT48_3_1,
172:   DOT48_3_2,
173: };
174:
175: typedef struct {
176:   short dif_x;
177:   short dif_y;
178: } move;
179:
180: typedef union {
```

```
181:   move *move_array;
182:   int (*move_func)();
183: } move_def;
184:
185: typedef struct SP_DATA {
186:   enum sp_type type;
187:   short life;
188:   short registance;
189:   short work0;
190:   short work1;
191:   short work2;
192:   short work3;
193:   sprite body[3][3];
194:   move_def   def_move;
195:   short  *anime_def;
196: } *Sprite;
197:
198: /* スクロールレジスタと同等の構造体 */
199: typedef struct {
200:   short sc0_x_reg;
201:   short sc0_y_reg;
202:   short sc1_x_reg;
203:   short sc1_y_reg;
204:   short sc2_x_reg;
205:   short sc2_y_reg;
206:   short sc3_x_reg;
207:   short sc3_y_reg;
208: } CRTC_REG;
209:
210: EXTERN CRTC_REG scroll_data
211: #ifdef MAIN
212:  =
213: {
214:     ((H_BLOCK_NUM * 16) % 256) / 2,
215:     0,
216:     ((H_BLOCK_NUM * 16) % 256) / 2,
217:     0,
218:     ((H_BLOCK_NUM * 16) % 256) / 2,
219:     0,
220:     ((H_BLOCK_NUM * 16) % 256) / 2,
221:     0
222: };
223: #else
224: ;
225: #endif
226:
```

```
227: EXTERN volatile int vsync_counter;
228: EXTERN volatile int sp_is_ready;
229: EXTERN volatile sprite request_def;
230: EXTERN volatile int palet_def;
231: EXTERN unsigned short palet_buf[256];
232: EXTERN BG_REG bg_array[32];
233:
234: EXTERN int use_half_tone;
235: EXTERN unsigned short half_def_data;
236:
237: EXTERN jmp_buf err_buf;
238:
239: typedef struct
240: {
241:   short num_p;
242:   PCG **pcg_ptr_ptr;
243: } GAME_PCG;
244:
245: EXTERN struct load_file
246: {
247:   GAME_PCG *pcg;
248:   char     *file_name;
249:   char     *move_file_name;
250:   short    *moves;
251: } load_files[]
252: #ifdef MAIN
253:  =
254: {
255:   { NULL, "自機.PCG", NULL, NULL},
256:   { NULL, "蓄積.PCG", NULL, NULL},
257:   { NULL, "弾.PCG", NULL, NULL},
258:   { NULL, "風船.PCG", NULL, NULL},
259:   { NULL, "変形ミサイル.PCG", NULL, NULL},
260:   { NULL, "爆発0.PCG", NULL, NULL},
261:   { NULL, "ゲージ.PCG",NULL,NULL},
262:   { NULL, "数字.PCG", NULL, NULL},
263: };
264: #else
265: ;
266: #endif
267:
268: enum キャラタイプ
269: {
270:   C_自機,
271:   C_蓄積,
272:   C_弾,
```

```
273:   C_敵0,
274:   C_敵1,
275:   C_爆発0,
276:   C_ゲージ,
277:   C_数字,
278: };
279:
280: enum 移動方向
281: {
282:   静止,
283:   右,
284:   左,
285:   上,
286:   下,
287:   左上,
288:   左下,
289:   右上,
290:   右下
291: };
292:
293:
294: /* BG 画面用ダミー */
295: EXTERN Sprite ゲージ;
296: EXTERN Sprite 数字;
297:
298: /* ユーザ操作の自機 */
299: EXTERN Sprite 自機;
300: EXTERN Sprite 蓄積sp;
301: EXTERN Sprite やられ自機;
302:
303: /* 敵キャラクタ */
304: EXTERN Sprite 敵[MAX_NUM];
305: EXTERN Sprite タイプA;
306: EXTERN Sprite タイプB;
307:
308: /* 画面に存在する弾 */
309: #define MAX_BOM 12
310: EXTERN Sprite 弾[MAX_BOM];
311: EXTERN Sprite 弾ダミー;
312:
313: /* 爆発パターン */
314: EXTERN Sprite 爆発0ダミー;
315: EXTERN Sprite 爆発[MAX_NUM];
316:
317: /* 現在の敵キャラ数 */
318: EXTERN int 敵数;
```

```
319:
320: /* 現在の弾キャラ数 */
321: EXTERN int 弾数;
322:
323: /* 現在の爆発数 */
324: EXTERN int 爆発数;
325:
326: /* 経過時間カウンタ */
327: EXTERN int 経過時間;
328: EXTERN int ランク
329: #ifdef MAIN
330:  = 30;
331: #else
332: ;
333: #endif
334:
335: /* 自機が弾を発射するためのフラグ */
336: EXTERN int 弾発射;
337:
338: /* スコアカウンタ */
339: EXTERN int スコア;
340: EXTERN int 蓄積;
341: EXTERN int 終;
342: EXTERN int ゲーム開始;
343: EXTERN int 面クリア;
344:
345: EXTERN int 自機スピード
346: #ifdef MAIN
347: = 2;
348: #else
349: ;
350: #endif
351:
352: /* グラフィックパレット */
353: unsigned int g_palet[256];
354: unsigned short c_palet[PAL_C_MAX - PAL_C_MIN];
355:
356:
357: /* マップグラフィックへのポインタ */
358: unsigned short *map_data[256];
359:
360:
361: /* sprite.c */
362: void display_sprite (Sprite,int);
363: void delete_sprite (Sprite);
364: Sprite dup_sprite (Sprite);
```

```
365: Sprite def_sprite (enum sp_type, PCG ***, int);
366: Sprite def_sprite_dummy (enum sp_type, PCG ***, int);
367: void move_sprite_diff (Sprite, int, int);
368: void move_sprite_abs (Sprite, int, int);
369: void select_sprite_pcg (Sprite, int);
370: void select_sprite_color (Sprite, int);
371: void select_sprite_h_invert (Sprite, int);
372: void select_sprite_v_invert (Sprite, int);
373: /*割り込み処理関数 */
374: void vsync_disp (void);
375:
376: /* utiles.c */
377: volatile void game_abort (char *);
378: void *xmalloc (int);
379: void ロードデータ (void);
380: void 画面初期化 (void);
381: void 画面終了処理 (void);
382: void init_trap14 (void);
383: void 消去記録 (Sprite, int);
384: void 消去実行 (void);
385:
386: /* clash.c */
387: void 当たり判定 (void);
388:
389: /* enemy.c */
390: void 敵キャラ発生移動 (void);
391:
392: /* mychr.c */
393: void 自機移動処理 (void);
394: void 弾発射移動 (void);
395:
396: /* map.c */
397: void ロードマップ (int);
398: void 背景移動処理 (void);
399: void raster_scroll (void);
400:
401: /* back.c */
402: void BGデータ初期化 (void);
403: void スコア表示 (void);
404: void ゲージ表示 (void);
405:
406: /* game.c */
```

▶4行〜6行

『付録ディスク』の**LIBC**環境でなく，**XC Ver.1.0**および**XC Ver.2.0**環境でコンパイルをする場合には，この**#define**を有効にしてください。なお，**XC Ver.2.0**のSCSI関係を拡張したiocslib.hを**GCC**で使うと，エラーになる部分があります。エラーになるのは，構造体の一部で，配列の大きさを記述してない部分です。この部分は，配列の大きさを次のように0と記述しておけば，**XC**と**GCC**で互換になります。

```
#ifdef __GNUC__
   size_zero_array[0];
#else  /* XC or MS-C */
   size_zero_array[];
#endif
```

size_zero_arrayは単なるサンプルです。実際のヘッダがこのように記述されているわけではないので，念のため。

▶8行〜10行

XCのヘッダには，一部，実際の関数の引数の数とプロトタイプ宣言の引数の数が一致していないバグがあります。そのバグのためにエラーになった場合には，この**#define**を有効にしてください。本来はそのバグがあるヘッダを書き換えておくべきです。

▶12行〜14行

このゲームでは，いわゆる溜め撃ちが可能になっています。その溜め撃ちを行った場合に，敵のキャラクタとの当たり判定で衝突するごとに一定量だけ弾の威力を削減する機能を使う際には，この**#define**を有効にしてください。現在は，R–TYPE(アイレム)のように，弾の威力以下の敵に当たった場合は，ノーダメージで貫通するような当たり判定になっています。

▶16行〜18行

背景のグラフィックマップデータをファイルから読み取る場合には，この**#define**を有効にしてください。現在は，map.cから**#include**で固定マップデータを配列としてプログラム内部に保持しています。当然のことですが，マップデータは自前で作成する必要があります。詳しくは，4.9.1「グラフィック画面の処理」(P.223)を読んでください。

▸20 行〜22 行

背景のグラフィック描画を行うには，CPU が直接転送する方法と，DMA 転送を行う方法とがあります。CPU が直接転送を行う場合には，この**#define** を有効にしてください。DMA 転送についてはP.225を参照してください。

▸26 行〜46 行

一連のシステムヘッダの**#include** です。**LIBC** 環境では，**GCC** に`-Wall`を指定して警告レベルを上昇させても，警告は出ないようになっています。

▸48 行〜52 行

プログラム全体にみえる変数を，全体で一度だけ定義するために行う処理のマクロです(2.3.2「変数のスコープ」P.37参照)。game.c では先頭に`#define MAIN`が存在するので，`EXTERN`のついた変数はすべて定義になり，それ以外のファイルでは変数はすべて宣言になります。

▸54 行

デバッグを行うための処理を追加削除するマクロです。

▸56 行〜58 行

P.68で説明した「何も指しているものがない」ポインタ値 0 を定義するマクロです。よく勘違いされるのが，Fig. 2.4(P.65) で説明した文字列のいちばん最後に付加されるヌル文字 (値が 0 の文字コード) と，この`NULL`です。`NULL`は，普通ポインタ値に用いられ，ヌル文字には使われません。コンパイラにとっては，両方とも値が 0 なので同じ意味なのですが，文字配列と文字へのポインタ配列の混同が，この`NULL`のために多数起きるのも事実です。

普通，この`NULL`はシステムヘッダで**#define** されていますが，保険的な意味でしばしばこのように`#ifndef NULL`を用いて**#define** されます。

▸60 行

ゲームで使うファイルをオープンするためのディレクトリパスを**#define** しておきます。ここを変更することで，ファイルをオープンするディレクトリを変更できます。

▸62 行〜96 行

スプライトを管理するためのデータを扱うためのマクロ群です。詳しい内容は Table 4.3 (P.187) を見てください。

▸98 行〜107 行

ゲームの各種ユーティリティマクロで (Table 4.2 参照)。漢字を用いたマクロは，完全に **X68000/X68030 版 GCC** に依存しており，他のコンパイラではまず使えません。

●Table 4.2　ユーティリティのマクロ

絶対値()	当たり判定に使うマクロ。
表示 非表示	前述のNO_USE, DISP, USEとは異なったスプライトの状態を表すマクロ。
MAX_NUM	画面内に現れる最大の敵キャラクタの数です。
最終面	いちおう面クリアタイプなので，最終面の番号を決めている。1 面しかないので，0 になっている。
PAL_C_MIN PAL_C_MAX	パレットを使ったアニメーションで変更するパレット番号の最小番号と最大番号。
H_BLOCK_NUM	画面横方向のマップブロックの数。ラスタースクロールを使用。このため，実際に表示されるブロック数より左右に 1 ブロック大きくなっているので，18 になる。

▸109 行〜196 行

スプライトを管理する関数群で扱うデータ構造の宣言部分です。詳しい内容は List 4.3(P.163) を参照してください。

▸198 行〜225 行

グラフィック画面スクロールのためのデータ構造宣言です。グラフィックスクロールレジスタの構造については，すでに Fig. 3.6(P.80) で扱っています。

▸227 行〜358 行

プログラム全体でみえる変数 (グローバル変数) の宣言が並んでいます。各変数の役割は以下のようになります。

`EXTERN volatile int vsync_counter;`

垂直帰線期間割り込み処理で，割り込み処理を 1 回行うごとに 1 増えるカウンタです。ゲーム全体の時間進行を計るためのカウンタとして利用されます。

`EXTERN volatile int sp_is_ready;`

垂直帰線期間割り込み処理と通常のゲーム処理との同期をとるためのフラグです。通常処理で割り込み処理のためのデータ処理が終わったら '1' をセットして，垂直帰線期間割り込み処理でフラグを '0' にします。

`EXTERN volatile sprite request_def;`

垂直帰線期間割り込みに PCG 設定を行わせるためのフラグを兼ねた，スプライト構造体ポインタです。NULLでないときに PCG 設定の要求があったことを示して，垂直帰線期間割り込みで PCG への転送が行われるとNULLが設定されます。

`EXTERN volatile int palet_def;`

グラフィック画面のパレット設定要求があった場合に，'1' がセットされます。このフラグがセットされると，割り込み処理はグラフィックのパレットを設定し，このフラグを '0' にします。

これらの変数はすべて割り込みで操作しますから，**volatile** 宣言をしておきます。P.111で説明したように，この宣言がない場合には，**GCC** の最適化によってプログラムが予期しない無限ループに陥ることがあります。

`EXTERN unsigned short palet_buf[256];`

グラフィック画面のパレットデータを格納するための配列です。

`EXTERN BG_REG bg_array[32];`

バックグランド画面表示用の BG 構造体の配列です。

`EXTERN int use_half_tone;`

半透明機能や特殊プライオリティを使うときには '1' を設定します。半透明や特殊プライオリティは，'1' を設定した後の垂直帰線期間割り込みで，次のhalf_def_dataをハードウェアに設定することによって実行されます。

`EXTERN jmp_buf err_buf;`

エラーハンドラsetjmp，longjmpのための変数です。

`GAME_PCG`

PCG データを管理するための構造体名です。構造体メンバnum_pには，次の PCG データポインタの数が格納されます。PCG **pcg_ptr_ptrには，PCG データへのポインタのポインタが格納され，その数がnum_pです。

`struct load_file`

ファイルから PCG データ等を読み込むための構造体です。

`enum キャラタイプ;`

`enum 移動方向;`

enum は本書初登場です。**enum** は「列挙」とも呼ばれ，データがある特定の種類に分類されるときに，普通に番号を割り付ける代わりに，もっとわかりやすい抽象的な名称を割り振る際に用います。

たとえば,

赤色 → 0
青色 → 1
緑色 → 2

のように番号を割り当てるとします。赤は'0'になるのですが，このままだと数値の0なのか，色を表す0なのかがまったく区別できません。そこで,

```
enum COLOR {
    red,    /* 赤 */
    blue,   /* 青 */
    green   /* 緑 */
};
```

のように,「列挙COLOR」を宣言します。これによって0, 1, 2の代わりにred, blue, greenを使うことができるようになります。この機能はPascalにもありますが，Cの場合，この「列挙」はきわめて貧弱な扱いになっており，列挙を代入する変数に変な値を入れても，エラーにも警告にもなりません。**enum**は先頭から整数の値で0, 1, 2･･･と割り当てられていきます。明示的に値を設定したい場合は,

```
enum COLOR {
   red,
   blue = 6,
   green,
   yellow = 12
};
```

のように宣言します。この場合は

red → 0
blue → 6
green → 7
yellow → 12

となります。

List 4.2

```
1:
2: enum COLOR {
3:   red,
4:   blue,
5:   green
6: };
```

```
 7:
 8: enum COLOR x;
 9:
10: foo0()
11: {
12:   x = red;
13: }
14:
15: foo1()
16: {
17:   x = 10;
18: }
```

List 4.2 を見てください。12 行目でenum COLOR xにredを代入しています。これは不思議ではありませんが，17 行目では整数 10 を代入しています。にもかかわらず，このプログラムはエラーにも警告にもなりません。**enum** と宣言された変数は，実はコンパイラにとってはただの整数の変数です。このように貧弱な C の **enum** は，しばしば批判の対象とされますが，プログラムの読みやすさに貢献するだけのものと割り切って使えばよいでしょう。

```
EXTERN Sprite ゲージ;
EXTERN Sprite 数字;
```

バックグランドの表示を行うための，PCG データを保持するためのワーク変数です。

```
EXTERN Sprite 自機;
EXTERN Sprite 蓄積 sp;
EXTERN Sprite やられ自機;
```

プレイヤーが操る自機関係のスプライト表示を行うためのワーク変数です。

```
EXTERN Sprite 敵 [MAX_NUM];
EXTERN Sprite タイプ A;
EXTERN Sprite タイプ B;
```

敵キャラクタのスプライト表示用のワーク変数です。

```
#define MAX_BOM 12
EXTERN Sprite 弾 [MAX_BOM];
EXTERN Sprite 弾ダミー;
```

自機が発射する弾のスプライト表示用ワーク変数です。MAX_BOMは画面に弾が同時に存在できる最大数を定義します。

```
EXTERN Sprite 爆発 0 ダミー;
EXTERN Sprite 爆発 [MAX_NUM];
```

キャラクタが破壊されたときに現れる，爆発パターンを表示するためのスプライ

トワーク変数です。

```
EXTERN int 敵数;
```

```
EXTERN int 弾数;
```

```
EXTERN int 爆発数;
```

これらは現在表示を行っているスプライトの数を管理するためのカウンタです。

```
EXTERN int 経過時間;
```

ゲーム開始以降の時間を計る時間計測用のカウンタです。この変数とスコアで，ランク (ゲームの難易度) を変化させます。

```
EXTERN int ランク;
```

ゲームの難易度を変化させるための変数です。値が大きくなるほど，敵キャラクタの時間あたりの発生頻度が大きくなります。

```
EXTERN int 弾発射;
```

ジョイスティックの状態を調べて，弾を発射する操作が行われた場合に 1 になるフラグです。

```
EXTERN int スコア;
```

ゲームの得点カウンタです。

```
EXTERN int 蓄積;
```

ジョイスティックのトリガボタンを押している時間を計測するためのカウンタです。

```
EXTERN int 終;
```

ゲームが最終面までプレイされたら 1 になる，終了指示フラグです。

```
EXTERN int ゲーム開始;
```

各種初期設定が終了して，実際にスクロールや敵キャラクタの処理を開始する状態になったら 1 になるフラグです。

```
EXTERN int 面クリア;
```

体裁は面クリアタイプのゲームなので，ある面がクリアされたら 1 になるフラグが用意されています。

```
EXTERN int 自機スピード;
```

自機の移動速度を決定するワーク変数です。実際には未使用になっています。

```
unsigned int g_palet[256];
```

```
unsigned short c_palet[PAL_C_MAX - PAL_C_MIN];
```

パレットの初期値設定用のデータバッファ，パレットアニメーションのためのパレットバッファです。

```
unsigned short *map_data[256];
```

グラフィック画面に書き込む，16×16 ドットのパターンデータ保持用ワーク変数へのポインタ配列です。

ヘッダの残りの部分はすべて関数のプロトタイプ宣言です。これらは各ソースごとに分割して宣言を行っています。こうしておけば，コンパイラが誤った関数の呼び出しを指摘

するので，安全性が高くなります。こういった開発方法を使うには，マルチバッファでそれを同時に表示編集できる機能をもったエディタが必須です。

■ 4.4 スプライト管理部分の作成

まず，スプライト全体を管理する部分を作成します。この部分はできるだけハードウェアを覆い隠すように制作しておけば，移植性が低いゲームプログラムでも，多少は他の機種への移植が楽になるでしょう。

List 4.3 sprite.c

```
 1:
 2: #include "game.h"
 3:
 4: static SP_CTRL  disp_sp[128];
 5: static char this_no_is_use[128];
 6: static char pcg_is_use[128];
 7:
 8: static void
 9: active_sprite (sprite sp)
10: {
11:   register int ch_no = SP_CHAR_NO (sp);
12:   this_no_is_use[ch_no] = DISP;
13:   SP_PRIORITY (sp) = 1;
14: }
15:
16: static void
17: suspend_sprite (sprite sp)
18: {
19:   register int ch_no = SP_CHAR_NO (sp);
20:   this_no_is_use[ch_no] = USE;
21:   SP_PRIORITY (sp) = 0;
22: }
23:
24: static void
25: free_sprite (sprite sp)
26: {
27:   int c_no = SP_CHAR_NO (sp);
28:   int anime = SP_ANIME (sp);
29:   if (c_no < 128)
30:     {
31:       /* スクロールレジスタ割り当てを解除 */
32:       this_no_is_use[c_no] = NO_USE;
33:
```

```
34:       /* PCGの使用カウントを減らす */
35:       do {
36:         pcg_is_use[SP_PCG_NO (sp, anime)] --;
37:       } while (--anime >= 0);
38:       free (sp);
39:     }
40:   else if (c_no == 128)
41:     {
42:       /* PCGの使用カウントを減らす */
43:       do {
44:         pcg_is_use[SP_PCG_NO (sp, anime)] --;
45:       } while (--anime >= 0);
46:       free (SP_SP_PTR (sp));
47:       free (sp);
48:     }
49:   else
50:    game_abort ("free_sprite:スクロールレジスタ番号異常");
51: }
52:
53: static sprite
54: alloc_sprite (void)
55: {
56:   register int c_no;
57:   /* 空きのスクロールレジスタを捜す */
58:   for (c_no = 0; c_no < 128; c_no ++)
59:     if (!this_no_is_use[c_no])
60:       break;
61:   if (c_no < 128)
62:     {
63:       /* データを作成する */
64:       sprite sp = (sprite) xmalloc (sizeof (SPRITE));
65:       PCG **pcg;
66:
67:       /* 各種ポインタの初期化 */
68:       SP_SP_PTR (sp) = &disp_sp[c_no];
69:       SP_CHAR_NO(sp) = c_no;
70:       pcg = SP_PCG_DATA_PTR(sp);
71:       *pcg ++ = 0;
72:       *pcg ++ = 0;
73:       *pcg ++ = 0;
74:       *pcg ++ = 0;
75:       *pcg ++ = 0;
76:       *pcg ++ = 0;
77:       *pcg ++ = 0;
78:       *pcg ++ = 0;
79:       /* 割り当ては行うが表示はしない */
```

```
 80:       suspend_sprite (sp);
 81:       return sp;
 82:     }
 83:   else
 84:     game_abort ("alloc_sprite:定義キャラクタ過剰");
 85: }
 86:
 87: static sprite
 88: alloc_sprite_dummy (void)
 89: {
 90:   /* データを作成する */
 91:   sprite sp = (sprite) xmalloc (sizeof (SPRITE));
 92:   PCG **pcg;
 93:
 94:   /* 各種ポインタの初期化 */
 95:   SP_SP_PTR (sp) = xmalloc (sizeof (SP_CTRL));
 96:   SP_CHAR_NO(sp) = 128;
 97:
 98:   pcg = SP_PCG_DATA_PTR(sp);
 99:   *pcg ++ = 0;
100:   *pcg ++ = 0;
101:   *pcg ++ = 0;
102:   *pcg ++ = 0;
103:   *pcg ++ = 0;
104:   *pcg ++ = 0;
105:   *pcg ++ = 0;
106:   *pcg ++ = 0;
107:   return sp;
108: }
109:
110: static sprite
111: make_sprite (int anime, PCG **pcg_data, int sc_alloc)
112: {
113:   register PCG **pcg;
114:   register sprite sp = sc_alloc ? alloc_sprite () : alloc_sprite_dummy ();
115:   SP_ANIME(sp) = anime;
116:   pcg = SP_PCG_DATA_PTR (sp);
117:   do {
118:     *pcg ++ = *pcg_data++;
119:   } while (--anime >= 0);
120:   SP_DEFINED (sp) = 0;
121:   return sp;
122: }
123:
124: static sprite
125: copy_sprite (sprite sp)
```

```
126: {
127:   register int anime;
128:   register sprite ret = alloc_sprite ();
129:   anime = SP_ANIME (ret) = SP_ANIME (sp);
130:
131:   if (!SP_DEFINED (sp))
132:     {
133:       request_def = sp;
134:       while (request_def)
135:         ;
136:     }
137:   SP_DEFINED (ret) = 1;
138:   do {
139:     int pcg_no;
140:     pcg_no = SP_PCG_NO (ret, anime) = SP_PCG_NO (sp, anime);
141:     SP_PCG_DATA (ret, anime) = SP_PCG_DATA (sp, anime);
142:     pcg_is_use[pcg_no] ++;
143:   } while (--anime >= 0);
144:   return ret;
145: }
146:
147: void
148: display_sprite (Sprite sp, int sw)
149: {
150:    int x,y;
151:    int end_x,end_y;
152:    void (*func)(sprite) = sw ? active_sprite : suspend_sprite;
153:
154:    switch (SPD_TYPE (sp))
155:      {
156:      case DOT16:
157:        end_x = 1;
158:        end_y = 1;
159:        break;
160:      case DOT32_1_2:
161:        end_x = 1;
162:        end_y = 2;
163:        break;
164:      case DOT32_2_1:
165:        end_x = 2;
166:        end_y = 1;
167:        break;
168:      case DOT32_2_2:
169:        end_x = 2;
170:        end_y = 2;
171:        break;
```

```
172:      case DOT48_1_3:
173:        end_x = 1;
174:        end_y = 3;
175:        break;
176:      case DOT48_2_3:
177:        end_x = 2;
178:        end_y = 3;
179:        break;
180:      case DOT48_3_3:
181:        end_x = 3;
182:        end_y = 3;
183:        break;
184:      case DOT48_3_1:
185:        end_x = 3;
186:        end_y = 1;
187:        break;
188:      case DOT48_3_2:
189:        end_x = 3;
190:        end_y = 2;
191:        break;
192:      default:
193:        game_abort ("delete_sprite:未知のスプライトタイプ");
194:        break;
195:      }
196:
197:    for (y = 0; y < end_y; y++)
198:      for (x = 0; x < end_x; x++)
199:        func (SPD_BODY (sp, x, y));
200: }
201:
202:
203: void
204: delete_sprite (Sprite sp)
205: {
206:    int x,y;
207:    int end_x,end_y;
208:    switch (SPD_TYPE (sp))
209:      {
210:      case DOT16:
211:        end_x = 1;
212:        end_y = 1;
213:        break;
214:      case DOT32_1_2:
215:        end_x = 1;
216:        end_y = 2;
217:        break;
```

```
218:      case DOT32_2_1:
219:        end_x = 2;
220:        end_y = 1;
221:        break;
222:      case DOT32_2_2:
223:        end_x = 2;
224:        end_y = 2;
225:        break;
226:      case DOT48_1_3:
227:        end_x = 1;
228:        end_y = 3;
229:        break;
230:      case DOT48_2_3:
231:        end_x = 2;
232:        end_y = 3;
233:        break;
234:      case DOT48_3_3:
235:        end_x = 3;
236:        end_y = 3;
237:        break;
238:      case DOT48_3_1:
239:        end_x = 3;
240:        end_y = 1;
241:        break;
242:      case DOT48_3_2:
243:        end_x = 3;
244:        end_y = 2;
245:        break;
246:      default:
247:        game_abort ("delete_sprite:未知のスプライトタイプ");
248:        break;
249:      }
250:
251:    for (y = 0; y < end_y; y++)
252:      for (x = 0; x < end_x; x++)
253:        free_sprite (SPD_BODY (sp, x, y));
254: }
255:
256: Sprite
257: dup_sprite (Sprite sp)
258: {
259:   int x,y;
260:   int end_x,end_y;
261:   Sprite ret_val;
262:
263:   switch (SPD_TYPE (sp))
```

```
264:      {
265:      case DOT16:
266:        end_x = 1;
267:        end_y = 1;
268:        break;
269:      case DOT32_1_2:
270:        end_x = 1;
271:        end_y = 2;
272:        break;
273:      case DOT32_2_1:
274:        end_x = 2;
275:        end_y = 1;
276:        break;
277:      case DOT32_2_2:
278:        end_x = 2;
279:        end_y = 2;
280:        break;
281:      case DOT48_1_3:
282:        end_x = 1;
283:        end_y = 3;
284:        break;
285:      case DOT48_2_3:
286:        end_x = 2;
287:        end_y = 3;
288:        break;
289:      case DOT48_3_3:
290:        end_x = 3;
291:        end_y = 3;
292:        break;
293:      case DOT48_3_1:
294:        end_x = 3;
295:        end_y = 1;
296:        break;
297:      case DOT48_3_2:
298:        end_x = 3;
299:        end_y = 2;
300:        break;
301:      default:
302:        game_abort ("def_sprite_file:未知のスプライトタイプ");
303:        break;
304:      }
305:
306:    ret_val = (Sprite) xmalloc (sizeof (struct SP_DATA));
307:
308:    SPD_TYPE (ret_val) = SPD_TYPE (sp);
309:    SPD_MOVE_FUNC (ret_val) = 0;
```

```
310:    SPD_LIFE (ret_val) = 0;
311:    SPD_REGIST (ret_val) = 0;
312:    SPD_ANIME_PTR (ret_val) = 0;
313:
314:    for (y = 0; y < 3; y++)
315:      for (x = 0; x < 3; x++)
316:        SPD_BODY (ret_val, x, y) = 0;
317:
318:    for (y = 0; y < end_y; y++)
319:      for (x = 0; x < end_x; x++)
320:        SPD_BODY (ret_val, x, y) = copy_sprite (SPD_BODY (sp, x, y));
321:
322:    return ret_val;
323: }
324:
325:
326: Sprite
327: def_sprite (enum sp_type type, PCG ***pcg, int anime)
328: {
329:   int x,y;
330:   int end_x,end_y;
331:   Sprite ret_val;
332:   int pcg_index;
333:
334:   switch (type)
335:      {
336:      case DOT16:
337:        end_x = 1;
338:        end_y = 1;
339:        break;
340:      case DOT32_1_2:
341:        end_x = 1;
342:        end_y = 2;
343:        break;
344:      case DOT32_2_1:
345:        end_x = 2;
346:        end_y = 1;
347:        break;
348:      case DOT32_2_2:
349:        end_x = 2;
350:        end_y = 2;
351:        break;
352:      case DOT48_1_3:
353:        end_x = 1;
354:        end_y = 3;
355:        break;
```

```
356:      case DOT48_2_3:
357:        end_x = 2;
358:        end_y = 3;
359:        break;
360:      case DOT48_3_3:
361:        end_x = 3;
362:        end_y = 3;
363:        break;
364:      case DOT48_3_1:
365:        end_x = 3;
366:        end_y = 1;
367:        break;
368:      case DOT48_3_2:
369:        end_x = 3;
370:        end_y = 2;
371:        break;
372:      default:
373:        game_abort ("def_sprite:未知のスプライトタイプ");
374:        break;
375:      }
376:
377:    ret_val = (Sprite) xmalloc (sizeof (struct SP_DATA));
378:
379:    SPD_TYPE (ret_val) = type;
380:    SPD_MOVE_FUNC (ret_val) = 0;
381:    SPD_LIFE (ret_val) = 0;
382:    SPD_REGIST (ret_val) = 0;
383:    SPD_ANIME_PTR (ret_val) = 0;
384:
385:    for (y = 0; y < 3; y++)
386:      for (x = 0; x < 3; x++)
387:        SPD_BODY (ret_val, x, y) = 0;
388:    pcg_index = 0;
389:    for (y = 0; y < end_y; y++)
390:      for (x = 0; x < end_x; x++)
391:        SPD_BODY (ret_val, x, y) = make_sprite (anime, pcg[pcg_index ++], 1);
392:
393:    return ret_val;
394: }
395:
396: Sprite
397: def_sprite_dummy (enum sp_type type, PCG ***pcg, int anime)
398: {
399:   int x,y;
400:   int end_x,end_y;
401:   Sprite ret_val;
```

```
402:   int pcg_index;
403:
404:   switch (type)
405:      {
406:      case DOT16:
407:        end_x = 1;
408:        end_y = 1;
409:        break;
410:      case DOT32_1_2:
411:        end_x = 1;
412:        end_y = 2;
413:        break;
414:      case DOT32_2_1:
415:        end_x = 2;
416:        end_y = 1;
417:        break;
418:      case DOT32_2_2:
419:        end_x = 2;
420:        end_y = 2;
421:        break;
422:      case DOT48_1_3:
423:        end_x = 1;
424:        end_y = 3;
425:        break;
426:      case DOT48_2_3:
427:        end_x = 2;
428:        end_y = 3;
429:        break;
430:      case DOT48_3_3:
431:        end_x = 3;
432:        end_y = 3;
433:        break;
434:      case DOT48_3_1:
435:        end_x = 3;
436:        end_y = 1;
437:        break;
438:      case DOT48_3_2:
439:        end_x = 3;
440:        end_y = 2;
441:        break;
442:      default:
443:        game_abort ("def_sprite:未知のスプライトタイプ");
444:        break;
445:      }
446:
447:    ret_val = (Sprite) xmalloc (sizeof (struct SP_DATA));
```

```
448:
449:    SPD_TYPE (ret_val) = type;
450:    SPD_MOVE_FUNC (ret_val) = 0;
451:    SPD_LIFE (ret_val) = 0;
452:    SPD_REGIST (ret_val) = 0;
453:    SPD_ANIME_PTR (ret_val) = 0;
454:
455:    for (y = 0; y < 3; y++)
456:      for (x = 0; x < 3; x++)
457:        SPD_BODY (ret_val, x, y) = 0;
458:    pcg_index = 0;
459:    for (y = 0; y < end_y; y++)
460:      for (x = 0; x < end_x; x++)
461:        SPD_BODY (ret_val, x, y) = make_sprite (anime, pcg[pcg_index ++], 0);
462:
463:    return ret_val;
464: }
465:
466: void
467: move_sprite_diff (Sprite sp, int dx, int dy)
468: {
469:    int x,y;
470:    int end_x,end_y;
471:    switch (SPD_TYPE (sp))
472:      {
473:      case DOT16:
474:        end_x = 1;
475:        end_y = 1;
476:        break;
477:      case DOT32_1_2:
478:        end_x = 1;
479:        end_y = 2;
480:        break;
481:      case DOT32_2_1:
482:        end_x = 2;
483:        end_y = 1;
484:        break;
485:      case DOT32_2_2:
486:        end_x = 2;
487:        end_y = 2;
488:        break;
489:      case DOT48_1_3:
490:        end_x = 1;
491:        end_y = 3;
492:        break;
493:      case DOT48_2_3:
```

```
494:        end_x = 2;
495:        end_y = 3;
496:        break;
497:      case DOT48_3_3:
498:        end_x = 3;
499:        end_y = 3;
500:        break;
501:      case DOT48_3_1:
502:        end_x = 3;
503:        end_y = 1;
504:        break;
505:      case DOT48_3_2:
506:        end_x = 3;
507:        end_y = 2;
508:        break;
509:      default:
510:        game_abort ("move_sprite_dif:未知のスプライトタイプ");
511:        break;
512:      }
513:
514:    for (y = 0; y < end_y; y++)
515:      for (x = 0; x < end_x; x++)
516:        {
517:                  SP_POS_X (SPD_BODY (sp, x, y)) += dx;
518:                  SP_POS_Y (SPD_BODY (sp, x, y)) += dy;
519:        }
520: }
521:
522: void
523: move_sprite_abs (Sprite sp, int ax, int ay)
524: {
525:    int x,y;
526:    int end_x,end_y;
527:    switch (SPD_TYPE (sp))
528:      {
529:      case DOT16:
530:        end_x = 1;
531:        end_y = 1;
532:        break;
533:      case DOT32_1_2:
534:        end_x = 1;
535:        end_y = 2;
536:        break;
537:      case DOT32_2_1:
538:        end_x = 2;
539:        end_y = 1;
```

```
540:         break;
541:       case DOT32_2_2:
542:         end_x = 2;
543:         end_y = 2;
544:         break;
545:       case DOT48_1_3:
546:         end_x = 1;
547:         end_y = 3;
548:         break;
549:       case DOT48_2_3:
550:         end_x = 2;
551:         end_y = 3;
552:         break;
553:       case DOT48_3_3:
554:         end_x = 3;
555:         end_y = 3;
556:         break;
557:       case DOT48_3_1:
558:         end_x = 3;
559:         end_y = 1;
560:         break;
561:       case DOT48_3_2:
562:         end_x = 3;
563:         end_y = 2;
564:         break;
565:       default:
566:         game_abort ("move_sprite_dif:未知のスプライトタイプ");
567:         break;
568:       }
569:
570:     for (y = 0; y < end_y; y++)
571:       for (x = 0; x < end_x; x++)
572:         {
573:                   SP_POS_X (SPD_BODY (sp, x, y)) = ax + x * 16;
574:                   SP_POS_Y (SPD_BODY (sp, x, y)) = ay + y * 16;
575:         }
576: }
577:
578: void
579: select_sprite_pcg (Sprite sp, int sel)
580: {
581:     int x,y;
582:     int end_x,end_y;
583:     switch (SPD_TYPE (sp))
584:       {
585:       case DOT16:
```

```
586:        end_x = 1;
587:        end_y = 1;
588:        break;
589:      case DOT32_1_2:
590:        end_x = 1;
591:        end_y = 2;
592:        break;
593:      case DOT32_2_1:
594:        end_x = 2;
595:        end_y = 1;
596:        break;
597:      case DOT32_2_2:
598:        end_x = 2;
599:        end_y = 2;
600:        break;
601:      case DOT48_1_3:
602:        end_x = 1;
603:        end_y = 3;
604:        break;
605:      case DOT48_2_3:
606:        end_x = 2;
607:        end_y = 3;
608:        break;
609:      case DOT48_3_3:
610:        end_x = 3;
611:        end_y = 3;
612:        break;
613:      case DOT48_3_1:
614:        end_x = 3;
615:        end_y = 1;
616:        break;
617:      case DOT48_3_2:
618:        end_x = 3;
619:        end_y = 2;
620:        break;
621:      default:
622:        game_abort ("select_sprite_pcg:未知のスプライトタイプ");
623:        break;
624:      }
625:
626:    for (y = 0; y < end_y; y++)
627:      for (x = 0; x < end_x; x++)
628:        {
629:                if (SP_DEFINED (SPD_BODY (sp, x, y)))
630:                  SP_CODE (SPD_BODY (sp, x, y))
                        = SP_PCG_NO (SPD_BODY (sp, x, y),sel);
```

```
631:                 else
632:                   {
633:                     request_def = SPD_BODY (sp, x, y);
634:                     while (request_def)
635:                       ;
636:                     SP_CODE (SPD_BODY (sp, x, y))
                          = SP_PCG_NO (SPD_BODY (sp, x, y),sel);
637:                   }
638:         }
639: }
640:
641: void
642: select_sprite_color (Sprite sp, int col)
643: {
644:    int x,y;
645:    int end_x,end_y;
646:    switch (SPD_TYPE (sp))
647:      {
648:      case DOT16:
649:        end_x = 1;
650:        end_y = 1;
651:        break;
652:      case DOT32_1_2:
653:        end_x = 1;
654:        end_y = 2;
655:        break;
656:      case DOT32_2_1:
657:        end_x = 2;
658:        end_y = 1;
659:        break;
660:      case DOT32_2_2:
661:        end_x = 2;
662:        end_y = 2;
663:        break;
664:      case DOT48_1_3:
665:        end_x = 1;
666:        end_y = 3;
667:        break;
668:      case DOT48_2_3:
669:        end_x = 2;
670:        end_y = 3;
671:        break;
672:      case DOT48_3_3:
673:        end_x = 3;
674:        end_y = 3;
675:        break;
```

```
676:      case DOT48_3_1:
677:        end_x = 3;
678:        end_y = 1;
679:        break;
680:      case DOT48_3_2:
681:        end_x = 3;
682:        end_y = 2;
683:        break;
684:      default:
685:        game_abort ("select_sprite_color:未知のスプライトタイプ");
686:        break;
687:      }
688:
689:    for (y = 0; y < end_y; y++)
690:      for (x = 0; x < end_x; x++)
691:        SP_PALET (SPD_BODY (sp, x, y)) = col;
692: }
693:
694: void
695: select_sprite_h_invert (Sprite sp, int inv)
696: {
697:    int x,y;
698:    int end_x,end_y;
699:    switch (SPD_TYPE (sp))
700:      {
701:      case DOT16:
702:        end_x = 1;
703:        end_y = 1;
704:        break;
705:      case DOT32_1_2:
706:        end_x = 1;
707:        end_y = 2;
708:        break;
709:      case DOT32_2_1:
710:        end_x = 2;
711:        end_y = 1;
712:        break;
713:      case DOT32_2_2:
714:        end_x = 2;
715:        end_y = 2;
716:        break;
717:      case DOT48_1_3:
718:        end_x = 1;
719:        end_y = 3;
720:        break;
721:      case DOT48_2_3:
```

```
722:         end_x = 2;
723:         end_y = 3;
724:         break;
725:       case DOT48_3_3:
726:         end_x = 3;
727:         end_y = 3;
728:         break;
729:       case DOT48_3_1:
730:         end_x = 3;
731:         end_y = 1;
732:         break;
733:       case DOT48_3_2:
734:         end_x = 3;
735:         end_y = 2;
736:         break;
737:       default:
738:         game_abort ("select_sprite_color:未知のスプライトタイプ");
739:         break;
740:       }
741:
742:     for (y = 0; y < end_y; y++)
743:       for (x = 0; x < end_x; x++)
744:         SP_H_INV (SPD_BODY (sp, x, y)) = inv;
745: }
746:
747: void
748: select_sprite_v_invert (Sprite sp, int inv)
749: {
750:     int x,y;
751:     int end_x,end_y;
752:     switch (SPD_TYPE (sp))
753:       {
754:       case DOT16:
755:         end_x = 1;
756:         end_y = 1;
757:         break;
758:       case DOT32_1_2:
759:         end_x = 1;
760:         end_y = 2;
761:         break;
762:       case DOT32_2_1:
763:         end_x = 2;
764:         end_y = 1;
765:         break;
766:       case DOT32_2_2:
767:         end_x = 2;
```

```
768:         end_y = 2;
769:         break;
770:       case DOT48_1_3:
771:         end_x = 1;
772:         end_y = 3;
773:         break;
774:       case DOT48_2_3:
775:         end_x = 2;
776:         end_y = 3;
777:         break;
778:       case DOT48_3_3:
779:         end_x = 3;
780:         end_y = 3;
781:         break;
782:       case DOT48_3_1:
783:         end_x = 3;
784:         end_y = 1;
785:         break;
786:       case DOT48_3_2:
787:         end_x = 3;
788:         end_y = 2;
789:         break;
790:       default:
791:         game_abort ("select_sprite_color:未知のスプライトタイプ");
792:         break;
793:       }
794:
795:     for (y = 0; y < end_y; y++)
796:       for (x = 0; x < end_x; x++)
797:         SP_V_INV (SPD_BODY (sp, x, y)) = inv;
798: }
799:
800: /* スプライトパターンの定義 */
801: /* 巨大なマクロ */
802: #define DEF_SP(pat_no,sp_dat)                                    \
803: {                                                                \
804:   int *sp_reg = (int *)(0xeb8000 + 0x80 * (pat_no));             \
805:   int *dat = (int *)(sp_dat);                                    \
806:   /* 0 */                                                        \
807:   *sp_reg ++ = *dat ++;                                          \
808:   *sp_reg ++ = *dat ++;                                          \
809:   *sp_reg ++ = *dat ++;                                          \
810:   *sp_reg ++ = *dat ++;                                          \
811:   *sp_reg ++ = *dat ++;                                          \
812:   *sp_reg ++ = *dat ++;                                          \
813:   *sp_reg ++ = *dat ++;                                          \
```

```
814:   *sp_reg ++ = *dat ++;                                    \
815:   *sp_reg ++ = *dat ++;                                    \
816:   *sp_reg ++ = *dat ++;                                    \
817:   *sp_reg ++ = *dat ++;                                    \
818:   *sp_reg ++ = *dat ++;                                    \
819:   *sp_reg ++ = *dat ++;                                    \
820:   *sp_reg ++ = *dat ++;                                    \
821:   *sp_reg ++ = *dat ++;                                    \
822:   *sp_reg ++ = *dat ++;                                    \
823:   /* 1 */                                                  \
824:   *sp_reg ++ = *dat ++;                                    \
825:   *sp_reg ++ = *dat ++;                                    \
826:   *sp_reg ++ = *dat ++;                                    \
827:   *sp_reg ++ = *dat ++;                                    \
828:   *sp_reg ++ = *dat ++;                                    \
829:   *sp_reg ++ = *dat ++;                                    \
830:   *sp_reg ++ = *dat ++;                                    \
831:   *sp_reg ++ = *dat ++;                                    \
832:   *sp_reg ++ = *dat ++;                                    \
833:   *sp_reg ++ = *dat ++;                                    \
834:   *sp_reg ++ = *dat ++;                                    \
835:   *sp_reg ++ = *dat ++;                                    \
836:   *sp_reg ++ = *dat ++;                                    \
837:   *sp_reg ++ = *dat ++;                                    \
838:   *sp_reg ++ = *dat ++;                                    \
839:   *sp_reg ++ = *dat ++;                                    \
840: }
841:
842: /* 割り込み処理関数 */
843:
844: static  CRTC_REG *const crtc = (CRTC_REG *) 0xe80018;
845:
846: void
847: vsync_disp (void)
848: {
849:
850:   *(short *)0xe82500 = 0x09e4;
851:   if (sp_is_ready || request_def)
852:     {
853:       /* SP スクロールレジスタをCPUに開放 */
854:       *(short *)0xeb0808 &= 0xfdfc;
855:
856:       /* PCGの定義を行う。1/60秒に4PCGを最大とする。間に合うのか？ */
857:       if (request_def)
858:         {
859:           sprite def_req;
```

```
860:          if (def_req = request_def)
861:            {
862:              int anime = SP_ANIME (def_req);
863:              do {
864:                int i;
865:                for (i = 1; i < 128; i++)
866:                  if (!pcg_is_use[i])
867:                    break;
868:                if (i < 128)
869:                  {
870:                    pcg_is_use[i] = 1;
871:                    SP_PCG_NO (def_req, anime) = i;
872:                    DEF_SP (i, SP_PCG_DATA (def_req, anime));
873:                  }
874:              } while (--anime >= 0);
875:              SP_DEFINED (def_req) = 1;
876:              request_def = 0;
877:            }
878:        }
879:      if (sp_is_ready)
880:        {
881:          int i;
882:          SP_CTRL *sp = disp_sp;
883:          SP_CTRL *sp_scr_reg = (SP_CTRL *)0xeb0000;
884:          /* スプライトスクロールレジスタ書き込み */
885:          for (i = 0; i < 128; i++,sp_scr_reg ++)
886:            if (this_no_is_use[i] != NO_USE)
887:              *sp_scr_reg  = sp[i];
888:          ++vsync_counter;
889:          sp_is_ready = 0;
890:        }
891:       {
892:         BG_REG *bg_top = (BG_REG *) 0xebe000;
893:         int i;
894:         for (i = 0; i < 32; i++)
895:           *bg_top++ = bg_array[i];
896:         *(short *)0xeb0800 = -2;
897:         *(short *)0xeb0802 = -2;
898:         *(short *)0xeb0804 = -2;
899:         *(short *)0xeb0806 = -2;
900:       }
901:
902:       /* SP,BG 表示 */
903:       *(short *)0xeb0808 |= 0x0203;
904:
905:       /* グラフィック画面スクロール */
```

```
906:        *crtc = scroll_data;
907:        /* パレットを設定 */
908:        if (palet_def)
909:          {
910:            unsigned short *pal = (unsigned short *) 0xe82000;
911:            unsigned short *p = palet_buf;
912:            int i;
913:            for (i = 0; i < 256; i++)
914:              *pal ++ = *p++;
915:            palet_def = 0;
916:          }
917:         else
918:          {
919:            unsigned short *pal =
920:             (unsigned short *) (0xe82000 + (PAL_C_MIN * sizeof (short)));
921:            unsigned short *p = c_palet;
922:            int i;
923:            for (i = 0; i < PAL_C_MAX - PAL_C_MIN; i++)
924:              *pal ++ = *p++;
925:          }
926:        if (use_half_tone)
927:          *(unsigned short *) 0xe82600 = half_def_data;
928:        else
929:          *(unsigned short *) 0xe82600 = 0b1_0_0_0_0_0_0_010_0_1111;
930:      }
931:    IRTE ();
932: }
```

このスプライト管理処理関数群は，次のようなスプライト管理方法を使っています。まず，論理的にスプライト管理を 2 階層に分割します。

1. 全部で 128 枚のスプライトをハードウェアに密着して管理する下位管理関数群。
2. スプライトを複数組み合わせて，1 つのスプライトとして扱えるようにみせる上位関数群。

ゲームのメイン処理部分はすべて **2.** の範疇の関数群を通じて処理を行い，下位管理部分はメインにみせないようにします。このため，スプライトのデータ構造自体も 2 階層に分割されています。List 4.3 で**#include** する game.h の下位階層部分のデータ構造を，List 4.1 より以下に抜粋します。

```
109: /* 機種，コンパイラに完全に依存する記述 */
110: /* PCG レジスタの構造体 */
111: typedef  union  {
112:      unsigned short dummy;
```

```
113:     struct {
114:       unsigned pos0: 4;
115:       unsigned pos1: 4;
116:       unsigned pos2: 4;
117:       unsigned pos3: 4;
118:     } sp;
119: } SP_REG_U;
120:
121: typedef SP_REG_U  PCG[64];
122:
123: typedef struct {
124:     unsigned v_invert : 1;
125:     unsigned h_invert : 1;
126:     unsigned dummy    : 2;
127:     unsigned color    : 4;
128:     unsigned sp_code  : 8;
129: } BG_REG;
130:
131: /* スプライトスクロールレジスタ構造体 */
132: typedef struct {
133:   short int sp_x;
134:   short int sp_y;
135:   union {
136:     short int dummy;
137:     BG_REG sp;
138:   } sp_ctrl;
139:   union {
140:     short int dummy;
141:     struct {
142:       unsigned dummy   : 13;
143:       unsigned ext     :  1;
144:       unsigned pwr     :  2;
145:     } sp;
146:   } sp_pwr;
147: } SP_CTRL;
```

これらのデータ構造は，ハードウェア構造をそのままCの構造体に対応させた構造をしています。このままでは，実際のゲームに使うには扱いが困難です。たとえば，あるスプライトがアニメーションを行う場合には，この1つのSP_CTRL構造体，つまりハードウェアの1スプライトスクロールレジスタに複数のPCG番号を割り当てて，それを順次切り替えてアニメーションを行います。これらの構造体には，こうした管理を行うためのデー

タ構造が存在していないからです。

そこで，低水準データ構造に階層をもたせます。List 4.1 の

```
149: /* スプライトデータ構造体 */
150:
151:
152: typedef struct {
153:   SP_CTRL *spr;           /* スプライトスクロールレジスタ   */
154:   short   char_no;        /* キャラクタナンバ */
155:   short   pcg_defined;    /* PCGがすでに定義されている場合は1 */
156:   short   anime;          /* アニメーションすれば anime > 0 */
157:   short   pcg_no[8];      /* アニメ用PCG定義番号 */
158:   PCG     *pcg_data[8];   /* アニメ4パターン,PCGのデータへの
                                  ポインタ */
159: } SPRITE;
160:
161: typedef SPRITE *sprite;
```

が低水準管理用のスプライトデータ構造です。各メンバの役割はコメントに記述してあるとおりです。このスプライト管理構造である 1 つのスプライトを表示させるには，次の手順を踏むことになります。

1. SPRITE構造体のメモリを確保する。
2. ハードウェアのスプライトスクロールレジスタを割り当ててメンバsprに格納する。
3. PCG データへのポインタをメンバpcg_data[8]に格納する。
4. 割り込み処理で PCG データを PCG に転送する。同時に，メンバpcg_no[]に実際の PCG 番号が格納される。
5. sprに格納されているスプライトスクロールレジスタに，PCG 番号やスプライトパレット，優先順位等を設定する。
6. 割り込み関数に，表示を通知する。
7. 割り込み処理が実際にスプライトスクロールレジスタに転送する。
8. 画面に表示される。

こういった手順になるのですが，これでは 16×16 ドットの 1 スプライトを表示するだけです。スプライトはいつもこの大きさで管理できるわけではありません。複数のスプライトを組み合わせて表示を行う場合には，この作業を組み合わされたスプライトの数だけ行う必要があるのです。

こういった細かい作業をいちいち上位の処理で行っていたのでは，効率が悪くて実際には使いものになりません。そこで，あらかじめいくつかのスプライトを組み合わせたスプライト群を管理する関数を作成して，こういった煩雑な処理をすべて上位にみせないようにします。List 4.1 の

```
163: enum sp_type {
164:   DOT16,
165:   DOT32_1_2,
166:   DOT32_2_1,
167:   DOT32_2_2,
168:   DOT48_1_3,
169:   DOT48_2_3,
170:   DOT48_3_3,
171:   DOT48_3_1,
172:   DOT48_3_2,
173: };
174:
175: typedef struct {
176:   short dif_x;
177:   short dif_y;
178: } move;
179:
180: typedef union {
181:   move *move_array;
182:   int (*move_func)();
183: } move_def;
184:
185: typedef struct SP_DATA {
186:   enum sp_type type;
187:   short life;
188:   short registance;
189:   short work0;
190:   short work1;
191:   short work2;
192:   short work3;
193:   sprite body[3][3];
194:   move_def   def_move;
195:   short  *anime_def;
196: } *Sprite;
```

がそのための最上位のスプライトデータ構造です。この構造では，最大 48×48 ドットのスプライトを移動，表示，反転表示できるようになっています。これらのメンバは，直接メンバ名を指定して扱わずに，後述するマクロですべてアクセスします。メンバ名を直接記述した場合には，今まで説明した構造体の中身を改変したときに，作業量が大幅に増加してしまいます。データ階層が複雑になっているので，各構造体のメンバへのアクセスを直接.演算子や->演算子で行うと，これらの演算子の嵐になります。これでは全体に見通し

が悪化しますので，各メンバへのアクセスをマクロを使って簡略化します。以下がそのマクロ群の説明です。

●Table 4.3　スプライト管理マクロ

スプライト管理低水準マクロ	
SP_CHAR_NO()	内部管理のスプライト番号をアクセスするマクロです。PCG の番号ではありません。
SP_SP_PTR()	スプライトに割り当てられたスプライトスクロールレジスタイメージへのポインタをアクセスするマクロです。
SP_POS_X()	スプライトスクロールレジスタの現在の X 座標の値をアクセスするマクロです。
SP_POS_Y()	スプライトスクロールレジスタの現在の Y 座標の値をアクセスするマクロです。
SP_CODE()	スプライトスクロールレジスタの現在のスプライトコード (PCG 番号) の値をアクセスするマクロです。
SP_PALET()	スプライトスクロールレジスタの現在のパレット番号の値をアクセスするマクロです。
SP_PRIORITY()	スプライトスクロールレジスタの現在のパレット番号の値をアクセスするマクロです。
SP_DEFINED()	スプライトスクロールレジスタの現在の PCG 番号の PCG が，実際に PCG に登録されているかどうかを示すフラグをアクセスするマクロです。
SP_PCG_DATA_PTR()	スプライトの PCG データが格納されているポインタをアクセスするマクロです。
SP_ANIME()	スプライトのアニメーション数にアクセスするマクロです。
SP_PCG_NO()	スプライトの実際の PCG 番号をアクセスするマクロです。
SP_PCG_DATA()	スプライトの実際の PCG データをアクセスするマクロです。
SP_H_INV()	スプライトの水平方向反転フラグをアクセスするマクロです。
SP_V_INV()	スプライトの垂直方向反転フラグをアクセスするマクロです。
スプライト管理高水準マクロ	
SPD_BODY()	スプライト群の個別部分にアクセスするマクロです。
SPD_TYPE()	スプライト群のタイプをアクセスするマクロです。
SPD_MOVE()	スプライト群の動きを定義する配列をアクセスするマクロです。
SPD_MOVE_FUNC()	スプライト群の動きを定義する関数へのポインタをアクセスするマクロです。
SPD_LIFE()	スプライト群の寿命をアクセスするマクロです。
SPD_REGIST()	スプライト群のいわゆる「カタサ」をアクセスするマクロです。
SPD_ANIME_PTR()	スプライト群のアニメーション定義のポインタへアクセスするマクロです。
SPD_WORK0() SPD_WORK1() SPD_WORK2() SPD_WORK3()	スプライト群汎用ワークへアクセスするマクロです。
SPD_POS_X()	スプライト群の x 座標アクセスマクロです。
SPD_POS_Y()	スプライト群の y 座標アクセスマクロです。
MOVE_X()	スプライト群の動き，x 方向の定義配列を時間基準でアクセスするマクロです。
MOVE_Y()	スプライト群の動き，y 方向の定義配列を時間基準でアクセスするマクロです。
NO_USE	スプライトが使われていない。
DISP	スプライトが表示中。
USE	スプライトが非表示中。

それでは，具体的にソースの説明をしましょう。ここからは，特に理由がない限り，細かい関数の記述には言及しません。List 4.3 をご覧ください。

▶4 行〜6 行

物理的に 128 個に固定されたスプライトスクロールレジスタと，128 個 (または 256 個)

のPCGをうまく管理しないと，同じPCGのデータに別のPCGを割り当てたりといった，ムダが生じます。スプライトスクロールレジスタの割り当てとPCGの割り当ては別個に行う必要があります。そのために，管理用の配列を用意しておきます。

●Table 4.4　スプライトスクロールレジスタとPCGの管理用配列

変　数	用　途
disp_sp[128];	スプライトスクロールレジスタバッファ
this_no_is_use[128];	スプライトスクロールレジスタ割り当て管理配列
pcg_is_use[128];	PCG割り当て管理配列

▸8行～14行

```
static void active_sprite (sprite sp);
```

下位構造のスプライトspを表示状態にします。

▸16行～22行

```
static void suspend_sprite (sprite sp);
```

下位構造のスプライトspを非表示状態にします。

▸24行～51行

```
static void free_sprite (sprite sp);
```

下位構造のスプライトspに割り当てられているスクロールレジスタを開放します。引数spに割り当てられていたPCGエリアは，共有しているスプライトがない場合は開放されます。make_sprite()ではPCGの割り当てが行われていないので，make_sprite()が返した値をそのままfree_sprite()に渡すことはできない点に注意してください。

▸53行～85行

```
static sprite alloc_sprite (void);
```

128個のスクロールレジスタの中から，使われていないレジスタを捜して，それを割り当てた下位構造スプライトへのポインタを返します。ここで，スクロールレジスタが不足の場合と，malloc()がNULLを返す(メモリが足りない)場合にエラーが生じます。戻り値には，割り当てられたスクロールレジスタイメージのポインタと，スクロールレジスタ番号をセットします。PCGポインタはNULLで初期化しておきます。

▸87行～108行

```
static sprite alloc_sprite_dummy (void);
```

バックグランドや常時発生消滅するキャラクタのような，スプライトスクロールレジスタの割り当てが必要なく，PCG データのみハードに登録したい場合に用いる関数です。alloc_sprite()と異なるのは，スプライトスクロールレジスタが割り当てられない点だけです。

▸110行～122行

```
static sprite make_sprite (int anime, PCG **pcg_data, int sc_alloc);
```

引数sc_allocの値に応じて下位スプライト構造を割り当て，PCG データポインタフィールドを引数pcg_dataの内容で初期化します。注意しなければならないのは，この関数では構造体内部のポインタを設定するだけで，ハードウェアの PCG には何も設定しないことです。この関数を呼び出すだけではキャラクタは表示できません。

▸124行～145行

```
static sprite copy_sprite (sprite sp);
```

下位スプライト構造spと PCG パターンを共有するスプライトを生成して，そのポインタを返します。引数spの PCG パターンがハードウェアに未登録の場合は，登録してから値を返します。そのために，1 垂直帰線期間のウエイトが入ることがありますので，注意してください。

ここまでの関数は，すべて低水準のスプライト構造を扱う関数群です。これらの関数はすべて **static** 宣言されていますので，sprite.c の内部以外からでは呼び出しできませんし，その必要もありません。

実際にスプライトを操作するには，これ以降の上位関数群を呼び出します。上位の関数はすべてSprite型の変数を扱います。表示の切り替えや，表示位置の移動等は内部データが変更されるだけで，sp_is_readyを '1' にして，垂直帰線期間割り込みがハードウェアに書き込みを行うまでは，表面的には何の変化も起こりません。逆にいえば，何回スプライトを移動しても，割り込みルーチンに通知しなければ問題ありません。

▸147行～200行

```
void display_sprite (Sprite sp, int sw);
```

引数spで与えられたSpriteをint swで表示，非表示に切り替えます。swが'0'の場合は非表示，それ以外では表示状態にします。

▶203行～254行

```
void delete_sprite (Sprite sp);
```

引数spで与えられたSpriteをシステムから削除します。引数spは非表示状態にしておき，非表示状態に設定した後，垂直帰線期間割り込みを1回経過する必要があります。これを怠った場合，該当スプライトが画面上に残ることがあります。

▶256行～323行

```
Sprite dup_sprite (Sprite sp);
```

引数spで与えられたSpriteと，PCGを共用するSpriteを生成します。ハードウェアにPCGが未登録の場合は，登録してから複写します。この場合，関数から戻るまでに1垂直帰線期間×登録数の時間がかかることがあります。

▶326行～394行

```
Sprite def_sprite (enum sp_type type, PCG ***pcg, int anime);
```

スプライトのタイプsp_typeでアニメーション数animeのスプライトを，スプライトスクロールレジスタを割り当てて作成します。登録用のPCGパターンはPCG ***pcgからデータを得ます。ハードウェアへのPCG転送は行われませんので，ウエイトはなく呼び出し元へ復帰します。

▶396行～464行

```
Sprite def_sprite_dummy (enum sp_type type, PCG ***pcg, int anime);
```

スプライトのタイプsp_typeでアニメーション数animeのスプライトを，スプライトスクロールレジスタを割り当てずに作成します。登録用のPCGパターンはPCG ***pcgからデータを得ます。ハードウェアへのPCG転送は行われませんので，ウエイトはなく呼び出し元へ復帰します。

▶466行～520行

```
void move_sprite_diff (Sprite sp, int dx, int dy);
```

spで与えられたスプライトを，dx，dyで与えられたドット数だけ相対移動します。

▸522行～576行

```
void move_sprite_abs (Sprite sp, int ax, int ay);
```

spで与えられたスプライトをドット位置ax，ayに移動します。

▸578行～639行

```
void select_sprite_pcg (Sprite sp, int sel);
```

spで与えられたスプライトのアニメーションを行います。selはアニメーション番号で0～7を指定します。PCGがどの番号に割り当てられているかについては，考慮する必要はありません。

▸641行～692行

```
void select_sprite_color (Sprite sp, int col);
```

spで与えられたスプライトのスプライトパレット番号をcolにします。

▸694行～745行

```
void select_sprite_h_invert (Sprite sp, int inv);
```

spで与えられたスプライトのスプライト水平反転ビットをinvにします。

▸747行～798行

```
void select_sprite_v_invert (Sprite sp, int inv);
```

spで与えられたスプライトのスプライト垂直反転ビットをinvにします。

▸800行～932行

```
void vsync_disp (void);
```

垂直帰線期間割り込み処理関数の定義部分です。次のような処理を行っています。

1. **グラフィック画面の垂直スクロール。**

2. **PCG データを PCG に転送。**
3. **スプライトスクロールレジスタの書き込み。**
4. **バックグランドの書き換え。**
5. **グラフィック画面パレットの設定。**
6. **半透明および特殊プライオリティの設定。**

このスプライト管理関数群は複雑な処理を行っていますが，これでもまだ「手抜き」があっちこっちに散在しています。ライブラリの仕様書ライクにスペックを記述すれば，以下のようになります。

1. 最大 48 × 48 ドットのスプライトを使用できます。スプライトの組み合わせの種類は
   16 × 16
   16 × 32
   32 × 16
   32 × 32
   16 × 48
   48 × 16
   32 × 48
   48 × 32
   48 × 48
   の 9 種類です。
2. スプライト 1 つに 8 パターンまでのアニメーションを登録できます。パターンの指定は，関数呼び出しで簡単に行えます。
3. 1 スプライトに 1 スプライトパレットです。複数のスプライトで構成されていた場合，任意の部分だけパレットを指定するといった操作はできません。
4. 反転は上下左右に反転表示可能です。複数のスプライトで構成されていた場合，任意の部分を反転するといった操作はできません。
5. スプライトの優先順位は指定できません。重ね合わせで不都合が起きないように，あらかじめ考慮する必要があります。

3.～**5.** の仕様は,「手抜き」の結果でもあります。もっと細かい機能をちゃんと織り込むことも工夫すればできるでしょうが，本書で作成するゲームではこの仕様で必要十分ですので，妥協することにします。

■ 4.5 各種下請け処理の制作

ある程度の規模のプログラムでは，雑多な下請けを行う関数を必要とします。たとえば，`malloc()`はメモリを割り当てる関数ですが，呼び出すごとに本当に確保できたかどうかを

チェックしていたのでは，ムダなコーディング作業が増えてしまいます。このような下請け関数や，画面の初期化，画面データのロード等を行う下請け関数をここで作成しておきます。その前に，少しゲームとは直接関係のない話を…。

プログラムを作成していると，バグは当然のごとく発生します。**X68000/X68030** では，CPU のプログラム誤動作に対する保護機能が強力なので，大きな被害につながるような暴走は生じにくいのですが，それでも「バスエラー」や「アドレスエラー」の白い窓が開くことはしばしばです。未完成のプログラムならともかく，完成したプログラムで予期せぬバグのために，この白い窓が開くのは本意ではないでしょう。このゲームは，**X68000/X68030** のフリーなライブラリ **LIBC** で作成されています。この **LIBC** はシグナル関係のライブラリが充実しており，このような致命的なエラーをプログラムが引き起こしたときに，「白い窓」を開くことなくプログラムを中断させることができるようになっています。本書のサンプルゲームでは，このシグナルハンドラを利用して致命的なエラーを扱っていますが，**XC Ver.1.0** および **XC Ver.2.0** 環境では，自前でこれらのエラーをトラップしておくしか方法がありません。このエラーハンドリングを行う方法は，ゲームプログラムの本質的な部分ではないので，Appendix 1(P.248) に収録しておきます。この下請け関数の制作では，**LIBC** だけでなく，**XC** でもエラーハンドルを行うようなソースになっていますが，この部分を詳細には解説しませんので，詳しい内容は Appendix 1 を参照してください。

List 4.4 が下請け関数群のソースリストです。

List 4.4 utiles.c

```
 1: /* C Magagize X68000 Game by Yohino (C) 1992 03 05 */
 2:
 3: #include "game.h"
 4:
 5: #ifdef USE_XC_LIB
 6: static struct {
 7:  int flag;                  /* エラー種別フラグ */
 8:  unsigned short sr_reg;     /* エラー時の SR    */
 9:  int pc_reg;                /* エラー時の PC    */
10:  char *mes;                 /* メッセージ       */
11: } trapbuf;
12:
13: static void (*trap_14)();   /* 元の trap 14 ベクタを保存 */
14: #endif
15:
16:
17: volatile void
18: game_abort (char *mes)
19: {
20:   VDISPST (0, 0, 1);
21:   CRTCRAS (0, 0);
22:   CRTMOD (16);
```

```
23:   B_CURON ();
24:   printf ("\n%s\n", mes);
25:   printf ("  異常終了します\n");
26: #ifdef USE_XC_LIB
27:   INTVCS(0x2e,trap_14);
28: #endif
29:   longjmp (err_buf, 1);
30: }
31:
32: void *
33: xmalloc (int size)
34: {
35:   void *ret = malloc (size);
36:   if (!ret)
37:     game_abort ("メモリ不足です");
38:   return ret;
39: }
40:
41: static GAME_PCG *
42: PCGロード (char *fname)
43: {
44:   FILE *fp = fopen (fname, "rb");
45:   char mes[64];
46:   if (!fp)
47:     {
48:       sprintf (mes, "ファイル %s が見つかりません",fname);
49:       game_abort (mes);
50:     }
51:   else
52:     {
53:       int num = filelength (fileno (fp))/ sizeof (PCG);
54:       GAME_PCG *retval = xmalloc (sizeof (GAME_PCG));
55:       PCG **ret = xmalloc (sizeof (PCG *) * num);
56:       int i;
57:       retval -> num_p = num -1;   /* Dubug 05 09 from Pekin */
58:       retval -> pcg_ptr_ptr = ret;
59:       for (i = 0; i < num; i++)
60:         ret[i] = xmalloc (sizeof (PCG));
61:       for (i = 0; i < num; i++)
62:         {
63:           int size = fread (ret[i], sizeof (char), sizeof (PCG), fp);
64:           if (size != sizeof (PCG))
65:             {
66:               sprintf (mes, "ファイル %s が異常です", fname);
67:               game_abort (mes);
68:             }
```

```
69:         }
70:       return retval;
71:     }
72: }
73:
74: static void
75: ダミー初期化 (void)
76: {
77:   自機 = def_sprite (DOT16,
78:                      &(load_files[C_自機].pcg->pcg_ptr_ptr),
79:                      load_files[C_自機].pcg->num_p);
80:   move_sprite_abs (自機, 128, 220);
81:   select_sprite_pcg (自機, 0);
82:   select_sprite_color (自機, 1);
83:   display_sprite (自機, 表示);
84:   SPD_WORK1 (自機) = 0;
85:   SPD_REGIST (自機) = 3;
86:
87:   蓄積sp = def_sprite (DOT16,
88:                        &(load_files[C_蓄積].pcg->pcg_ptr_ptr),
89:                          load_files[C_蓄積].pcg->num_p);
90:   move_sprite_abs (蓄積sp, 128, 220 - 10);
91:   select_sprite_pcg (蓄積sp, 0);
92:   select_sprite_color (蓄積sp, 3);
93:   display_sprite (蓄積sp, 非表示);
94:
95:   弾ダミー = def_sprite_dummy (DOT16,
96:                                &(load_files[C_弾].pcg->pcg_ptr_ptr),
97:                                load_files[C_弾].pcg->num_p);
98:   select_sprite_pcg (弾ダミー, 0);
99:
100:  タイプA = def_sprite_dummy (DOT16,
101:                              &(load_files[C_敵0].pcg->pcg_ptr_ptr),
102:                              load_files[C_敵0].pcg->num_p);
103:  select_sprite_pcg (タイプA, 0);
104:
105:  タイプB = def_sprite_dummy (DOT16,
106:                              &(load_files[C_敵1].pcg->pcg_ptr_ptr),
107:                              load_files[C_敵1].pcg->num_p);
108:  select_sprite_pcg (タイプB, 0);
109:  爆発0ダミー = def_sprite_dummy (DOT16,
110:                                  &(load_files[C_爆発0].pcg->pcg_ptr_ptr),
111:                                  load_files[C_爆発0].pcg->num_p);
112:  select_sprite_pcg (爆発0ダミー, 0);
113:
114:  ゲージ = def_sprite_dummy (DOT16,
```

```
115:                                &(load_files[C_ゲージ].pcg->pcg_ptr_ptr),
116:                                load_files[C_ゲージ].pcg->num_p);
117:   select_sprite_pcg (ゲージ, 0);
118:
119:   数字 = def_sprite_dummy (DOT16,
120:                            &(load_files[C_数字].pcg->pcg_ptr_ptr),
121:                            load_files[C_数字].pcg->num_p);
122:   select_sprite_pcg (数字, 0);
123: }
124:
125: /* PCG データをロードする */
126: void
127: ロードデータ (void)
128: {
129:   char fname[64];
130:   /* 自機 */
131:   strcpy (fname, LOAD_DIR);
132:   strcat (fname, load_files[C_自機].file_name);
133:   load_files[C_自機].pcg = PCGロード(fname);
134:   strcpy (fname, LOAD_DIR);
135:   strcat (fname, load_files[C_蓄積].file_name);
136:   load_files[C_蓄積].pcg = PCGロード(fname);
137:
138:   /* 弾 */
139:   strcpy (fname, LOAD_DIR);
140:   strcat (fname, load_files[C_弾].file_name);
141:   load_files[C_弾].pcg = PCGロード(fname);
142:
143:   /* 敵キャラクタ */
144:   strcpy (fname, LOAD_DIR);
145:   strcat (fname, load_files[C_敵0].file_name);
146:   load_files[C_敵0].pcg = PCGロード(fname);
147:   strcpy (fname, LOAD_DIR);
148:   strcat (fname, load_files[C_敵1].file_name);
149:   load_files[C_敵1].pcg = PCGロード(fname);
150:
151:   /* 爆発 */
152:   strcpy (fname, LOAD_DIR);
153:   strcat (fname, load_files[C_爆発0].file_name);
154:   load_files[C_爆発0].pcg = PCGロード(fname);
155:
156:   /* ゲージ */
157:   strcpy (fname, LOAD_DIR);
158:   strcat (fname, load_files[C_ゲージ].file_name);
159:   load_files[C_ゲージ].pcg = PCGロード(fname);
160:
```

```
161:   /* 数字 */
162:   strcpy (fname, LOAD_DIR);
163:   strcat (fname, load_files[C_数字].file_name);
164:   load_files[C_数字].pcg = PCGロード(fname);
165:
166:   ダミー初期化 ();
167: }
168:
169: #ifndef USE_XC_LIB
170: static int
171: getl (FILE *fp)
172: {
173:   int i = fgetc (fp);
174:   i = ((i << 8) | fgetc (fp));
175:   i = ((i << 8) | fgetc (fp));
176:   i = ((i << 8) | fgetc (fp));
177:   return i;
178: }
179: #endif
180:
181: void
182: 画面初期化 (void)
183: {
184:   int i;
185:   CRTMOD (10);
186:   G_CLR_ON ();
187:   B_CUROFF ();
188:   for (i = 0; i < 128; i++)
189:     SP_REGST (i, -1, 0, 0, 0, 0);
190:   SP_INIT ();
191:   SP_ON ();
192:   {
193:     int j;
194:     FILE *fp = fopen (LOAD_DIR "パレット.PAL", "rb");
195:     if (!fp)
196:       game_abort ("パレットファイルが読めません");
197:     for (j = 1; j < 4; j++)
198:       for (i = 0; i < 16; i++)
199:         SPALET (0x80 + i, j, getl (fp));
200:   }
201:   VDISPST (vsync_disp, 0, 1);
202:   CRTCRAS (raster_scroll, 1);
203: }
204:
205: void
206: 画面終了処理 (void)
```

```
{
  VDISPST (0, 0, 1);
  CRTCRAS (0, 0);
  CRTMOD (16);
  B_CURON ();
  while (B_KEYSNS ())
    B_KEYINP ();
}

#ifdef USE_XC_LIB
static void
trap14(void)
{
  PRAMREG (code, d7);       /* パラメータレジスタ d7 */
  PRAMREG (add,  a6);       /* パラメータレジスタ a6 */
  SET_FRAME (a5);           /* フレームポインタを a5 に指定 */
  char *p =(char *)add;

  /* code からエラーを分析 */
  switch (code&0xffff)
    {
    case 2:
      /* バスエラー */
      trapbuf.flag = code;
      trapbuf.mes = "バスエラーが発生しました";
      break;
    case 3:
      /* アドレスエラー */
      trapbuf.flag = code;
      trapbuf.mes = "アドレスエラーが発生しました";
      break;
    case 4:
      /* おかしな命令 */
      trapbuf.flag = code;
      trapbuf.mes = "おかしな命令を実行しました";
      break;
    }
 if ((code &0xff00))
   {
     if ((code &0xff) == 2)
       {
         trapbuf.flag = -1;
         trapbuf.mes  = "ドライブの準備ができていません";
       }
    }
 if (trapbuf.flag || (code&0xffff) == 0x1f || (code&0xffff) == 0x301f)
```

```
253:     {
254:       trapbuf.sr_reg = *(short *)p;
255:       p+=2;
256:       trapbuf.pc_reg = *(int *)p;
257:       asm ("\tmove.w #$ff,d0\n\ttrap #15\n");
258:       IJUMP_RTE();
259:     }
260:   else
261:     IJUMP(trap_14);
262: }
263: #endif
264:
265: static void
266: my_abort (int val)
267: {
268: #ifdef USE_XC_LIB
269:   INTVCS(0x2e,trap_14);
270:   VDISPST (0, 0, 1);
271:   CRTCRAS (0, 0);
272:   CRTMOD (16);
273:   B_CURON ();
274:   if (trapbuf.flag > 0)
275:     {
276:       extern (*_main)();
277:       extern int _SSTA;
278:       extern int _PSTA;
279:       trapbuf.flag = 0;
280:       fprintf (stderr,"%s\r\n",trapbuf.mes);
281:       fprintf (stderr," pc=%X",trapbuf.pc_reg);
282:       fprintf (stderr," offset=%X\r\n",trapbuf.pc_reg - (int) &_main);
283:     }
284:   else
285:     {
286:       fprintf (stderr,"%s\r\n",trapbuf.mes);
287:       game_abort ("");
288:     }
289:   exit (1);
290: #else
291:   switch (val)
292:     {
293:     case SIGSEGV:
294:       game_abort ("バスエラー発生");
295:       break;
296:     case SIGBUS:
297:       game_abort ("アドレスエラー発生");
298:       break;
```

```
299:     case SIGILL:
300:       game_abort ("おかしな命令実行");
301:       break;
302:     default:
303:       game_abort ("なぜ？なぜ？");
304:       break;
305:     }
306: #endif
307: }
308:
309: void
310: init_trap14 (void)
311: {
312: #ifdef USE_XC_LIB
313:   trap_14 = (void *)INTVCG (0x2e);
314:   INTVCS (0x2e, trap14);
315:   INTVCS (0xfff2, my_abort);
316:   INTVCS (0xfff1, my_abort);
317: #else
318:   signal (SIGILL, my_abort);
319:   signal (SIGBUS, my_abort);
320:   signal (SIGSEGV, my_abort);
321: #endif
322: }
```

それでは，各関数について説明しましょう。

▶17行～30行

```
volatile void game_abort (char *mes);
```

ゲーム実行中に致命的なエラーが発生した場合に，この関数が呼ばれます。この関数では，すべての割り込み処理を登録解除し，画面モードを標準高解像度モードに戻し，カーソル表示を許可した後，longjmp()を使ってmain()へ直接復帰します。main()ではlongjmp()復帰後，exit()しています。**volatile**属性の関数は**GCC**拡張です。この属性の関数は，呼び出し元に帰らない関数であることをコンパイラに知らせるのに用います。

▶32行～39行

```
void *xmalloc (int size);
```

sizeバイトのメモリをmalloc()で確保します。メモリ不足で確保できなかった場合は，game_abort()を呼び出してプログラムを終了させます。

▸41 行〜72 行

```
static GAME_PCG *PCGロード (char *fname);
```

fnameで与えられたファイルからPCGデータをロードします。同時にPCGファイルの大きさから，そのPCGファイルのアニメーション数を計算します。PCGデータを格納するメモリはxmalloc()で確保しています。

▸74 行〜123 行

```
static void ダミー初期化 (void);
```

スプライトやバックグランドの表示を準備します。画面上に常時表示されている自機と，弾の溜め撃ちを表示するスプライトは，ここでスプライトスクロールレジスタを割り当てておきます。P.190で，def_sprite()は，ハードウェアにPCGデータを転送しない仕様であると述べました。実際にPCGデータが転送されるのは，関数select_sprite_pcg()が呼び出されたときです。この関数が呼び出されると，PCGデータがハードウェアに転送されて，PCG番号が決定され，表示可能な状態になります。

常時表示されないキャラクタは，単に表示準備のためにPCGにデータを転送しておき，実際に表示するときには，dup_sprite()でこのダミーデータにスプライトスクロールレジスタを割り当て，スプライトを作成して表示を行います。

▸125 行〜167 行

```
void ロードデータ (void);
```

ゲームで使用するPCGパターンをファイルから読み取ります。すべてを正常に読み取ることができたら，前述の関数ダミー初期化 ()を呼び出して，PCG関係のデータロードを完了します。ファイルが読めないといったエラーが発生した場合には，game_abort()が呼び出されて，プログラムが中断されます。

▸169 行〜179 行

```
static int getl (FILE *fp);
```

LIBCには，ファイルfpから機種依存形式で**int**を読み出す関数getl()がないので，この**XC**の関数を使って，モトローラ形式で32ビット整数をファイルfpから読み出します。この関数では，エラーチェックを行っていません。

▸181 行～203 行

```
void 画面初期化 (void);
```

表示画面を初期化します。スプライトのハードウェアを初期化したり，カーソル表示の禁止，垂直帰線期間割り込みの設定，ラスター割り込みの設定などを行っています。

▸205 行～214 行

```
void 画面終了処理 (void);
```

ゲーム終了時の設定を行います。最後に，ゲーム実行時に溜まってしまったキーボードのキー押し下げ情報をすべてクリアにしています。これを行わないと，ゲーム終了と同時に，押されていたキー情報がどっと表示されて，みっともない画面になります。

▸216 行～263 行

```
static void trap14 (void);
```

XC のライブラリを使用するためのエラーハンドラです。詳しくは Appendix 1(P.248) を参照してください。

▸265 行～307 行

```
static void my_abort (int val);
```

LIBC のシグナルハンドラに登録する関数です。

▸309 行～322 行

```
void init_trap14 (void);
```

LIBC 環境ならシグナル関係の登録を行い，**XC** 環境なら自前のエラーハンドラを登録します。

■ 4.6 自機の管理

自機のスプライトの移動や，ジョイスティックの読み取りを行うのが mychr.c です。「溜め撃ち可能」なので，R-TYPE(アイレム) もどきのアニメーション処理や，自機が発射する弾の大きさの設定等を行っています。全リストを List 4.5 に示します。

List 4.5 mychr.c

```
 1: /* C Magazine X68000 Game, 自機処理 */
 2:
 3: #include "game.h"
 4:
 5: /* 自機の移動方向と弾の発射を読み取る*/
 6: enum 移動方向
 7: 方向読取 (void)
 8: {
 9:   unsigned char data;
10:   static short アニメ;
11:   {
12:     /* GCC asm 文を使って trapを発行する */
13:     register int d0 asm ("d0");
14:     register int d1 asm ("d1");
15:     d1 = 0;
16:     d0 = 0x3b;
17:     asm ("trap #15": "=d" (d0):"0" (d0), "d" (d1));
18:     data = ~d0;
19:   }
20:   if (data & 0b01000000)
21:     終 = 1;
22:
23:   if (やられ自機)
24:     {
25:       display_sprite (蓄積sp, 非表示);
26:       蓄積 = 0;
27:       return 静止;
28:     }
29:
30:   /* 自機の弾発射処理 */
31:   if (data & 0b00100000)
32:     {
33:       if (弾発射 == 0)
34:         弾発射 = 2;
35:       else if (蓄積 < 30 * 5)
36:         {
37:           蓄積 += 4;
38:         }
39:       if (蓄積)
40:         {
41:           short sel = (アニメ >> 2) % 5;
42:           if (sel)
43:             {
```

```
44:                 display_sprite (蓄積sp, 表示);
45:                 select_sprite_pcg (蓄積sp, sel - 1);
46:               }
47:             else
48:               display_sprite (蓄積sp, 非表示);
49:             アニメ++;
50:           }
51:       }
52:     else if (弾発射 == 2)
53:       {
54:         アニメ = 0;
55:         弾発射 = 1;
56:       }
57:
58:     /* 移動方向の決定 */
59:     switch (data & 0b00001111)
60:       {
61:       case 0b1000:
62:         return 右;
63:       case 0b0100:
64:         return 左;
65:       case 0b0001:
66:         return 上;
67:       case 0b0010:
68:         return 下;
69:       case 0b0101:
70:         return 左上;
71:       case 0b0110:
72:         return 左下;
73:       case 0b1001:
74:         return 右上;
75:       case 0b1010:
76:         return 右下;
77:       }
78:     return 静止;
79: }
80:
81: /* 自機の移動処理を行う。
82:    1ドット移動を可能にするため少し処理を加えておく */
83: void
84: 自機移動処理 (void)
85: {
86:     static enum 移動方向 前回移動方向 = 静止;
87:     static int 同一方向カウンタ;
88:     static int 傾斜カウンタ;
89:     enum 移動方向 移動方向 = 方向読取 ();
```

```
 90:
 91:   if (SPD_WORK1 (自機))
 92:     if ((--SPD_WORK1 (自機)) == 0)
 93:       select_sprite_color (自機, 1);
 94:
 95:   if (移動方向 == 静止)
 96:     {
 97:       同一方向カウンター--;
 98:       if (同一方向カウンタ > 2)
 99:         select_sprite_pcg (自機, 1);
100:       else
101:         {
102:           select_sprite_pcg (自機, 0);
103:           select_sprite_h_invert (自機, 0);
104:         }
105:       前回移動方向 = 静止;
106:       return;
107:     }
108:   else
109:     {
110:       int 移動量;
111:       /* 前回同じ方向に動いていれば加速して動く */
112:       if (移動方向 == 前回移動方向)
113:         {
114:           if (同一方向カウンタ < 4)
115:             {
116:               移動量 = 1;
117:               同一方向カウンタ++;
118:             }
119:           else
120:             移動量 = 2;
121:         }
122:       else
123:         {
124:           同一方向カウンタ = 0;
125:           移動量 = 1;
126:         }
127:
128:       switch (移動方向)
129:         {
130:         case 上:
131:           if (SPD_POS_Y (自機) > 32)
132:             {
133:               move_sprite_diff (自機, 0, -移動量);
134:               move_sprite_diff (蓄積sp, 0, -移動量);
135:             }
```

```
136:            select_sprite_pcg (自機, 0);
137:            select_sprite_h_invert (自機, 0);
138:            break;
139:          case 下:
140:            if (SPD_POS_Y (自機) < 256- 16)
141:              {
142:                move_sprite_diff (自機, 0, 移動量);
143:                move_sprite_diff (蓄積sp, 0, 移動量);
144:              }
145:            select_sprite_pcg (自機, 0);
146:            select_sprite_h_invert (自機, 0);
147:            break;
148:          case 右:
149:            if (SPD_POS_X (自機) < 256 - 16)
150:              {
151:                move_sprite_diff (自機, 移動量, 0);
152:                move_sprite_diff (蓄積sp, 移動量, 0);
153:              }
154:            if (移動量 == 1)
155:              select_sprite_pcg (自機, 1);
156:            else
157:              select_sprite_pcg (自機, 2);
158:            select_sprite_h_invert (自機, 1);
159:            break;
160:          case 左:
161:            if (SPD_POS_X (自機) > 32)
162:              {
163:                move_sprite_diff (自機, -移動量, 0);
164:                move_sprite_diff (蓄積sp, -移動量, 0);
165:              }
166:            if (移動量 == 1)
167:              select_sprite_pcg (自機, 1);
168:            else
169:              select_sprite_pcg (自機, 2);
170:            select_sprite_h_invert (自機, 0);
171:            break;
172:          case 左上:
173:            if (前回移動方向 == 移動方向)
174:              傾斜カウンタ++;
175:            else
176:              傾斜カウンタ = 0;
177:            if (移動量 == 1)
178:              select_sprite_pcg (自機, 1);
179:            else
180:              select_sprite_pcg (自機, 2);
181:            if (傾斜カウンタ % 3 == 2)
```

```
182:            移動量 /= 2;
183:          if (SPD_POS_X (自機) > 32 && SPD_POS_Y (自機) > 32)
184:            {
185:              move_sprite_diff (自機, -移動量, -移動量);
186:              move_sprite_diff (蓄積sp, -移動量, -移動量);
187:            }
188:          select_sprite_h_invert (自機, 0);
189:          break;
190:        case 左下:
191:          if (移動量 == 1)
192:            select_sprite_pcg (自機, 1);
193:          else
194:            select_sprite_pcg (自機, 2);
195:          if (前回移動方向 == 移動方向)
196:            傾斜カウンタ++;
197:          else
198:            傾斜カウンタ = 0;
199:          if (傾斜カウンタ % 3 == 2)
200:            移動量 /= 2;
201:          if (SPD_POS_X (自機) > 32 && SPD_POS_Y (自機) < 256- 16)
202:            {
203:              move_sprite_diff (自機, -移動量, 移動量);
204:              move_sprite_diff (蓄積sp, -移動量, 移動量);
205:            }
206:          select_sprite_h_invert (自機, 0);
207:          break;
208:        case 右上:
209:          if (移動量 == 1)
210:            select_sprite_pcg (自機, 1);
211:          else
212:            select_sprite_pcg (自機, 2);
213:          if (前回移動方向 == 移動方向)
214:            傾斜カウンタ++;
215:          else
216:            傾斜カウンタ = 0;
217:          if (傾斜カウンタ % 3 == 2)
218:            移動量 /= 2;
219:          if (SPD_POS_X (自機) < 256 - 16 && SPD_POS_Y (自機) > 32)
220:            {
221:              move_sprite_diff (自機, 移動量, -移動量);
222:              move_sprite_diff (蓄積sp, 移動量, -移動量);
223:            }
224:          select_sprite_h_invert (自機, 1);
225:          break;
226:        case 右下:
227:          if (移動量 == 1)
```

```
228:             select_sprite_pcg (自機, 1);
229:           else
230:             select_sprite_pcg (自機, 2);
231:           if (前回移動方向 == 移動方向)
232:             傾斜カウンタ++;
233:           else
234:             傾斜カウンタ = 0;
235:           if (傾斜カウンタ % 3 == 2)
236:             移動量 /= 2;
237:           if (SPD_POS_X (自機) < 256 - 16 && SPD_POS_Y (自機) < 256- 16)
238:             {
239:               move_sprite_diff (自機, 移動量, 移動量);
240:               move_sprite_diff (蓄積sp, 移動量, 移動量);
241:             }
242:           select_sprite_h_invert (自機, 1);
243:           break;
244:         case 静止:
245:           game_abort ("自機移動処理異常");
246:           break;
247:         }
248:       前回移動方向 = 移動方向;
249:     }
250: }
251:
252: /* 弾キャラクタの移動処理を行う。もし消去されたら 1 を返す */
253: static int
254: 弾移動処理 (Sprite 対象)
255: {
256:   move_sprite_diff (対象, 0, -6);
257:   if (SPD_POS_Y (対象) < 12)
258:     return 1;
259:   else
260:     return 0;
261: }
262:
263: /* 弾の発生と移動を行う */
264: void
265: 弾発射移動 (void)
266: {
267:   int 弾数local;
268:
269:   if (弾発射 == 1)
270:     {
271:       Sprite 発生キャラ;
272:       /* Debug 05 09 */
273:       弾発射 = 0;
```

```
274:       /* 弾の数がMax以下ならキャラを発生させる */
275:       if (弾数 < MAX_BOM && !やられ自機)
276:         {
277:           int 空き;
278:           for (空き = 0; 空き < MAX_BOM; 空き++)
279:             if (!弾[空き])
280:               break;
281:           弾[空き] = 発生キャラ = dup_sprite (弾ダミー);
282:           move_sprite_abs (発生キャラ, SPD_POS_X (自機), SPD_POS_Y (自機) - 6);
283:
284:           select_sprite_pcg (発生キャラ, 蓄積/30);
285:           SPD_REGIST (発生キャラ) = 蓄積/3 + 1;
286:           弾発射 = 蓄積 = 0;
287:           display_sprite (蓄積sp, 非表示);
288:           select_sprite_color (発生キャラ, 3);
289:           /* 表示を行う */
290:           display_sprite (発生キャラ, 表示);
291:           /* 寿命を初期化しておく */
292:           SPD_WORK0 (発生キャラ) = C_弾;
293:           弾数++;
294:         }
295:     }
296:   for (弾数local = 0; 弾数local < MAX_BOM; 弾数local++)
297:     if (弾[弾数local] && 弾移動処理 (弾[弾数local]))
298:       消去記録 (弾[弾数local], 弾数local);
299: }
```

▶5行〜 79行

```
enum 移動方向 方向読取 (void);
```

ジョイスティックから現在の状態を読み取って，弾の発射の処理や自機の移動方向の決定を行っています。まず，**GCC** 拡張の **asm** 文を使って，IOCS コールでジョイスティックの現在の状態を読み出しています。

ゲーム終了側のトリガスイッチが押されていれば，ゲーム終了を示すフラグ終を 1 にします。次に，自機が破壊されていないかをチェックします。破壊されていたら，ジョイスティックの状態に関係なく静止状態を返します。こうしておかないと，自機が破壊されているのにジョイスティックで動かせてしまうといった「おマヌケ状態」になります。

弾の発射トリガスイッチが押されていれば，前回の読み取りのときに押されていたかどうかを判断します。押されていれば「溜め撃ち」状態なので，スプライトのアニメーション処理を行います。アニメーションは，**static** なカウンタアニメの値を使って，4 種類のアニメパターンを順次表示します。このジョイスティック読み取りは，約 1/60 秒に 1 回通

過しますので，毎回アニメーションパターンを変更していると，変更速度が速すぎて妙な表示に見えてしまいます。そこで，この変数を2ビット右シフトして，適当な速度に落として表示させます。定数割り算をシフトで代行させている悪い例ですが，ここでは誤解を招くことはないと思われるので，このままにしてあります。

もし弾の発射トリガスイッチが離されていた場合には，前回の読み取りで押されていたかどうかを判断します。押されていれば，弾が発射された状態に移行します。前回離された状態にあれば，特に行う処理はありません。

弾の発射関係の処理が終わったら，自機の移動方向を判断し，その方向を返して，ジョイスティックの読み取り処理を終了します。

▸81行～250行

```
void 自機移動処理 (void);
```

自機の移動処理を行う関数です。スプライトの移動については，若干注意すべき点があります。この関数も垂直帰線期間周期で実行されますが，上下左右に動く場合と斜め移動を行う場合とでは，単位時間あたりの見た目の動く量が異なってくるという点です。直角二等辺三角形の斜辺距離だけ移動するのが斜め移動なので，上下左右の移動に比べると約1.4倍速い移動になります。そこで，斜めに連続移動を行う場合は，1回ごとの移動量を2/3にして，ほぼ同じ速度で移動するように移動量を調節してあります。

また，垂直帰線期間周期で1ドット移動する速度は，普通のゲームで自機を移動する速度としては遅すぎます。ですが，毎回2ドット単位で移動すると「パッドたたき」(自機をドット単位で移動させる)で移動できないので，プレイヤーに余計なストレスを感じさせます。そこで，前回の移動方向を記録しておいて，前回と異なった移動を行う場合は，まず1ドット単位で移動処理を行い，前回と同じ方向に移動する場合は，移動量を増やすような処理を行います。移動と同時に，自機はアニメーション処理を行いますので，そのための処理も同時に行います。

この関数では，自機の移動処理だけでなく，自機が敵に衝突してダメージを受けた場合の,「無敵時間」の時間計測処理とスプライトパレットの復帰処理も行っています。自機が当たり判定で敵との接触を起こした場合は，当たり判定の関数が自機のスプライトパレットを変更すると同時に，自機のスプライト構造体ワークに無敵時間を設定します。この設定時間を計測して，規定時間が経過したら，スプライトパレットを規定の色に戻しています。

▸252行～261行

```
static int 弾移動処理 (Sprite 対象);
```

Sprite 対象で与えられた自機が発射した弾のスプライトを，規定量だけ移動させる処理を行います。移動後，画面から外にはみ出す場合は'1'を返して，呼び出し元に通知し

ます。

▶263行～299行

```
void 弾発射移動 (void);
```

自機の弾の発射と，画面全体の弾の移動を行う関数です。まず弾の発射フラグを参照して，弾が発射されたら，画面全体の弾の総数が規定数以下の場合に弾の発射処理を行います。弾の発射のときは,「溜め撃ち」の蓄積時間を参照して，弾のスプライトパターンを選択します。

次に，先ほどの関数弾移動処理 ()で画面全体の弾について処理を行い，さらに画面の外にはみ出した弾について，スプライトの消去処理を行う関数消去記録 ()で画面からの消去を登録しておきます。

この自機移動処理を行う部分では，ジョイスティックでしか操作できないという手抜きをしています。普通のゲームとしてみると，この状態では「欠陥商品」です。キーボードも操作の対象にするのは比較的容易です。ご自分で改造してみてください。また，自機の移動速度を変更できるようにするのも，現在のゲームの一般的な形です。これもトライしてみてください。

■ 4.7 敵キャラクタ管理

このゲームでは，敵のキャラクタはたった2種類しかありません。1つは，球の形で膨らんだり縮んだりしながら自機の x 座標にフラフラと寄ってくるタイプ，もう1つは，ある範囲で自機の x 座標に一致したら，急速に変形して突進してくるタイプです。普通にスプライトゲームを作成すると，どうしてもキャラクタを動かすことに夢中になってしまって，キャラクタのアニメーション処理が怠りがちになります。

アニメーション処理は，データ作成だけでなく，プログラム上もめんどうなので，つい手を抜きたくなりますが，見た目の説得力はアニメ処理のあるなしでまったく違って見えるので，凝った処理を行いたい部分でもあります。このサンプルゲームのようなつまらないゲームでも，そこそこゲームらしく見えるのは，自機のアニメーションや敵キャラクタのアニメーションをいちおう行ってあるせいでもあります。

List 4.6が敵キャラクタの移動処理を行うプログラムです。

List 4.6 enemy.c

```
1:
2: #include "game.h"
3:
4: /*
```

```
 5:   敵キャラクタの移動処理を行う
 6:   もし消去されたら 1 を返す
 7: */
 8:
 9: static int
10: 敵移動処理 (Sprite 対象)
11: {
12:   switch (SPD_WORK0 (対象))
13:     {
14:     case C_敵0:
15:       SPD_WORK2 (対象)++;
16:       if (SPD_WORK2 (対象) == 16)
17:         select_sprite_pcg (対象, 1);
18:       else if (SPD_WORK2 (対象) == 32)
19:         select_sprite_pcg (対象, 2);
20:       else if (SPD_WORK2 (対象) == 48)
21:         {
22:           SPD_WORK2 (対象) = 0;
23:           select_sprite_pcg (対象, 0);
24:         }
25:       SPD_WORK3 (対象)++;
26:       if (SPD_WORK3 (対象) == 6)
27:         {
28:           SPD_WORK3 (対象) = 0;
29:           if (SPD_POS_X (自機) == SPD_POS_X (対象))
30:             move_sprite_diff (対象, 0, 1);
31:           else if (SPD_POS_X (自機) > SPD_POS_X (対象))
32:             move_sprite_diff (対象, 1, 1);
33:           else
34:             move_sprite_diff (対象, -1, 1);
35:         }
36:       else
37:         move_sprite_diff (対象, 0, 1);
38:       if (SPD_POS_Y (対象) > 273)
39:         return 1;
40:       else
41:         return 0;
42:       break;
43:
44:     case C_敵1:
45:       if (SPD_WORK2 (対象) == 0)
46:         if (SPD_POS_X (自機) == SPD_POS_X (対象))
47:           move_sprite_diff (対象, 0, 2);
48:         else if (SPD_POS_X (自機) > SPD_POS_X (対象))
49:           move_sprite_diff (対象, 1, 2);
50:         else
```

```
51:           move_sprite_diff (対象, -1, 2);
52:       else if (SPD_WORK2 (対象) > 4 * 8)
53:          move_sprite_diff (対象, 0, 6);
54:       else
55:         select_sprite_pcg (対象, (++SPD_WORK2 (対象)) / 8);
56:
57:       if (SPD_WORK2 (対象) == 0
58:           && 絶対値 (SPD_POS_X (対象), SPD_POS_X (自機)) < 8)
59:         select_sprite_pcg (対象, (++SPD_WORK2 (対象)) / 8);
60:
61:       if (SPD_POS_Y (対象) > 273)
62:         return 1;
63:       else
64:         return 0;
65:       break;
66:     default:
67:       game_abort ("スプライトタイプ異常");
68:       break;
69:     }
70: }
71:
72: /*
73:   敵キャラクタをランダムX位置に発生させ移動も行う
74: */
75: void
76: 敵キャラ発生移動 (void)
77: {
78:   static int 前回時間;
79:   Sprite 発生キャラ;
80:   int キャラ数;
81:
82:   /* 敵の発生時間間隔をランクに応じて変える */
83:   if (経過時間 - 前回時間 > MAX_NUM + 2 - ランク)
84:     {
85:       前回時間 = 経過時間;
86:
87:       /* 敵の数が現在のランク以下ならキャラを発生させる */
88:       if (敵数 < ランク)
89:         {
90:
91:           int 空き;
92:           Sprite type;
93:           for (空き = 0; 空き < MAX_NUM; 空き++)
94:             if (敵[空き] == 0)
95:               break;
96:           if (空き == MAX_NUM)
```

```
 97:             game_abort ("異常発生  キャラクタ発生");
 98:
 99:           敵[空き] = 発生キャラ =
100:             dup_sprite (type = (unsigned) rand ()< 0x2000
                                 ? タイプB :  タイプA);
101:
102:           /* いい加減なランダム位置に初期化する */
103:           move_sprite_abs (発生キャラ, (rand ()% 200) + 16, 0);
104:
105:           /* 敵キャラパターンをセレクト */
106:           select_sprite_pcg (発生キャラ, 0);
107:
108:           /* パレットを選ぶ  */
109:           select_sprite_color (発生キャラ, 3);
110:
111:           /* 表示を行う */
112:           display_sprite (発生キャラ, 表示);
113:
114:           /* カタサを設定する。大きくするとカタくなる */
115:           SPD_REGIST (発生キャラ) = type == タイプA ? 20 :  40;
116:
117:           /* 寿命を初期化しておく */
118:           SPD_LIFE (発生キャラ) = 256;
119:
120:           SPD_WORK0 (発生キャラ) = type == タイプA
                                    ? (int) C_敵0 :  (int) C_敵1;
121:           SPD_WORK1 (発生キャラ) = 0;
122:           SPD_WORK2 (発生キャラ) = 0;
123:           SPD_WORK3 (発生キャラ) = 0;
124:
125:           敵数++;
126:         }
127:     }
128:
129:   /* 各敵について移動処理を行う */
130:   for (キャラ数 = 0; キャラ数 < MAX_NUM; キャラ数++)
131:     {
132:       Sprite 対象;
133:       if (対象 = 敵[キャラ数])
134:         {
135:           if (SPD_WORK1 (対象))
136:             if (--SPD_WORK1 (対象) == 0)
137:               select_sprite_color (対象, 3);
138:           if (敵移動処理 (対象))
139:             消去記録 (対象, キャラ数);
140:         }
```

```
141:     }
142: }
```

▶ 9 行～70 行

```
static int 敵移動処理 (Sprite 対象);
```

Sprite 対象の敵キャラクタについて移動処理を行います。Sprite 対象の汎用ワーク SPD_WORK0(対象)には，そのスプライトのキャラクタの種類が格納されています。その種類に応じて，移動処理を **switch** 文で切り分けています。キャラクタの種類を増やすのは簡単で，このワークに入るスプライトの種類を増やして，その種類のキャラクタの移動方法をプログラムすればそれで終わりです。もちろん，スプライトのデータも作成したほうがよいのですが，スプライトの形に移動方法が依存しているわけではありませんから，同じ表示をするキャラクタが別の動きをするようにもプログラムできます。

この関数では，キャラクタが移動した後，画面からはみ出す場合は '1'，そうでない場合は '0' を返します。

▶ 75 行～142 行

```
void 敵キャラ発生移動 (void);
```

画面全体の敵キャラクタの発生と移動を行います。まず，現在のランクと今までの経過時間によって，キャラクタを新しく発生させるかどうかを決定します。キャラクタ発生条件を満たした場合は，乱数で発生するキャラクタや x 座標を決定して，スプライトを作成します。

次に，画面上の敵キャラクタについて移動処理を行います。このとき，画面の外にはみ出したキャラクタは，消去記録 ()を呼び出して画面から消去します。

この敵キャラ発生移動 ()部分は，いくらでも改造の余地があります。このゲームでは「板つきキャラクタ」が存在していないのが「欠陥」の１つですが，追加できないようなことはないでしょう。

■ 4.8　当たり判定とスプライトの消去

ここでは，ゲーム全体の当たり判定処理とスプライトの消去を行います。このゲームの古いバージョンでは，スプライトの消去は個々に行っていましたが，スプライトを消去するには，

1. いったんスプライトを非表示状態にする。

2. 垂直帰線期間割り込みの処理を一度経過する。
3. スプライトを削除する。

といった段階を経なければならないため，個別に行っていてはその処理のコーディングの手間がたいへんなので，一括して消去を管理する関数を作成しました。

当たり判定については，現バージョンではかなりいい加減な方法を使っています。小さなキャラクタとの当たり判定は，キチンと行わないと「見た目が変」になりますが，このゲームのキャラクタではそれほど違和感がなかったので，そのままにしてあります。

List 4.7が，当たり判定とスプライト消去を行う部分のソースリストです。

List 4.7 clash.c

```
 1:
 2: #include "game.h"
 3:
 4: static Sprite 消去バッファ[128];
 5: static int 消去カウンタ;
 6:
 7: void 消去記録 (Sprite 対象, int 番号)
 8: {
 9:   display_sprite (対象, 非表示);
10:   switch (SPD_WORK0 (対象))
11:     {
12:     case C_自機:
13:       game_abort ("自機を消去しています");
14:       break;
15:     case C_弾:
16:       if (対象 == 弾[番号])
17:         {
18:           弾[番号] = 0;
19:           弾数--;
20:         }
21:       else
22:         game_abort ("消去キャラクタ異常 弾");
23:       break;
24:     case C_敵0:
25:     case C_敵1:
26:       if (対象 == 敵[番号])
27:         {
28:           敵[番号] = 0;
29:           敵数--;
30:         }
31:       else
32:         game_abort ("消去キャラクタ異常 敵");
33:       break;
```

```
34:     case C_爆発0:
35:       if (対象 == 爆発[番号])
36:         {
37:           爆発[番号] = 0;
38:           爆発数--;
39:         }
40:       else
41:         game_abort ("消去キャラクタ異常 爆発");
42:       break;
43:     }
44:   消去バッファ[消去カウンタ++] = 対象;
45: }
46:
47: void 消去実行 (void)
48: {
49:   int i;
50:   for (i = 0; i < 消去カウンタ; i++)
51:     delete_sprite (消去バッファ[i]);
52:   消去カウンタ = 0;
53: }
54:
55: /* 爆発パターンを生成する */
56: static void
57: クラッシュ (Sprite 対象)
58: {
59:   int 空き;
60:   Sprite 発生キャラ;
61:   for (空き = 0; 空き < MAX_NUM; 空き++)
62:     if (爆発[空き] == 0)
63:       break;
64:   if (空き == MAX_NUM)
65:     game_abort ("異常 爆発キャラクタ発生");
66:   爆発[空き] = 発生キャラ = dup_sprite (爆発0ダミー);
67:   move_sprite_abs (発生キャラ, SPD_POS_X (対象), SPD_POS_Y (対象));
68:   select_sprite_pcg (発生キャラ, 0);
69:   select_sprite_color (発生キャラ, 3);
70:   display_sprite (発生キャラ, 表示);
71:   SPD_WORK0 (発生キャラ) = C_爆発0;
72:   SPD_LIFE (発生キャラ) = 0;
73:   爆発数++;
74: }
75:
76: static void
77: 爆発処理 (void)
78: {
79:   int i;
```

```
  for (i = 0; i < MAX_NUM; i++)
    if (爆発[i])
      {
              if (++SPD_LIFE (爆発[i]) > 23)
                消去記録 (爆発[i], i);
              else
                {
            select_sprite_pcg (爆発[i], SPD_LIFE (爆発[i]) / 8);
            select_sprite_h_invert (爆発[i], 1 & SPD_LIFE (爆発[i]));
            select_sprite_v_invert (爆発[i], 2 & SPD_LIFE (爆発[i]));
          }
      }
}

/* 弾と敵との当たり判定を行う。かなり手抜きかもしれない。
   弾を消去すべきときには1を返す */

static int
当たり判定サブ0 (Sprite 弾)
{
  int 敵番号;
  int 座標x = SPD_POS_X (弾);
  int 座標y = SPD_POS_Y (弾);

  for (敵番号 = 0; 敵番号 < MAX_NUM; 敵番号++)
    {
      Sprite 対象;
      if ((対象 = 敵[敵番号]) && SPD_LIFE (対象) > 0)
        {
          int 敵座標x = SPD_POS_X (対象);
          if (絶対値 (座標x, 敵座標x) < 10)
            {
              int 敵座標y = SPD_POS_Y (対象);
              if (絶対値 (座標y, 敵座標y) < 10)
                {
                  int ダメージ = SPD_REGIST (弾);
#ifdef NON_LIKE_R_TYPE
                  SPD_REGIST (弾) -= SPD_REGIST (対象);
#endif
                  /* 敵キャラの処理 */
                  SPD_REGIST (対象)-= ダメージ;
                  if (SPD_REGIST (対象) < 0)
                    {
                      クラッシュ (対象);
                      消去記録 (対象, 敵番号);
                      スコア++;
```

```
126:                     }
127:                   else
128:                     {
129:                       SPD_WORK1 (対象) = 3;
130:                       select_sprite_color (対象, 2);
131: #ifndef NON_LIKE_R_TYPE
132:                       return 1;
133: #endif
134:                     }
135:                 }
136: #ifdef NON_LIKE_R_TYPE
137:               if (SPD_REGIST (弾) < 0)
138:                 return 1;
139:               else
140:                 select_sprite_pcg (弾, SPD_REGIST (弾)/10);
141: #endif
142:             }
143:         }
144:     }
145:   return 0;
146: }
147:
148: static int
149: 当たり判定サブ1 (void)
150: {
151:   int 敵番号;
152:   int 座標x = SPD_POS_X (自機);
153:   int 座標y = SPD_POS_Y (自機);
154:   int 当たり = 0;
155:
156:   for (敵番号 = 0; 敵番号 < MAX_NUM; 敵番号++)
157:     {
158:       Sprite 対象;
159:       if ((対象 = 敵[敵番号]) && SPD_LIFE (対象) > 0)
160:         {
161:           int 敵座標x = SPD_POS_X (対象);
162:           if (絶対値 (座標x, 敵座標x) < 10)
163:             {
164:               int 敵座標y = SPD_POS_Y (対象);
165:               if (絶対値 (座標y, 敵座標y) < 10)
166:                 {
167:                   SPD_REGIST (対象)--;
168:                   if (SPD_REGIST (対象) < 0)
169:                     {
170:                       クラッシュ (対象);
171:                       消去記録 (対象, 敵番号);
```

```
172:                         スコア++;
173:                       }
174:                     else
175:                       {
176:                         SPD_WORK1 (対象) = 3;
177:                         select_sprite_color (対象, 2);
178:                       }
179:                     当たり = 1;
180:                   }
181:               }
182:           }
183:       }
184:     return 当たり;
185: }
186:
187: /* 当たり判定を行う */
188: void
189: 当たり判定 (void)
190: {
191:   int 弾数local;
192:   for (弾数local = 0; 弾数local < MAX_BOM; 弾数local++)
193:     if (弾[弾数local] && 当たり判定サブ0 (弾[弾数local]))
194:       消去記録 (弾[弾数local], 弾数local);
195:
196:   /* 自機に当たり判定を行う */
197:   if (やられ自機)
198:     {
199:       SPD_LIFE (やられ自機)++;
200:       if (SPD_LIFE (やられ自機) == 30)
201:         select_sprite_pcg (やられ自機, 1);
202:       if (SPD_LIFE (やられ自機) == 60)
203:         select_sprite_pcg (やられ自機, 2);
204:       if (SPD_LIFE (やられ自機) == 120)
205:         終 = 1;
206:     }
207:   else if (SPD_WORK1 (自機) == 0 && 当たり判定サブ1 ())
208:     {
209:       select_sprite_color (自機, 2);
210:       SPD_WORK1 (自機) = 16;
211:       if ((--SPD_REGIST (自機)) == 0)
212:         {
213:           やられ自機 = dup_sprite (爆発0ダミー);
214:           move_sprite_abs (やられ自機, SPD_POS_X (自機), SPD_POS_Y (自機));
215:           select_sprite_pcg (やられ自機, 0);
216:           SPD_LIFE (やられ自機) = 0;
217:           select_sprite_color (やられ自機, 3);
```

```
218:            display_sprite (やられ自機, 表示);
219:            display_sprite (自機, 非表示);
220:          }
221:      }
222:    爆発処理 ();
223: }
```

▶4行〜5行

```
static Sprite 消去バッファ[128];
static int 消去カウンタ;
```

この変数は，スプライトを一括消去するためのバッファと，その数を数えるカウンタです。関数消去記録 ()でこの消去バッファに消去すべきスプライトを保持しておいて，垂直帰線期間割り込みが実行された後，関数消去実行 ()で実際にスプライトを削除します。

▶7行〜 45行

```
void 消去記録 (Sprite 対象, int 番号);
```

敵キャラクタや弾のスプライトを消去登録する関数です。敵キャラクタや弾のスプライトは配列変数に保持されていますので，そのインデックス番号を番号で受け取っています。本来，この**対象**はこの配列のどこかに格納されているので，このインデックスを受け取らなくてもいいのですが，いちいち検索していては処理時間がもったいないので，引数で受け取っています。

内部チェックとして，たしかに対象がそのインデックス番号の配列に格納されているかどうかについては確認しています。

▶47行〜53行

```
void 消去実行 (void);
```

消去バッファに格納されているスプライトを削除します。削除する前には必ず「非表示状態」にして，垂直帰線期間割り込み処理を経過しなければならないので，消去実行 ()が独立した関数として用意してあります。

▶56行〜74行

```
static void クラッシュ (Sprite 対象);
```

Sprite 対象の座標位置に，爆発のパターンスプライトを生成して表示します。ここでは，次のことに注意しなければいけません。この時点でスプライトスクロールレジスタが割り当てられます。対象のスプライトにも，スプライトスクロールレジスタが割り当てられています。この対象は非表示状態にされるとはいえ，ハードウェアは開放されていませんので，非常に多数のスプライトを表示していた場合には，スプライトスクロールレジスタが不足する可能性があります。

▶76 行～92 行

```
static void 爆発処理 (void);
```

爆発パターンのスプライトのアニメーション処理を行います。爆発のスプライトは移動しないので，処理は簡単なものになっています。

▶97 行～146 行

```
static int 当たり判定サブ0 (Sprite 弾);
```

Sprite 弾と全敵キャラクタとの当たり判定を行います。NON_LIKE_R_TYPEが**#define** されていない状態では，敵キャラクタのカタサが弾の威力より小さい場合，弾の威力は減少しないで貫通するようになっています。NON_LIKE_R_TYPEが**#define** されていると，弾の威力はキャラクタのカタサに応じて減少されます。

▶148 行～185 行

```
static int 当たり判定サブ1 (void);
```

自機と全敵キャラクタとの当たり判定を行います。自機と衝突したキャラクタも同時にダメージを受けて，破壊された場合には爆発パターンに変化します。

▶188 行～223 行

```
void 当たり判定 (void);
```

画面全体の当たり判定を行います。まず，自機が発射した弾について当たり判定を行った後，自機について当たり判定を行います。ここでも，自機がすでに破壊された状態になっていた場合には，自機については当たり判定を行わずに，アニメーション処理だけを行っています。また，前回に自機が敵キャラクタと当たりになっている場合は，一定時間当たり判定をしないようになっています。

■ 4.9　背景の処理

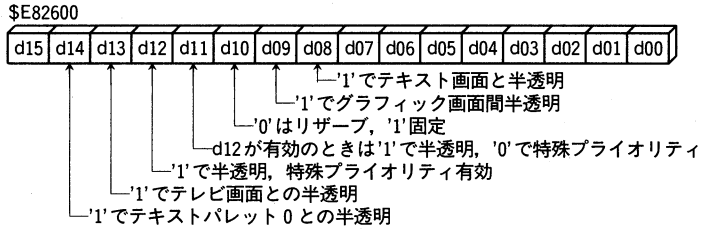
このゲームでは，グラフィック画面２面とバックグランド画面１面を使って，背景の表示を行っています。背景の処理は，グラフィック部分とバックグランドの部分とに分離されています。

4.9.1 ■　グラフィック画面の処理

List 4.8 (P.226) が，背景をスクロールさせる処理 map.c です。背景をただスクロールさせるだけでは芸がないので，背景のアニメーション処理によく使われるパレット書き換えによる芸を加えてあります。さらに，**X68000/X68030** のゲームで最近常套手段になっている，ラスタースクロールも実現してみました。

アクションゲームに限らず，ゲームではさまざまな特殊な映像処理効果を用いたビジュアルな演出が行われます。ラスタースクロールもその手法の１つです。特殊効果には，ハードウェアで行うものと，ソフトウェアで行うものがありますが，**X68000/X68030** では，ハードウェアでわりあい簡単に作れる特殊効果として「半透明機能」があります。

X68000/X68030 では，テキスト画面，グラフィック画面，スプライト&バックグランド画面と，たくさんの同時表示画面をもつことができます。さらに，グラフィック画面では，同時表示色数を制限することで，複数のグラフィック画面をもつことも可能になっています。半透明機能というのは，この多重化された画面を表示するときに使う映像演出効果機能です。Fig. 4.2 が半透明機能に用いられるレジスタの構成です。



●Fig. 4.2　パレットレジスタ

この半透明機能は，単純にいえば，ある画面の絵をセロハン紙に焼いて，それを他の画面に重ねてみるといった機能です。このセロハン紙に焼く部分を任意の範囲で指定できます。『付録ディスク』のゲームを動かしてみてください。「川」 のように流れて見える部分が半透明を使った部分です。この部分は，２枚のグラフィック画面を使って半透明を実現しています。しばらくゲームを続けていくと，画面全体が半透明状態になります。

半透明機能をゲームで生かすことは，プログラムテクニックとはほとんど無縁です。半透明にするためには，Fig. 4.2 に示したシステムポートに適切なデータを出力するだけでよく，後はハードがよろしく合成してくれるからです。重要なのは，半透明を生かすよう

なステージ構成とグラフィックデザインです。たとえば，ドラマでありがちな演出ですが，「稲妻がきらめくと一瞬ガラス窓の向こうが見える」といった演出効果を，半透明を使って簡単に実現できます。これについて，ちょっと考えてみましょう。

グラフィック画面を2画面もてるモードにして，優先順位の高いプレーンに，暗い色で窓とその周辺を描いておきます。また，優先順位の低い画面には，「一瞬見える背景」を描いておきます。これらの画面を重ね合わせると，背景は通常状態ではまったく見えない状態になります。ここで，半透明を使います。半透明を行う範囲は窓の暗い色の部分です。ここの色を表すデータを奇数のパレットにしておき，半透明にしない部分を偶数パレットで描画します。これは，半透明にする範囲をVRAMデータの最下位ビットで行うというのが，**X68000/X68030**の半透明機能の約束になっているからです。半透明機能を使うためには，奇数番号パレットと偶数番号パレットを一対にして，同じ色を設定しておくという制約があるのです。このため，**X68000/X68030**でグラフィック画面間で半透明を行う場合には，使える色数が128色に制限されます。もっとも，ゲームにおいては128色あれば十分と思いますが。このように窓のグラフィックを描画しておけば，後は簡単です。垂直帰線期間をみて適当な時間だけ，半透明機能を有効にしてやります。有効になっている時間だけ，通常は見えない，優先順位の低い背景のグラフィックが窓に見えるようになります。暗いセロハンを通して見るような具合になりますので，背景描画は明るい色で行うのがよいでしょう。

プログラム上は本当に簡単なのですが，データ作成の際にはうまく効果が上がるように，頭をひねらなければなりません。実際の話，このゲームで半透明を使った部分を作るのにかかった時間のほとんどが，グラフィックデータの再構成でした。

次はラスタースクロールです。ラスタースクロールというのは，CRTCが水平方向を順次描画している間に，スクロールレジスタを操作して特殊な映像効果を得る手法です。ラスタースクロールのアルゴリズム自体は単純です。ゲームにインプリメントするには，与えられたマップデータに基づいて決められた範囲内だけを，ラスタースクロールで「ゆらゆらさせる」ようにする必要があります。このゲームの背景は16×16ドットのタイルで画面を構成するように作られていますが，その背景の**属性**を保持するデータを扱う変数はありません。

通常はこの属性を使って，ラスタースクロールの範囲を決めたり，「当たり判定」のある部分やない部分を区別したりするのですが，さすがに時間的な余裕がないので，マップの属性を扱うようなデータを新しく作成したりせずに，ただ単にグラフィックデータの特定データが存在する部分を，ラスタースクロールを行う範囲だと認識するようにしました。「水が上から下へ流れる」ように見える部分が存在する範囲の，半透明で透けて見える背景を，ラスタースクロールさせるようにインプリメントすることを方針として設定しました。

指定ラスター位置での割り込みは，ゲームが始まった直後から常時行われるようになっています。この割り込みの登録は，utiles.cの画面関係初期化の関数によって行われます。初期化時点でマップデータを解析し，グラフィックの横ライン数の何本目を描画した位置からラスタースクロールを開始して，何本目でラスタースクロールを終了するかをあらか

じめ計算しておきます。繰り返しテストした結果，256 × 256 高解像度 31KHz 水平周波数では，おおむね 40 ラスターが表示開始位置で，そこから 512 ラスターで表示が終わるようにプログラムを行えばうまく動くようです。市販品のゲームのように何種類かの画面モードに対応するためには，各モードでのラスター開始位置等を確認して (本当は CRTC に設定する値から計算できるハズです)，モードに応じた処理を行うのが適切でしょう。

マップデータを解析した結果から実際のスクロール量を計測して，ラスタースクロール開始位置が画面に表示されたら「ラスタースクロール開始」のフラグをたてて，実際にラスタースクロールを行います。最初は，画面上方から中ほどまでがラスタースクロール範囲になります。スクロール範囲の画面上での下限が，ゲーム進行に応じて下方に移動していきます。これを管理している変数が`end_raster`です。画面全体がラスタースクロールを行うようになると，`end_raster`の値は実際には存在しないラスター位置を示すようになります。この値をそのまま設定すると「ハマリ」ます。存在しないラスター位置で割り込みを設定すると，二度と割り込まなくなってしまいます。私自身，最初はこれにやっぱりハマリました。そこで，最終ラスター位置の値をカット&トライで設定するという非科学的な方法を用いました。次に，マップが進行していって，ラスタースクロール終了位置が画面に表示されると，ラスタースクロール範囲の上限が画面下方に向かって移動するようになります。これを管理している変数が`start_raster`です。`start_raster`が最終ラスターを越えて移動した時点が，ラスタースクロールの終了点です。

また，この map.c では，背景のグラフィックデータを VRAM に転送するのに，ソフトウェアによる方法と，IOCS を呼び出して DMA(ダイレクトメモリアクセス) を用いる方法の 2 種類を選べるようにしてあります。DMA は，CPU の介在なしに I/O やメモリ間でデータの転送を行うためのハードウェアで，**X68000/X68030** では DMA 転送について「VRAM に対して行ってはならない」といった制限はありません。**X68000/X68030** では問題なく DMA 転送ができます。

IOCS による DMA 転送には，いくつかの種類がサポートされています。このゲームのグラフィックデータ構造が 16 × 16 ドットのタイル形式になっているので，ただ単にライブラリ関数の`memcpy()`を呼び出して転送するといった簡単な方法では，VRAM に転送はできません。このため，アレイチェインモードで転送を行っています。

アレイチェインモードでは，List 4.8 の 85 行〜89 行に出てくる

```
typedef struct
{
  unsigned char *addr;
  unsigned short count;
} array_chain;
```

のようなデータ構造を渡して転送を行います。`array_chain FOO.addr`には転送元あるいは転送先のアドレスを，`array_chain FOO.count`にはそのアドレスへの転送バイト数を入れてやります。IOCS では，この`array_chain`の配列の先頭ポインタを受け取って，そ

の配列数分だけ転送を行ってくれます。転送自体はハードウェアが行うので，転送が実際に終わらなくても呼び出し元に帰ってきます。転送のソースアドレス，ディステネーションアドレスの増減方法，転送方向などもすべて設定することができます。**XC** 付属の「プログラマーズ・マニュアル」の説明は少しわかりにくいので，ちょっとだけ説明してみましょう。

IOCS はデータの転送元，転送先に，C で書けば以下のようなデータを要求します。

```
int num = 転送数;
array_chain buf[転送数];
unsigned char *addr;
```

array_chain buf[]には，メンバに転送アドレスと転送バイト数をそれぞれ入れておきます。DMA 転送の転送方向は，addr→buf，buf→addrのどちらも可能です。

```
int i;
for (i = 0; i < num ; i++)
  {
    int c = buf[i].count;
    unsigned char *dest = addr;
    unsigned char *src  = buf[i].addr;
    while (c > 0)
     {
        if (direction == 0)
         *dest = *src;
        else
         *src = *dest;
        if (dest_inc_mode)
          dest++;
        else if (dest_dec_mode)
          dest--;
        if (src_inc_mode)
          src++;
        else if (src_dec_mode)
          src--;
     }
  }
```

ソフトウェアで記述すればこのようになる処理を，ハードウェアが自動的に行ってくれます。dest_inc_mode，etc... には，転送時にアドレスを更新するモードを指定してやります。データの転送方向はdirectionによって指定します。ハードウェアが行うのですから，非常に高速ですが，実は **X68030** では DMA コントローラより CPU の処理速度のほうが速くなっているために，DMA 転送よりソフトウェアで転送するほうが高速になります。

List 4.8 map.c

```
 1: #include "game.h"
 2:
 3: /* #define MAP_IS_FILE */
 4:
 5: #ifndef MAP_IS_FILE
 6: #include "mapdata.h"
 7: #endif
 8:
 9: #include "sindata.h"
10:
11: static int 面初期化 = 1;
12:
13: /* VRAM 書き込みアドレス */
14: static unsigned char *vram0 = (unsigned char *) (0xc80000 - 0x400 * 272);
15: static unsigned char *vram1 = (unsigned char *) (0xd00000 - 0x400 * 272);
16:
17: static int now_counter;
18: static unsigned char *map_data_ptr0;
19: static unsigned char *map_data_ptr1;
20:
21: #define RASTER_MIN 40
22: #define RASTER_MAX RASTER_MIN + 512
23: #define RASTER_STEP 32
24:
25: static int *raster_map;
26: static int scroll_counter;
27: static short do_raster;
28: static short start_raster = RASTER_MIN;
29: static short end_raster = RASTER_MIN;
30: static int index_val;
31: static CRTC_REG raster_val;
32:
33: /* 16*16 矩形を指定された VRAM アドレスに書き込み */
34:
35: #ifdef SOFT_TRANS
36: static void
37: write_vram (unsigned short *vr, unsigned short *dat)
38: {
39:   int i;
40:   for (i = 0; i < 16; i++)
41:     {
42:       unsigned short *p = vr;
43:       *p++ = *dat++;
```

```
44:       *p++ = *dat++;
45:       *p++ = *dat++;
46:       *p++ = *dat++;
47:       *p++ = *dat++;
48:       *p++ = *dat++;
49:       *p++ = *dat++;
50:       *p++ = *dat++;
51:       *p++ = *dat++;
52:       *p++ = *dat++;
53:       *p++ = *dat++;
54:       *p++ = *dat++;
55:       *p++ = *dat++;
56:       *p++ = *dat++;
57:       *p++ = *dat++;
58:       *p++ = *dat++;
59:       vr += 0x200;
60:     }
61: }
62: #else
63: #define DMA_SENS                                                \
64: ({ register int d0 asm ("d0");                                  \
65:    d0 = 0x8d;                                                   \
66:    asm volatile ("trap #15":"=d"(d0):"0"(d0));                  \
67:    d0;                                                          \
68: })
69:
70: #define DO_DMA(BUF,DAT)                                         \
71: do {  register int d0 asm ("d0");                               \
72:       register int d1 asm ("d1");                               \
73:       register int d2 asm ("d2");                               \
74:       register void *a1 asm ("a1");                             \
75:       register void *a2 asm ("a2");                             \
76:       d0 = 0x8b;                                                \
77:       d1 = 0b1_000_01_01;                                       \
78:       d2 = 16;                                                  \
79:       a1 = (BUF); a2 = (DAT);                                   \
80:       asm (                                                     \
81:       "trap #15"::"d"(d0),"d"(d1),"d"(d2),"a"(a1),"a"(a2)       \
82:           );                                                    \
83: } while (0)
84:
85: typedef struct
86: {
87:   unsigned char  *addr;
88:   unsigned short count;
89: } array_chain;
```

```
 90:
 91: static void
 92: write_vram (unsigned char *vr, unsigned char *dat)
 93: {
 94:   int i;
 95:   static array_chain buf[16];
 96:   while (DMA_SENS)
 97:     ;
 98:   for (i = 0; i < 16; i++)
 99:     {
100:       buf[i].addr =  vr;
101:       buf[i].count = 32;
102:       vr += 0x400;
103:     }
104:   DO_DMA (buf, dat);
105: }
106:
107: #endif
108:
109: void write_one_block0 ()
110: {
111:   static int write_count;
112:   write_vram (vram0, (unsigned char *)map_data[map_data_ptr0[now_counter]]);
113:   vram0 += 32;
114:   write_count ++;
115:   if (write_count == H_BLOCK_NUM)
116:     {
117:       write_count = 0;
118:       vram0 -= ((H_BLOCK_NUM * 32) + 0x400 * 16);
119:       if (vram0 < (unsigned char *) 0xc00000)
120:         vram0 =  (unsigned char *) (0xc80000 - 0x400 * 16);
121:     }
122: }
123:
124: void write_one_block1 ()
125: {
126:   static int write_count;
127:   write_vram (vram1, (unsigned char *)map_data[map_data_ptr1[now_counter]]);
128:   vram1 += 32;
129:   write_count ++;
130:   if (write_count == H_BLOCK_NUM)
131:     {
132:       write_count = 0;
133:       vram1 -= ((H_BLOCK_NUM * 32) + 0x400 * 16);
134:       if (vram1 < (unsigned char *) 0xc80000)
135:         vram1 =  (unsigned char *) (0xd00000 - 0x400 * 16);
```

```
136:     }
137: }
138:
139: void ロードマップ (int 面)
140: {
141:   static char file_name[] = "han0.dat";
142:   char name[128];
143:   unsigned char gbuf[256];
144:   FILE *file;
145:   file_name[3] += 面;
146:   strcpy (name, LOAD_DIR);
147:   strcat (name, file_name);
148:   file = fopen (name, "rb");
149:   if (!file)
150:     game_abort ("マップファイルがありません");
151:   fread (g_palet, sizeof (g_palet), sizeof (char), file);
152:   if (!map_data[0])
153:     {
154:       int i;
155:       for (i = 0; i < 256; i++)
156:         map_data[i] = xmalloc (sizeof (short) * 256);
157:     }
158:   {
159:     int i,j;
160:     unsigned short *p;
161:     unsigned char *q;
162:     for (i = 0; i < 256; i++)
163:       {
164:         fread (gbuf, sizeof (gbuf), sizeof (char), file);
165:         p = map_data[i];
166:         q = gbuf;
167:         for (j = 0; j < 256; j++)
168:           *p ++ = *q ++;
169:       }
170:   }
171:   fclose (file);
172: #ifdef MAP_IS_FILE
173:   strcpy (name, LOAD_DIR);
174:   file_name[5] = 'd';
175:   strcat (name, file_name);
176:   file_name[5] = 'p';
177:   file = fopen (name, "rb");
178:   if (!file)
179:     game_abort ("マップデータファイルがありません");
180:   map_data_ptr0 = xmalloc (now_counter = filelength (fileno (file)));
181:   fread (map_data_ptr0, now_counter, sizeof (char), file);
```

```
182:   fclose (file);
183: #else
184:   map_data_ptr0 = MAP_DATA_PTR0;
185:   map_data_ptr1 = MAP_DATA_PTR1;
186:   if (sizeof (MAP_DATA_PTR0) != sizeof (MAP_DATA_PTR1))
187:     game_abort ("マップデータ異常");
188:   now_counter = sizeof (MAP_DATA_PTR0);
189:   {
190:     int i,j;
191:     unsigned char buf[H_BLOCK_NUM];
192:     unsigned char *p;
193:     p = map_data_ptr0;
194:     for (j = 0; j < sizeof (MAP_DATA_PTR0)/ H_BLOCK_NUM; j++)
195:       {
196:         for (i = 0; i < H_BLOCK_NUM; i++)
197:           buf[i] = *p++;
198:         for (i = 0; i < H_BLOCK_NUM; i++)
199:           *--p = buf[i];
200:         p += H_BLOCK_NUM;
201:       }
202:     p = map_data_ptr1;
203:     for (j = 0; j < sizeof (MAP_DATA_PTR1)/ H_BLOCK_NUM; j++)
204:       {
205:         for (i = 0; i < H_BLOCK_NUM; i++)
206:           buf[i] = *p++;
207:         for (i = 0; i < H_BLOCK_NUM; i++)
208:           *--p = buf[i];
209:         p += H_BLOCK_NUM;
210:       }
211:   }
212: #endif
213:   {
214:     int i;
215:     int counter;
216:     raster_map =
217:       xmalloc (sizeof (int) * (now_counter/ H_BLOCK_NUM + 1));
218:     counter = 0;
219:     for (i = 16; i < now_counter / H_BLOCK_NUM; i++)
220:       {
221:               unsigned char dat =
222:                  map_data_ptr0[sizeof (MAP_DATA_PTR0) - 1 - i * H_BLOCK_NUM];
223:               counter += 32;
224:         if (dat == RASTER_DATA)
225:           raster_map[i - 16] = counter;
226:         else
227:           raster_map[i - 16] = 0;
```

```
228:       }
229:   }
230:   面初期化 = 1;
231: }
232:
233:
234: void
235: raster_scroll ()
236: {
237:   static short count;
238:   static short last;
239:   static short next = RASTER_MIN;
240:   /* ラスター位置指定レジスタアドレス */
241:   short *next_inter = (short *) 0xe80012;
242:   /* スクロールレジスタアドレス */
243:   if (do_raster)
244:     {
245:       /* スクロールレジスタアドレス */
246:       CRTC_REG *crtc = (CRTC_REG *) 0xe80018;
247:       short difs = sin_tbl[index_val][count];
248:       next += RASTER_STEP;
249:       if (!last && end_raster < RASTER_MAX && end_raster < next)
250:         {
251:           next = end_raster;
252:           last = 1;
253:         }
254:       else if (next > RASTER_MAX)
255:         last = 1;
256:       if (last)
257:         {
258:           if (start_raster > RASTER_MIN && start_raster < RASTER_MAX)
259:             next = start_raster;
260:           else
261:             next = RASTER_MIN;
262:           last = count = difs = 0;
263:         }
264:       raster_val.sc2_x_reg = raster_val.sc3_x_reg = scroll_data.sc2_x_reg + difs;
265:       *crtc = raster_val;
266:     }
267:   else
268:     next = RASTER_MIN;
269:   /* 次回割り込み位置を指定する */
270:   *next_inter = next;
271:   count ++;
272:   if (count > 32)
273:     count = 0;
```

```
274:   IRTE ();
275: }
276:
277: void
278: 背景移動処理 (void)
279: {
280:   static int cont = 0;
281:   static int cont_block = 0;
282:   static int pal_count;
283:   if (!面初期化)
284:     {
285:       if (!面クリア)
286:         {
287:           int i;
288:           int j = PAL_C_MIN + (cont >> 2) % (PAL_C_MAX - PAL_C_MIN);
289:           for (i = 0; i < PAL_C_MAX - PAL_C_MIN; i++)
290:             {
291:               c_palet[i] = g_palet[j];
292:               if (++j > PAL_C_MAX)
293:                 j = PAL_C_MIN;
294:             }
295:           if (cont_block < H_BLOCK_NUM)
296:             {
297:               now_counter --;
298:               write_one_block0 ();
299:               write_one_block1 ();
300:             }
301:           if (cont & 1)
302:             {
303:               scroll_data.sc0_y_reg --;
304:               scroll_data.sc1_y_reg --;
305:               scroll_data.sc2_y_reg --;
306:               scroll_data.sc3_y_reg --;
307:               raster_val = scroll_data;
308:               scroll_counter +=2;
309:               if (++index_val > 31)
310:                 index_val = 0;
311:               if (raster_map[scroll_counter/32])
312:                 end_raster += 2;
313:               if (end_raster > RASTER_MIN + RASTER_STEP)
314:                 do_raster = 1;
315:               if (end_raster > RASTER_MIN && raster_map[scroll_counter/32] == 0)
316:                 start_raster += 2;
317:               if (start_raster > RASTER_MAX - RASTER_STEP)
318:                 {
319:                   do_raster = 0;
```

```
320:                   end_raster = start_raster = RASTER_MIN;
321:                 }
322:             }
323:           cont ++;
324:           if (++cont_block > 31)
325:             cont_block = 0;
326:         }
327:       if (now_counter == 0)
328:         面クリア = 1;
329:       return;
330:     }
331:
332:   if (面初期化 == 1)
333:     {
334:       int i;
335:       面初期化 =  2;
336:       for (i = 0; i < 256; i++)
337:         GPALET (i, 0);
338:       for (i = 0; i < H_BLOCK_NUM * 16 + H_BLOCK_NUM; i++)
339:         {
340:           now_counter --;
341:           write_one_block0 ();
342:           write_one_block1 ();
343:         }
344:       half_def_data = 0b1_0_0_1_1_1_1_0_010_0_1111;
345:       use_half_tone = 1;
346:     }
347:   else if (面初期化 < (12 * 32 + 2))
348:     {
349:       面初期化 ++;
350:       if (面初期化 % 12 == 2)
351:         {
352:           int i,pal;
353:           pal_count++;
354:           for (i = 0; i < 256; i++)
355:             {
356:               pal = 0;
357:               if ((g_palet[i] >> 11) > pal_count)
358:                 pal |= pal_count << 11;
359:               else
360:                 pal |= (g_palet[i] & 0b11111_00000_00000_0);
361:               if (((g_palet[i] >> 6) & 0b11111) > pal_count)
362:                 pal |= pal_count << 6;
363:               else
364:                 pal |= (g_palet[i] & 0b00000_11111_00000_0);
365:               if (((g_palet[i] >> 1) & 0b11111) > pal_count)
```

```
366:                 pal |= pal_count << 1;
367:               else
368:                 pal |= (g_palet[i] & 0b00000_00000_11111_0);
369:               palet_buf[i] = pal;
370:             }
371:           palet_def = 1;
372:         }
373:     }
374:   else
375:     {
376:       ゲーム開始 = 1;
377:       pal_count = 0;
378:       面初期化 = 0;
379:     }
380: }
```

それでは，実際のソースリスト List 4.8 について説明しましょう。

▶5 行〜6 行

この**#include** で画面のマップをソースに取り込んでいます。大きなマップをコンパイルするには，大量のメモリを消費します。本格的なゲームに仕上げるには，MAP_IS_FILEを**#define** して，ソースと独立してマップを作成するべきでしょう。

▶9 行

ラスタースクロール用の *sin* カーブテーブルを取り込みます。

▶11 行〜31 行

```
static int 面初期化;
```

初期化の段階を制御する変数です。

```
static unsigned char *vram0;
static unsigned char *vram1;
```

グラフィック画面の VRAM アドレスを保持するためのポインタです。

```
static int now_counter;
static unsigned char *map_data_ptr0;
static unsigned char *map_data_ptr1;
```

マップデータを管理するためのカウンタと，転送マップを示すポインタです。

RASTER_MIN

画面表示開始ラスター位置です。

RASTER_MAX

画面表示終了ラスター位置です。

RASTER_STEP

ラスタースクロールの間隔です。

```
static int *raster_map;
```

マップ上でのラスター範囲を管理するためのポインタです。

```
static int scroll_counter;
```

スクロール量を計測するカウンタです。

```
static short do_raster;
```

ラスタースクロールを実行するためのフラグです。'1' でラスタースクロールを実行します。

```
static short start_raster;
```

ラスタースクロール開始ラスター位置を示す変数です。

```
static short end_raster;
```

ラスタースクロール終了ラスター位置を示す変数です。

```
static int index_val;
```

sin カーブでラスタースクロールさせるためのカウンタです。

```
static CRTC_REG raster_val;
```

スクロールレジスタに書き込むための変数です。

▶36 行〜61 行

```
static void write_vram (unsigned short *vr, unsigned short *dat);
```

CPU が VRAM に直接書き込む場合の，1 タイル (16 × 16 ドット) を VRAM に転送する関数です。

▶63 行〜107 行

```
static void write_vram (unsigned char *vr, unsigned char *dat);
```

DMA転送でグラフィックを書き込む場合のVRAM転送関数です。**GCC**の**asm**文を使ってIOCSを発行しています。

▸109行～137行

```
void write_one_block0 ();
void write_one_block1 ();
```

それぞれの独立したグラフィック画面に，16×16ドットのタイルを横方向に18個転送するグラフィック描画関数です。

▸139行～231行

```
void ロードマップ (int 面);
```

引数のint 面に応じたマップのグラフィックデータをロードします。マップデータを解析して，ラスタースクロール開始時間を決定しておきます。

▸234行～275行

```
void raster_scroll ();
```

これがラスタースクロールを行う本体の関数です。割り込み処理関数になっています。**X68000/X68030**では，指定ラスター位置をCRTCが走査したときに割り込みを発生させることができますが，次の割り込み発生は垂直帰線期間が経過してからになります。ラスタースクロールを行う場合，次々に割り込みが発生しないとラスタースクロールできませんから，割り込むつど，次の割り込み発生ラスター位置を設定して，ラスタースクロールを行う画面範囲に間断なく割り込みが起こるようにしておきます。

また，ラスタースクロールでは，割り込み処理の中で複雑な処理を行うと破綻します。割り込み処理を行っている間も，CRTCは次々とラスターを更新していますから，のんびりと処理していたのでは間に合わないのです。そこで，割り込み処理ではできるだけ複雑な処理を行わないように，事前に用意できるデータは用意しておいて，最低限の処理で割り込み処理から復帰するようにしておきます。

▸277行～380行

```
void 背景移動処理 (void);
```

グラフィックのスクロール制御を行う本体部分です。変数面初期化の値によって処理が変化します。

面初期化が'1'のときは，その面がスタートです。いったん，グラフィックパレットを全部暗転させて，見掛け上グラフィックが表示されていない状態にします。その後，グラフィックパレットを徐々に規定の色まで暗転状態から明るい方向へ設定します。このゲームでは行っていませんが，画面の真ん中に「READY!!!」を表示する場面です。個人的な考え方ですが，IOCS のコントラストを使って画面を徐々に見せる方法は，**X68000/X68030** ではあまりに安直すぎておもしろくありません。そこで，やや凝った方法でスタートを演出してみました。

ラスタースクロールのオン，オフもこの関数で行っています。あらかじめマップデータからラスタースクロールを行う範囲を計算してありますので，その結果を参照してラスタースクロールを行っています。

背景のグラフィックでは，アイデア次第でいろいろと凝った演出ができます。ラスタースクロールも今ではそれほど珍しくない手法ですが，ナイアス (EXACT) で最初にデモを見たときには，けっこう衝撃的な演出だと感じました。その感動はいまだ忘れられません。半透明機能や特殊プライオリティは，グラフィックデータをうまく作成すれば，効果的な演出をすることもできるでしょう。

4.9.2 ■ バックグランド画面の処理

このゲームでは，バックグランド画面はスコアの表示と，自機の「溜め撃ち」のエネルギー蓄積量 (?) の表示に使っています。バックグランド画面というのはスプライト表示画面の 1 つであり，PCG 番号を指定して，その番号のキャラクタで画面を埋めることができます。4.9.1「グラフィック画面の処理」(P.223) では，グラフィックを 16 × 16 ドットのタイルで埋めるようにグラフィック画面を扱っていましたが，このタイルがちょうど PCG 番号で指定するキャラクタに相当するイメージになります。バックグランド画面は，このゲームのように 256 × 256 ドットモードで使う場合には，8 × 8 ドットの PCG で画面を埋める仕様になっています。

このとき，PCG 番号は，スプライトの PCG 番号とは一致しないハードウェア仕様になっています。PCG の仕様については，3.5.1「プログラマブルキャラクタジェネレータ」の Fig. 3.9「キャラクタのドットとハードウェアの対応」(P.119) で説明していますが，8 × 8 ドットのタイルは Fig. 3.9 の 0，1，2，3 の順序で数字が増えていきます。スプライトでの PCG 番号を n とすると，バックグランド画面での PCG 番号は $2n + k$(k は 0〜3) になります。

List 4.9 back.c

```
1: #include "game.h"
2:
3: static BG_REG 数字フォント[10];
4: static BG_REG ゲージfont[13];
5:
6:
```

```
 7: void
 8: BGデータ初期化 (void)
 9: {
10:   {
11:     int pcg_no;
12:     int i;
13:
14:     pcg_no = SP_PCG_NO (SPD_BODY (数字, 0, 0), 0);
15:     for (i = 0; i < 4; i++)
16:       {
17:         数字フォント[i + 1].sp_code = pcg_no * 4 + i;
18:         数字フォント[i + 1].color = 3;
19:       }
20:
21:     pcg_no = SP_PCG_NO (SPD_BODY (数字, 0, 0), 1);
22:     for (i = 4; i < 8; i++)
23:       {
24:         数字フォント[i + 1].sp_code = pcg_no * 4 + (i - 4);
25:         数字フォント[i + 1].color = 3;
26:       }
27:
28:     pcg_no = SP_PCG_NO (SPD_BODY (数字, 0, 0), 2);
29:     数字フォント[9].sp_code = pcg_no * 4;
30:     数字フォント[9].color = 3;
31:     数字フォント[0].sp_code = pcg_no * 4 + 1;
32:     数字フォント[0].color = 3;
33:   }
34:   {
35:     int pcg_no;
36:     int i;
37:     pcg_no = SP_PCG_NO (SPD_BODY (ゲージ, 0, 0), 0);
38:     for (i = 0; i < 4; i++)
39:       {
40:         ゲージfont[i].sp_code = pcg_no * 4 + i;
41:         ゲージfont[i].color = 1;
42:       }
43:     pcg_no = SP_PCG_NO (SPD_BODY (ゲージ, 0, 0), 1);
44:     for (i = 4; i < 8; i++)
45:       {
46:         ゲージfont[i].sp_code = pcg_no * 4 + (i - 4);
47:         ゲージfont[i].color = 1;
48:       }
49:     pcg_no = SP_PCG_NO (SPD_BODY (ゲージ, 0, 0), 2);
50:     for (i = 8; i < 12; i++)
51:       {
52:         ゲージfont[i].sp_code = pcg_no * 4 + (i - 8);
```

```
53:         ゲージfont[i].color = 1;
54:       }
55:     ゲージfont[12].sp_code = ゲージfont[0].sp_code;
56:     ゲージfont[12].color = ゲージfont[0].color;
57:     ゲージfont[12].h_invert= 1;
58:   }
59: }
60:
61: void
62: スコア表示 (void)
63: {
64:     BG_REG *dest = &bg_array[4];
65:     int sc = スコア * 10;
66:     int i,val;
67:     for (i = 10000000; i > 0; i/=10)
68:       {
69:         val = sc / i;
70:         *dest++ = 数字フォント[val];
71:         sc -= (val * i);
72:       }
73: }
74:
75: static char gaji_val[][6] =
76: {
77:   { 0, 4, 4, 4, 4, 12 },
78:   { 1, 4, 4, 4, 4, 12 },
79:   { 2, 4, 4, 4, 4, 12 },
80:   { 3, 4, 4, 4, 4, 12 },
81:   { 3, 5, 4, 4, 4, 12 },
82:   { 3, 6, 4, 4, 4, 12 },
83:   { 3, 7, 4, 4, 4, 12 },
84:   { 3, 8, 4, 4, 4, 12 },
85:   { 3, 8, 5, 4, 4, 12 },
86:   { 3, 8, 6, 4, 4, 12 },
87:   { 3, 8, 7, 4, 4, 12 },
88:   { 3, 8, 8, 4, 4, 12 },
89:   { 3, 8, 8, 5, 4, 12 },
90:   { 3, 8, 8, 6, 4, 12 },
91:   { 3, 8, 8, 7, 4, 12 },
92:   { 3, 8, 8, 8, 4, 12 },
93:   { 3, 8, 8, 8, 5, 12 },
94:   { 3, 8, 8, 8, 6, 12 },
95:   { 3, 8, 8, 8, 7, 12 },
96:   { 3, 8, 8, 8, 8, 12 },
97:   { 3, 8, 8, 8, 8, 9 },
98:   { 3, 8, 8, 8, 8, 10 },
```

```
 99:   { 3, 8, 8, 8, 8, 11 },
100: };
101:
102:
103: void
104: ゲージ表示 (void)
105: {
106:     BG_REG *dest = &bg_array[20];
107:     int val = 蓄積/(150/(sizeof (gaji_val) / 6));
108:     int i;
109:     if (蓄積 && val == 0)
110:       val = 1;
111:     else if (val > sizeof (gaji_val) / 6 -1)
112:       val = sizeof (gaji_val) / 6 -1;
113:     for (i = 0; i < 6; i++)
114:       *dest++ = ゲージfont[gaji_val[val][i]];
115: }
```

バックグランドの表示を行うためには，PCG にデータを登録することと，スプライトスクロールレジスタを割り当てないSprite型のデータが必要です。これがダミーとして用意された数字という変数と，ゲージという変数です。これらはゲーム初期化のとき，ダミーとして初期化されます (List 4.4 P.193を参照してください)。

このゲームのスプライト管理方法では，各スプライトやバックグランドに割り当てられる PCG 番号は「重ね合わせの優先順位」をまったく意識していませんので，ハードウェアに登録されるまでその番号は不定です。そこで，ダミーの初期化が終わった後で，マクロSP_PCG_NO()によってハードに登録された PCG 番号を得ています。こうして得られた PCG 番号から，実際にバックグランド表示に使うBG_REG型の配列数字フォント []とゲージ font[] を作成しています。これらの変数は，その名前が示すように，数字やエネルギーの蓄積を表示するためのパターンを，ちょうど漢字フォントのようなイメージで並べたものです。たとえば，数字フォント [3]には，バックグランドに表示させると '3' を表示するデータが入っているように初期化されています。関数スコア表示 ()や関数ゲージ表示 ()では，ゲームのスコアや，ジョイスティックのトリガの押し下げ時間を表す変数を参照して，割り込み処理との連絡に使うBG_REG型の配列bg_array[] に適切なデータをセットします。

では，実際のソースリスト List 4.9 の説明です。

▶3行〜4行

```
static BG_REG 数字フォント[10];
static BG_REG ゲージfont[13];
```

これらはバックグランド画面に転送するためのデータを作成するバッファです。

▶7行～59行

```
void BGデータ初期化 (void);
```

ダミーのスプライト数字とゲージからPCG番号を取得して，数字フォントやゲージfontを作成しています。

▶61行～73行

```
void スコア表示 (void);
```

ゲームの得点から，バックグランド画面に表示する数字フォントを作成しています。わりあい簡単なアルゴリズムなので，読むのは簡単でしょう。

▶75行～115行

```
void ゲージ表示 (void);
```

「エネルギー蓄積量」からバックグランド用のゲージデータを作成しています。gaji_val[][]は，パターン検索用の配列です。

■4.10 メインルーチン

最後に，最も上位にあるmain()を含んだメインルーチンを示します(List 4.10)。

List 4.10 game.c

```
 1: /* 簡単なゲーム for C Magazine */
 2:
 3: #define MAIN
 4: #include "game.h"
 5:
 6: /* ゲームのメインルーチン */
 7: static void
 8: ゲームメイン (void)
 9: {
10:   int sc = 0;
11:   int lkup = 5;
12:   int 面 = 0;
13:
14:   while (!終 && 面 <= 最終面)
15:     {
```

```
16:       ロードマップ (面);
17:       do {
18:           背景移動処理 ();
19:           スコア表示 ();
20:           ゲージ表示 ();
21:           sp_is_ready = 1;
22:           while (sp_is_ready)
23:             ;
24:           消去実行 ();
25:       } while (!ゲーム開始);
26:
27:      自機移動処理 ();
28:      弾発射移動 ();
29:      sp_is_ready = 1;
30:      while (sp_is_ready)
31:       ;
32:
33:       do
34:         {
35:           /* 一時停止処理 */
36:           if (BITSNS (0) & 2)
37:             {
38:               while ((BITSNS (0) & 2))
39:                 ;
40:               while (!(BITSNS (0) & 2))
41:                 ;
42:               while ((BITSNS (0) & 2))
43:                 ;
44:             }
45:           自機移動処理 ();
46:           弾発射移動 ();
47:           敵キャラ発生移動 ();
48:           当たり判定 ();
49:           背景移動処理 ();
50:           スコア表示 ();
51:           ゲージ表示 ();
52:           sp_is_ready = 1;
53:           while (sp_is_ready)
54:             ;
55:           消去実行 ();
56:             if (((経過時間++) & 7) == 7)
57:             スコア += 1;
58:           if ((スコア >> 2) - sc > lkup)
59:             {
60:               sc = (スコア >> 2);
61:               ランク += ランク < MAX_NUM ? (ランク > 90 ? lkup += 5, 1 :  5) :  0;
```

```
62:             }
63:         }
64:       while (!終 && !面クリア);
65:       if (面クリア)
66:         {
67:           面クリア = 0;
68:           面++;
69:         }
70:     }
71: }
72:
73: void
74: main ()
75: {
76: #ifdef SOFT_TRANS
77:   volatile int ssp = 0;
78: #endif
79:   init_trap14 ();
80:
81: #ifdef SOFT_TRANS
82:   ssp = B_SUPER (0);
83: #endif
84:   if (setjmp (err_buf))
85:     {
86: #ifdef SOFT_TRANS
87:       if (ssp)
88:         B_SUPER (ssp);
89: #endif
90:       exit (1);
91:     }
92:
93:   画面初期化 ();
94:   ロードデータ ();
95:   BGデータ初期化 ();
96:   sp_is_ready = 1;
97:   while (sp_is_ready)
98:     ;
99:   ゲームメイン ();
100:  画面終了処理 ();
101: #ifdef SOFT_TRANS
102:  B_SUPER (ssp);
103: #endif
104:  printf ("経過時間  %d スコア %d ランク %d\n",
105:          経過時間, スコア, ランク);
106:  exit (0);
107: }
```

▶6行〜71行

```
static void ゲームメイン (void);
```

ゲームのメインループが定義されている関数です。形式的には面クリアの形になっていますので，複数面構成にできるはずです。割り込み処理との連絡はsp_is_readyを使って行います。自機移動，弾発射移動，敵キャラ発生移動，当たり判定，背景移動と順次処理して，sp_is_readyをセットします。割り込み処理が終わったら，スプライトを消去する関数消去実行 ()を呼び出します。

▶73行〜107行

main()です。画面の初期化やデータのロードを行います。**X68000/X68030** の性質により，割り込みを登録してから実際に割り込みが入るまで若干時間がかかるので，割り込みが実行されるまで関数ゲームメイン ()の呼び出しを保留しています。グラフィック VRAM への書き込みを DMA で行わない場合には，あらかじめスーパーバイザモードに移行しておきます。また，下位の関数でのエラートラップとして，エラーが発生した場合には，setjmp()で強制的にmain()に復帰するように設定しておきます。

APPENDIX

付録

1. X68000/X68030での致命的エラーハンドリング
2. 演算子の一覧表
3. GNU一般公有使用許諾書

■1. X68000/X68030での致命的エラーハンドリング

X68000/X68030 では，CPU が保護機能の強力な 68000/680EC30 ということもあって，プログラミングミスによる CPU の暴走が原因で，HDD が破壊されるといった類の事故は比較的起こりにくくなっています。ですが，プログラムにはバグはつきものです。ゲームを作成する場合には，このような不慮の場合も考えておかなければなりません。**X68000/X68030** で実際に起こり得るエラーには，本当に致命的なものとそうでないものがあります。本当に致命的なものの代表には，

1. **バスエラー** (バスエラーが発生しました)
2. **アドレスエラー** (アドレスエラーが発生しました)
3. **不当命令実行** (おかしな命令を実行しました)

があげられるでしょう。このような事態に陥った場合には，**Human68k** では画面中央に突然白い窓が開いて，上記 () 内のメッセージが表示されます。List A.1，A.2，A.3 が故意にこれらのエラーを起こさせるソースです。

List A.1

```
1: /* バスエラーを起こす */
2: main()
3: {
4:   char *p = 0;
5:   char c;
6:   c = *p;
7:   return c;
8: }
```

List A.2

```
1: /* アドレスエラーを起こす */
2:
3: char buf[10];
4:
5: main()
6: {
7:    int *x = (int *)&buf[1];
8:    int i = *x;
9:    return i;
10: }
```

List A.3

```
1: /* おかしな命令を実行する */
2:
3: char *p = "ABCDE";
4:
5: main()
6: {
7:   void (*fun)(void) = (void (*)(void)) p;
8:   fun ();
9: }
```

List A.1 と List A.3 は，保護機能がない機械では何も起きないか，暴走してリセットするしかないような事態を引き起こす，非常に危険きわまりないプログラムですが，**X68000/X68030** では画面に白い窓が開いて実行が停止させられ，'A' キーを押せば何もなかったかのように COMMAND.X に復帰できます。たいていの場合，RAM DISK は無事ですし，エラーが原因でその後の動作が変になることもありません。

これらのエラーは，発生した時点で致命的なものなので，素直に **Human68k** に復帰してしまうのが普通のやり方ですが，もしこれがエディタのような今までの作業をすべて水泡に帰してしまうアプリケーションの場合には，ちょっと問題があります。プログラムがバグっているために停止したとしても，今までの作業をすべて白紙に戻さずに作業バッファをファイルに書き出し，少しでも被害を食い止めるようにしてあれば，ユーザの被害を最小限に食い止めることができるかもしれません。

先ほど述べた致命的なエラーの他に，致命的ではないエラーがあります。たとえば List A.4 です。

List A.4

```
 1: #include <stdio.h>
 2:
 3: void
 4: main(int argc, char **argv)
 5: {
 6:   if (argc == 2)
 7:     {
 8:       FILE *fp = fopen (argv[1], "w");
 9:       if (!fp)
10:         {
11:           fprintf (stderr, "File Open Err\n");
12:           exit (0);
13:         }
14:       fclose (fp);
15:     }
```

```
16: }
```

List A.4 は，ただ単にコマンドラインから与えられたファイルを作るだけのプログラムですが，フロッピーが挿入されていないドライブに対してファイルを作成するように動かした場合には，これまた白い窓が開いて「ディスクが入っていません」といったメッセージが表示されるでしょう。

同様に，フォーマットされていないディスクに対してこのようなファイルオープンを行うと，またまた白い窓が開いてしまいます。この種類のエラーは，画面関係を変更して動いているアプリケーションにおいては致命的です。たとえそうでないにしても，この種類のエラーは自前でハンドリングを行ったほうがスマートです。このような種々のエラーに対応するには，どのようにしたらよいのでしょうか??

答えは **XC** に付属している「プログラマーズマニュアル」に記載されています。**Human 68k** はプロセスを生成する場合に，「プロセス管理ポインタ」と呼ばれるメモリブロックを作成してそのプロセスを管理します。このプロセス管理ポインタは，各プロセスごとにエラーハンドリングを行う処理へのポインタをもっていて，エラーが発生した場合には **Human68k** はそこを参照して処理を行います。List A.5 が，このプロセス管理ポインタを C 言語の構造体で表したものです。

List A.5

```
 1: struct プロセス管理ポインタ
 2: {
 3:     unsigned char *env;          /* プロセスの環境へのポインタ */
 4:     void          *end;          /* プロセス終了時の戻りアドレス */
 5:     void          *ctrl_c;       /* Ctrl-Cで中止された場合の戻りアドレス */
 6:     void          *abort;        /* プロセスがエラー中断した場合の戻りアドレス */
 7:     unsigned char *cmdline;      /* プロセスに与えられたコマンドライン */
 8:     unsigned long file[4];       /* ファイルハンドラの使用状況 */
 9:     void          *bss_ptr;      /* プロセスのbss領域へのポインタ */
10:     void          *heap_ptr;     /* プロセスのヒープ先頭 */
11:     void          *ini_stack;    /* プロセスの初期スタック */
12:     void          *usp;          /* 親プロセスのUSP */
13:     void          *ssp;          /* 親プロセスのSSP */
14:     short         *sr;           /* 親プロセスのSR */
15:     short         *abort_sr;     /* 中止のときはSR */
16:     void          *abort_ssp;    /* 中止のときはSSP */
17:     void          *power_off;    /* パワーオフ，リセットの処理ルーチン(trap 10)*/
18:     void          *break_key;    /* HDDリトラクトetc.(trap 11)*/
19:     void          *copy_key;     /* COPY キーの処理(trap 12)*/
20:     void          *ctrl_c_fnc;   /* ブレークチェックフラグ処理(trap 13)*/
21:     void          *err_handle;   /* エラーハンドラ(trap 14)*/
22:     long          process_flag; /* プロセスのフラグ(0で親あり) */
23:     long          un_used0[7];
```

```
24:     char            drv[2];         /* 実行されたファイルのあるドライブ名 */
25:     char            path[66];       /* 実行されたファイルのあるパス */
26:     char            name[24];       /* 実行されたファイル名 */
27:     long            un_used[9];
28: } プロセス管理ポインタ;
```

List A.5で明らかなように，まさにこれはポインタの固まりです。たくさんのメンバが定義されていますが，ここで注目していただきたいのは`void *err_handle`です。これはプロセスがエラーを起こした場合に参照されるベクタです。**Human68k**では，バスエラーやアドレスエラーが発生すると，その例外処理で`trap 14`を発行します。この`trap 14`の処理アドレスは，プロセス管理ポインタ構造体の`void *err_handle`になります。`trap 14`が発行されるときには，d7レジスタにそのエラーの原因がセットされています。もしシステムデフォルト(例の白い窓)でのエラー処理を行わない場合には，この`trap 14`処理を自前で用意すればいいのです。

C言語に詳しく，**UNIX**上でのプログラミング経験がある方なら「なんだ，要するに`signal()`のことか」と思うでしょう。そのとおりなのですが，**XC**ライブラリの`signal()`は貧弱で，Ctrl–Cまたは BREAK キーによる中断しか扱うことができません。**UNIX**では，`SIGBUS`や`SIGILL`で扱う種類のハンドラは，今のところ自前で処理してやるしか方法がないのです。

1.1 ■ エラーハンドラ(List A.6)

さて自前でエラーを処理するわけですが，先ほども書いたように，エラー種別のパラメータはレジスタd7経由で渡されるので，そのままではCで扱うことはできません。通常はアセンブラでハンドラを記述するのでしょうが，**X68000/X68030**に特化した**GCC**ではこれを容易に扱うことができます。List A.6がそのハンドラの例です。エラーハンドラは一種の割り込み処理なので，interrupt.hを**#include**して割り込み処理として記述します。

List A.6 trap14.c

```
 1: /* system include */
 2: #include <stdio.h>
 3: #include <interrupt.h>
 4:
 5: #ifdef DEV_GCC
 6: /* Develop 版 GCC 拡張 */
 7: DOSCALL INTVCG (short);
 8: DOSCALL INTVCS (short, void *);
 9: DOSCALL PRINT (char *);
10: #else
11: #include <doslib.h>
12: #endif
13:
```

```
14: /*
15: stdin,stdout が XC Ver 依存のため
16: sprintf でマクロにしておく
17: */
18: #define Fprint(buf,fmt,mes)    \
19: do {                           \
20:   sprintf ((buf),(fmt),(mes)); \
21:   PRINT ((buf));               \
22: } while (0)
23:
24:
25: static struct {
26:  int flag;                  /* エラー種別フラグ */
27:  unsigned short sr_reg;     /* エラー時の SR */
28:  int pc_reg;                /* エラー時の PC */
29:  char *mes;                 /* メッセージ */
30: } trapbuf;
31:
32: static void (*trap_14)();   /* 元の trap 14 ベクタを保存 */
33:
34: void (*usr_abort)(void);    /* ユーザのアボート処理関数 */
35:
36: extern int _main;           /* スタートアップのダミー */
37:
38: static void
39: trap14(void)
40: {
41:   PRAMREG (code, d7);       /* パラメータレジスタ d7 */
42:   PRAMREG (add,  a6);       /* パラメータレジスタ a6 */
43:   SET_FRAME (a5);           /* フレームポインタを a5 に指定 */
44:   char *p =(char *)add;
45:
46:   /* code からエラーを分析 */
47:   switch (code&0xffff)
48:     {
49:     case 2:
50:       /* バスエラー */
51:       trapbuf.flag = code;
52:       trapbuf.mes = "バスエラーが発生しました";
53:       break;
54:     case 3:
55:       /* アドレスエラー */
56:       trapbuf.flag = code;
57:       trapbuf.mes = "アドレスエラーが発生しました";
58:       break;
59:     case 4:
```

```
60:       /* おかしな命令 */
61:       trapbuf.flag = code;
62:       trapbuf.mes = "おかしな命令を実行しました";
63:       break;
64:     }
65:  if ((code &0xff00))
66:    {
67:      if ((code &0xff) == 2)
68:        {
69:          trapbuf.flag = -1;
70:          trapbuf.mes  = "ドライブの準備ができていません";
71:        }
72:     }
73:  if (trapbuf.flag || (code&0xffff) == 0x1f || (code&0xffff) == 0x301f)
74:     {
75:       trapbuf.sr_reg = *(short *)p;
76:       p+=2;
77:       trapbuf.pc_reg = *(int *)p;
78:       asm ("\tmove.w #$ff,d0\n\ttrap #15\n");
79:       IJUMP_RTE();
80:     }
81:   else
82:     IJUMP(trap_14);
83: }
84:
85: static void
86: my_abort(void)
87: {
88:   char buf[64];
89:   INTVCS(0x2e,trap_14);
90:   if (usr_abort)
91:     usr_abort ();
92:   if (trapbuf.flag > 0)
93:     {
94:       extern int _SSTA;
95:       extern int _PSTA;
96:       trapbuf.flag = 0;
97:       Fprint (buf,"%s\r\n",trapbuf.mes);
98:       Fprint (buf," pc=%X",trapbuf.pc_reg);
99:       Fprint (buf," offset=%X\r\n",trapbuf.pc_reg - (int) &_main);
100:      exit(512);
101:    }
102:  else
103:    {
104:      Fprint (buf,"%s\r\n",trapbuf.mes);
105:      exit (256);
```

```
106:     }
107: }
108:
109: void
110: init_trap14 (void)
111: {
112:   trap_14 = (void *)INTVCG (0x2e);
113:   INTVCS (0x2e,trap14);
114:   INTVCS (0xfff2,my_abort);
115:   INTVCS (0xfff1,my_abort);
116: }
```

それでは，List A.6 について詳細に説明します。

▸14 行〜22 行

XC Ver.1 と **XC Ver.2** では，FILE構造体にまったく互換性がありません。そこで，直接 stdio を使わないで，sprintfでメッセージを作成して，それを DOS print で出力する方法により，バージョン依存性を排除しておきました。このためのマクロがこの 14 行〜22 行です。

▸25 行〜30 行

エラーハンドラは割り込み処理に相当しますので，この処理内部ではいっさいの入出力は行わないで，このstruct trapbufを経由して情報を伝達します。メンバの詳細については，エラーハンドラ本体で再度詳しく説明します。

▸32 行

このエラーハンドラでは，すべてのエラーをハンドリングしませんので，自前で処理する以外のエラーは，もともと **Human68k** がもっている例の「白い窓」の処理ルーチンに処理させます。このtrap_14はその処理アドレスへのポインタです。

▸34 行

ゲーム等の割り込み処理をもったプログラムの場合，バスエラー等で停止して **Human68k** に復帰させるには，フックした割り込み処理を開放したり，画面モードを変更したりする必要があります。そのような処理を行わずに，ただ単に **Human68k** に復帰させたときには，割り込み処理が暴走することは，本書で何度か説明しました。そこで，エラーで停止したときには，**Human68k** に復帰する前に後始末をする関数をエラーハンドラに登録で

きるようにしておきます。それが，このvoid (*usr_abort)(void)です。

▶36行

エラーが起こった場合，その後のデバッグを考えると，エラーが起こったアドレスを後で確認できると便利です。そこで，**XC**のスタートアップである_mainからエラーが起こった場所のオフセットを表示させます。こうしておけば，db.xでプログラムをロードした直後に

-l .pc + 表示されたオフセット

で，即座にエラーが発生したアドレスを逆アセンブルして，デバッグの参考にすることができます。

▶38行～107行

エラーハンドラの本体です。**X68000/X68030版GCC**の拡張をふんだんに用いた，非常に特殊な関数になっています。

41行，42行

PRAMREGマクロは，非常に特殊なレジスタ宣言を行うマクロです。

PRAMREG(*var*,*regname*)により，変数*var*に*regname*の物理レジスタを割り当てます。この*regname*の物理レジスタは，本来保存すべきレジスタであっても，関数の入り口や出口で保存されることはありません。ですから，このエラーハンドラのように，レジスタを経由してパラメータが渡される処理を，C言語で記述できます。

レジスタd7はエラー種別ですから，パラメータレジスタとして扱うのは当然ですが，さらにa6をパラメータレジスタとして宣言しています。「プログラマーズマニュアル」には明確な記述はありませんが，このa6が示すアドレスにエラーが起こったときのステータスレジスタとプログラムカウンタが格納されています。そこで，その情報を取得するためにa6もパラメータレジスタとして宣言します。ただし，このレジスタは破壊してはならないので，参照を行うだけで変更はしません。

43行

XCや**GCC**の生成するコードに詳しい方なら知っておられると思いますが，a6レジスタは，通常，フレームポインタレジスタとして用いられるレジスタです。ところが，このエラーハンドラではa6を変更することはできませんので，フレームポインタレジスタとしてa5を使うようにコンパイラに指示します。そのマクロがこのSET_FRAME()マクロです。**X68000/X68030版GCC**で，SET_FRAME()マクロで指定できるアドレスレジスタはa3，a4，a5の3つです(a6はデフォルトですね。指定はできますけど)。

44 行

PRAMREGされたa6は値を変更できませんので，関数の入り口で別のポインタ変数にコピーしておきます。**GCC** のフロー解析では，ポインタ変数pの中身がa6に割り当てたaddと等価であるという理由で，変数pの代わりにadd，つまりa6を直接使ったりはしませんので，このような書き方ができます。

46 行〜64 行

エラーコードを保持しているcodeの内容を解析して，struct trapbufにエラーコードとメッセージをセットします。

65 行〜72 行

ディスクの準備ができていない場合のエラーの処理です。エラーコードはd7の下位 16 ビットですが，ディスクノットレディの場合には，上位 8 ビットに 0 ではない不定の値が入っているようです。

73 行〜83 行

この **if** 文は，このエラーハンドラが処理すべきエラーか否かによって，関数の終了時の処理を切り替えています。このエラーハンドラが処理するエラーの場合には，エラーが起こったプログラムカウンタとステータスレジスタをtrapbufに格納した後，アボート処理 (d0 = -1のtrap #15) を発行してrteします。IJUMP_RTE()は，関数の処理が終わった後，rteが格納されている番地にjmpしてrteを実行します。このエラーハンドラが扱わないタイプのエラーの場合は，元来の処理に制御を移す必要がありますので，マクロIJUMP()で変数trap_14に格納されている **Human68k** 本来のエラー処理ルーチンにjmpします。

IJUMP()は，関数の終わりの処理がrte命令であったり，jmp命令であったりする，非常に特殊な関数になります。こういった類の処理は，普通の C 言語ではまったく処理することができない (たとえ強力な **GCC** の **asm** 文を用いても) のですが，**X68000/X68030 版 GCC** では特に意識しなくても適切なコードをコンパイラが生成します。

85 行〜107 行

これが，実際にプログラムが中断される場合に最終的に呼び出される関数です。この関数は，returnしないexit()またはabort()を呼び出す必要があります。どんな場合でも，呼び出し元に復帰してはいけません。まず，自前で登録したエラー処理を元に戻します。この後に発生するエラーは **Human68k** が処理するようになりますので，バスエラー等が発生した場合には，再び「白い窓」が開くようになります。

その後，ユーザ定義の後始末をする関数 (usr_abort変数が指している関数) があれば，それを呼び出して後始末をした後，trapbufに格納されている情報を出力してexit()します。

▶109 行〜116 行

エラーハンドラを **Human68k** に登録する初期化ルーチンです。関数`main()`のいちばん初めに呼び出しておけば，その後に起こるバスエラー等は，すべて自前の処理ルーチンで処理されるようになります。

1.2 ■ ライブラリとしての使い方

それでは，実際に List A.1 のバスエラーをトラップしてみましょう。まず List A.7 のように，`init_trap14()`を呼び出すように改変しておきます (bustrap.c)。

List A.7 bustrap.c

```
1: /* バスエラーをトラップする */
2: main()
3: {
4:    char *p = 0;
5:    char c;
6:    init_trap14 ();
7:    c = *p;
8: }
```

この bustrap.c を

gcc bustrap.c -O -ltrap14

のようにコンパイルして，trap14lib.a(.l) をリンクします。このプログラムを実行すると，「白い窓」は表示されないで,「バスエラーが発生しました」のメッセージとともにプログラムのデバッグ情報が出力されるでしょう。デバッグを行う場合には，この情報は有効です。次の画面のように即座にデバッガに入って，表示されたオフセットを起動直後のプログラムカウンタに加算し，逆アセンブルすることで，プログラムのどの位置でバスエラーが発生したかを特定することができます。

◆画面 A.1

```
A>bustrap
バスエラーが発生しました
 pc=8264A offset=FFFFFFCC
A>db bustrap.x
X68k Debugger v1.01 copyright 1987 SHARP/Hudson
loading bustrap.x
PC=00089C5E USP=00088CF4 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  00089B20 0008BD6E 00089604 0007D110  00089C5E 00000000 00000000 00088CF4
```

```
__main:
lea         $0008BD6E,A7                ;0008BD6E(60)
```

-l .pc+ffffffcc

```
00089C2A        move.b        (A3),D0
00089C2C        ext.w         D0
00089C2E        ext.l         D0
00089C30        movea.l         (A7)+,A3
00089C32        rts
00089C34         undefined instruction $436F
00089C36        moveq         #$79,D0
00089C38        moveq         #$69,D1
```

また，usr_abort()に割り込みを元に戻す処理を登録しておけば，予期しないバスエラー等でプログラムが中断しても，リセットを行うことなくデバッグに入ることができます。このライブラリをリンクせずに割り込みを使っていると，バグのつどリセットするはめになりますから･･･。

■2. 演算子の一覧表

●Table A.1 算術演算子

演算子	機 能
()	関数コール
[]	配列添字
.	構造体のメンバ参照 (構造体の名前による)
->	構造体のメンバ参照 (構造体を指すポインタによる)
++	インクリメント演算子 (変数などを 1 つずつ増やす)
--	デクリメント演算子 (変数などを 1 つずつ減らす)
!	否定演算子
~	ビットごとの否定演算子
-	単項演算子の '-'
&	変数などのアドレス
*	ポインタからのデータ参照
sizeof	変数の型のサイズを得る
(*type*)	型キャスト
*	乗算
/	除算
%	剰余
+	加算
-	減算
>>	右シフト
<<	左シフト
<	より小さい
<=	以下
>	より大きい
>=	以上
==	等しい
!=	等しくない
&	ビットごとの AND
^	ビットごとの排他的 OR
|	ビットごとの OR
&&	論理積
||	論理和
$z?x:y$	z が真ならば x を，偽ならば y を評価する
=	代入
+=	加算してから代入
-=	減算してから代入
*=	乗算してから代入
/=	除算してから代入
%=	剰余を代入
&=	ビットごとの AND をしてから代入
^=	ビットごとの排他的 OR をしてから代入
|=	ビットごとの OR をしてから代入
<<=	左シフトしてから代入
>>=	右シフトしてから代入
,	カンマ演算子

■3. GNU一般公有使用許諾書

本書および『付録ディスク』に含まれるすべてのプログラムは，以下の「GNU一般公有使用許諾書」を適用しています。公的・私的にかかわらず，これらのプログラムを使用する場合は，必ず「GNU一般公有使用許諾書」全文をお読みください。

「GNU一般公有使用許諾書」全文

注意

英文文書 (GNU General Public License) を正式文書とする。この和文文書は弁護士の意見を採り入れて，できるだけ正確に英文文書を翻訳したものであるが，法律的に有効な契約書ではない。

和文文書自体の再配布に関して

いかなる媒体でも次の条件がすべて満たされている場合に限り，本和文文書をそのまま複写し配布することを許可する。また，あなたは第三者に対して本許可告知と同一の許可を与える場合に限り，再配布することが許可されています。

- 受領，配布されたコピーに著作権表示および本許諾告知が前もって載せられていること。
- コピーの受領者がさらに再配布する場合，その配布者が本告知と同じ許可を与えていること。
- 和文文書の本文を改変しないこと。

GNU一般公有使用許諾書

1989年2月，バージョン1

675 Mass Ave, Cambridge, MA 02139, USA

何人も，以下の内容を変更しないでそのまま複写する場合に限り，
本使用許諾書を複製したり頒布することができます。

はじめに

ソフトウェア会社の使用許諾契約書は，多くの場合，その企業の意のままにユーザを縛ろうとしています。それに対して，我々の一般公有使用許諾は，フリー・ソフトウェアを共有したり変更する自由をユーザに保証するためのもの，即ちフリー・ソフトウェアがそのユーザ全てにとってフリーであることを保証するためのものです。本使用許諾は，Free Software Foundationのソフトウェアに適用されるだけでなく，プログラムの作成者が本

使用許諾に依るとした場合のそのプログラムにも適用することができます。また，ユーザのプログラムのためにも利用することができます。

我々がフリー・ソフトウェアについて言う場合は自由のことに言及しているのであって，価格のことではありません。特に，一般公有使用許諾の各条項は，次の事柄を確実に実現することを目的として立案されています。

- フリー・ソフトウェアの複製物を自由に頒布したり販売できること。
- 希望しさえすればソース・コードを現実に入手できるか，あるいはその入手が可能であること。
- 入手したソフトウェアを変更したり，新しいフリー・プログラムの一部として使用できること。
- 以上の各内容を行なうことができるということをユーザ自身が知っていること。

このようなユーザの権利を守るために，我々は，何人もこれらの権利を否定したり，あるいは放棄するようにユーザに求めることはできないという制限条項を設ける必要があります。これらの制限条項は，ユーザが，フリー・ソフトウェアの複製物を頒布したり変更しようとする場合には，そのユーザ自身が守るべき義務ともなります。

例えば，あなたがフリー・ソフトウェアの複製物を頒布する場合，有償か無償かにかかわらず，あなたは自分の持っている権利を全て相手に与えなければなりません。あなたは，相手もまたソース・コードを受け取ったり入手できるということを認めなければなりません。さらにあなたは，相手が受領者へそれらの権利を持っているということを，その相手に知らせなければなりません。

我々は次の二つの方法でユーザの権利を守ります。(1) ソフトウェアに著作権を主張し，(2) 本使用許諾の条項の下でソフトウェアを複製・頒布・変更する権利をユーザに与えます。

また，各作成者や我々自身を守るために，本フリー・ソフトウェアが無保証であることを全ての人々が了解している必要があります。更に，他の誰かによって変更されたソフトウェアが頒布された場合，受領者はそのソフトウェアがオリジナル・バージョンではないということを知らされる必要があります。それは，他人の関与によって原開発者に対する評価が影響されないようにするためです。

複写・頒布・変更に対する正確な条項と条件を次に示します。

GNU 一般公有使用許諾の下での複製，頒布，変更に関する条項と条件

1. 本使用許諾は，一般公有使用許諾の各条項に従って頒布されるという著作権者からの告知文が表示されているプログラムやその他の作成物に適用されます。以下において「プログラム」とは，そのようなプログラムや作成物を指すものとし，また,「プログラム生成物」とは，上述した「プログラム」自身，及びその「プログラム」の全部又は一部の作成を，そのまま又は変更して内部に組み込んだ作成物を意味するものとします。本使用許諾によって許諾を受ける者を「あなた」と呼びます。

2. あなたは，どのような媒体上へ複製しようとする場合であっても，入手した「プログラム」のソース・コードをそのままの内容で複写した上で適正な著作権表示と保証の放棄と明確，且つ適正に付記する場合に限り，複製又は頒布することができます。その場合，本一般公有使用許諾及び無保証に関する記載部分は，全て元のままの形で表示して下さい。また,「プログラム」の頒布先に対しては,「プログラム」と共に本一般公有使用許諾書のコピーを渡して下さい。複製物を引き渡す際の実費は請求することができます。
3. 次の各条件を満たしている限り，あなたは,「プログラム」又はその一部分を，変更することができます。更に，上記第2項を満たせば，その変更版を複製したり頒布することもできます。

   (a) ファイルを変更した旨とその変更日とを，変更したファイル上に明確に表示すること。

   (b) 変更したか否かを問わず，凡そ「プログラム」又はその一部分を内部に組み込んでいる作成物を頒布する場合には，本一般公有使用許諾の条項に従って無償で使用許諾すること。但し，頒布先の全て又はその一部の者に対して，あなたが独自に保証することは構いません。

   (c) 変更したプログラムが実行時に通常の対話的な方法でコマンドを読むようになっているとすれば，最も単純，且つ普通の方法で対話的にそのプログラムを実行する時に，次の内容を示す文言がプリント・アウトされるか，或は画面に表示されること。

   - 適切な著作権表示。
   - 無保証であること (あなたが独自に保証する場合は，その旨)。
   - 頒布を受ける者も，本一般公有使用許諾と同一の条項に従ってそのプログラムを再頒布できること。
   - 頒布を受ける者が本一般公有使用許諾書の写しを回覧する方法。

   複製物の譲渡に要する実費は請求できること。また，あなた独自の保証を行なう場合はそれを有償とすることができること。

   本「プログラム」(または，その派生物) と他の別個のプログラムとを，保管や頒布のために同一の媒体上にまとめて記録したとしても，本使用許諾の条項は他の別個のプログラムには適用されません。
4. あなたは，以下のうちいずれか1つを満たす限り，上記第2項及び第3項に従って「プログラム」(または，上記の条項2のもとのその一部分あるいはその派生物) をオブジェクト・コードまたは実行可能な形式で複製および頒布することができます。

   - 対応する機械読み取り可能なソース・コード一式を一緒に引き渡すこと。その場合，そのソース・コードの引き渡しは上記第2項及び第3項に従って行

なわれること。

- 少なくとも3年間の有効期間を定め，且つその期間内であれば対応する機械読み取り可能なソース・コード一式を無償で(ただし，少額の頒布実費は請求できる)提供する旨，及びその場合には上記第2項及び第3項に従って提供される旨を記載した書面を，一緒に引き渡すこと。
- 対応するソース・コードを入手できる所について，あなたが得た情報を提供すること(この選択肢は，営利を目的としない頒布であって，且つあなたがオブジェクト・コードあるいは実行可能形式のプログラムしか入手していない場合にのみ適用される選択項目です)。

上記においてソース・コードとは，変更作業に適した記述形式を指します。また，実行可能形式のファイルに対応するソース・コード一式とは，それに含まれる全モジュールに対応する全てのソース・コードを指しますが，例外として，実行可能なファイルが動作するオペレーティング・システムに付随する標準ライブラリのモジュールのソース・コードやそのオペレーティング・システムに付随する定義ファイルのソース・コードを含ませる必要はありません。

5. 本一般公有使用許諾が明示的に許諾している場合を除き，あなたは，「プログラム」を複製，変更，サブライセンス，頒布，譲渡することができません。本使用許諾に従わずに「プログラム」を複製，変更，サブライセンス，頒布，譲渡しようとする行為は，それ自体が無効であり，且つ，本使用許諾があなたに許諾している「プログラム」の使用権限を自動的に消滅させます。その場合，本使用許諾に従ってあなたから複製物やその使用許諾を得ている第三者は，本使用許諾に完全に従っている限り，引続き有効な使用権限を持つものとします。

6. あなたが「プログラム」(あるいはその「プログラム生成物」)の複製，頒布，変更を行なえば，それ自体で，それらの各行為を行なう権利と，本使用許諾が定める全ての条項に従うことを，あなたが受け入れたものとみなします。

7. あなたが「プログラム」(あるいはその「プログラム生成物」)を再頒布すると自動的に，その受領者は，元の使用許諾者から，本使用許諾の条項に従って「プログラム」を複製，頒布，変更することを内容とする使用許諾を受けたものとします。あなたは，受領者に許諾された権利の行使について，更に制約を加えることはできません。

8. Free Software Foundationは随時，一般公有使用許諾の改訂版，又は新版を公表することがあります。そのような新しいバージョンは，現行のバージョンと基本的に変わるところはありませんが，新しい問題や懸案事項に対応するために細部では異なるかもしれません。

各バージョンは，バージョン番号によって区別します。「プログラム」中に使用許諾のバージョン番号の指定がある場合は，その指定されたバージョンか，又はその後にFree Software Foundationから公表されているいずれかのバージョンか

Yoyodyne, Inc. は，James Hacker が開発したプログラム 'Gnomovision' (コンパイラを起動してアセンブラにつなげるプログラム) についての著作権法上の全ての権利を放棄する。

<TY COON の署名>, 1 April 1989
Ty Coon，副社長

これで手続きは終りです!

INDEX

英数字

あ

か

さ

た

な

は

ま

ら

わ

後書き

最近は **X68000/X68030** の市販ソフトウェアのリリース数が減ってきました。CPU パワーが，インテル系列の CPU を採用しているコンピュータより劣るのは事実ですが，プログラミングの容易さからいえば，**X68000/X68030** は，アマチュアレベルでプロを凌げるプログラミングが可能な機械であり，まだまだ十分に現役といえましょう。

その証拠に，ソフトウェアの数自体は減っていますが，その完成度の高さでは他の機種のソフトウェアを超えている，と個人的には思っています。その **X68000/X68030** のソフトウェアを作成しようと思っておられる方々に，本書が多少でもお役にたてば，著者としてこれ以上の喜びはありません。

最後に，筆の遅い筆者を見捨てないで辛抱強く待ってくださったパソコン言語書籍編集部の方々，初心者として校正に付き合ってくれた妻に感謝致します。また，本書の『付録ディスク』にプログラムを収録することに快く応じてくださった作者の方々に深く感謝致します。

吉野　智興

GCC(ジーシーシー)による
X680x0(エックスロクマンハッセン)ゲームプログラミング

1993年11月29日 初版発行
1994年 2月 5日 第3刷発行

著　者 ····· 吉野(よしの)　智興(ちくみ)
発行者 ····· 橋本　五郎
発行所 ····· ソフトバンク株式会社出版事業部
〒103　東京都中央区日本橋浜町3–42–3
販売　03(5642)8101
編集　03(5642)8143
印　刷 ····· 東京書籍印刷株式会社

Printed in Japan　　ISBN4-89052-472-X