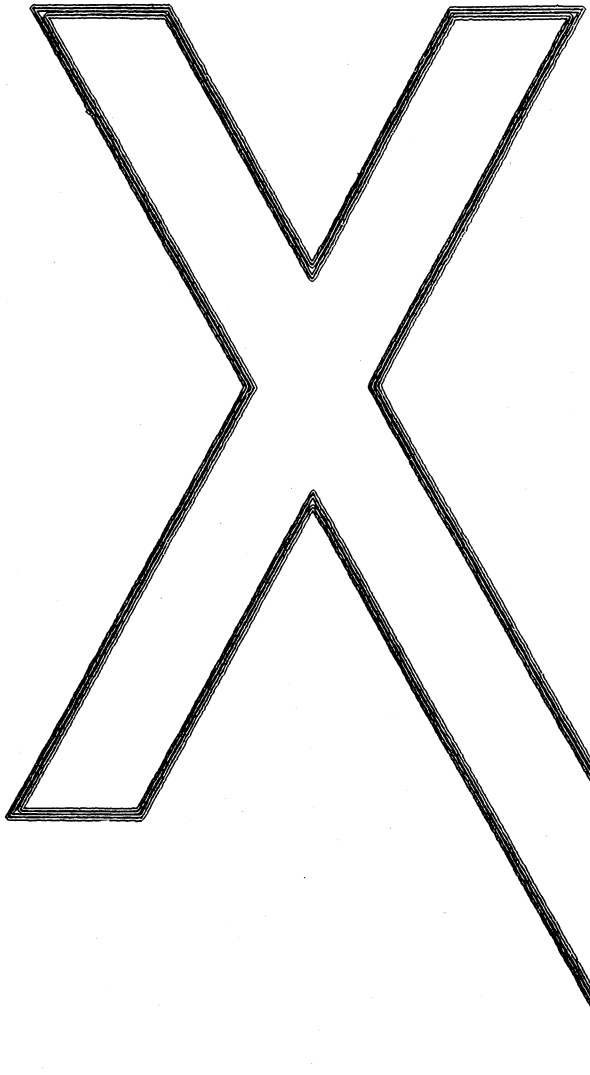


68000
マシン語プログラミング
入門編

著●村田敏幸

まえがき

1990年11月現在，『Oh!X』誌に連載中の「X68000マシン語プログラミング」のうち《入門編》と称した冒頭部分を１冊にまとめたら，こんな本になった。マシン語プログラミングに興味をもったX68000ユーザーのための副読本，とでもいったらよいのだろうか。少なくとも，教科書的なプログラミング入門書では決してない。むしろ，問題集であり，実践テキストのノリに近い。

マシン語にかぎらず，プログラミングに関する知識/技術は，実際のプログラミングの中でこそ身につき，磨かれるものだ。この不変の真理にもとづき，本書は読者に自分の頭と体とを使うことを強いるように書かれている。エッセンスを100倍くらいに薄めて吸い飲みでとろとろと流し込むような親切さは排除した。文章の襞に隠れた大小の謎は，サンプルプログラムを読み，動かし，改良することによって解き明かされるだろう。

もしかすると，遊びの部分の少なさが，この本を必要以上に難解に見せるかもしれない。扱うテーマは文字の入出力や，ファイルの取り扱い，デバイスドライバといった地味なものばかりで，グラフィックやスプライトなどのX68000らしい派手な要素はまるで顔を出さない。しかし，マシン語プログラミングはそれ自体が十分楽しい遊びとなるはずなのだ。

かつて，はじめてコンピュータにふれ，プログラミングをかじり，やがてはマシン語に手を出したころの，毎日が新しい発見ばかりだったあのころの，ワクワクした感じを僕はいまも覚えている。マシン語特有の概念が摑めずに何日も悩んだこともあったが，何かのはずみですっと壁が取り払われる至福の瞬間がそれも忘れさせてくれた。無理に知識を詰め込まれた脳みそが上げる悲鳴すら，心地よかった。

願わくば，この本を読んだ読者の多くにも同じような〝ワクワク〟を感じてもらえますように。

1990年秋

村田敏幸

CONTENTS

COVER DESIGN MASAKI KATSUMATA
COVER PHOTO FUMIO SAITO

CHAPTER 1

マシン語プログラミングの準備

CHAPTER 1

マシン語プログラミングの準備

ASSEMBLER

本書を手にしたからには，もう読者はすっかり〝マシン語する〟気になっているにちがいない。では，さっそく準備にとりかかってもらおう。マシン語プログラミングには何が必要なのか，そんな話をしておく。

マシン語開発ツール　これだけは準備しよう

マシン語プログラミングといっても，それほど特殊なものはいらない。X68000が1台以上と，後はいくつかのツールがあればいい。ツールとしては，ソースプログラムを作成するためのテキストエディタ，ソースをマシン語に変換するアセンブラ，そして，プログラムを分割して開発するときに必要なリンカ，この3つがあればひととおりのことはできる。

●テキストエディタ

Human68k上で使えるテキストエディタには，本体を買うとついてくるED.X，市販品ではWINDEX，FINAL，JAMESなど，さらには公開ソフトのMicroEMACSなどがある。僕はかなり長い間，ED.Xを使ってきたが(あれは，〝可もなし不可ちょっとあり〟の，ごくふつうのフルスクリーンエディタだ)，MicroEMACSがバージョンアップしてかなり高速になった時点で，ぼちぼちと乗り換えつつある。

●アセンブラ，リンカ

Human68kのアセンブラとリンカして標準的なのはAS.XとLK.Xだ。AS.Xは分割アセンブルにも対応した高級なアセンブラであり，アセンブリ言語で記述したソースファイルをアセンブルし，オブジェクトファイルを生成する。分割アセンブルをするときには，このオブジェクトファイルをつなぎあわせて1つの実行形式ファイルを作成するリンクという作業を行う必要があり，それを担当するのがリンカであるLK.Xというわけだ。もっとも，AS.Xの場合は(AS.Xにかぎったことではないが)プログラム作成の手順を統一するために，分割アセンブルをしない場合でもリンカを通すことになっている。この場合，リンカはたんに実行形式ファイルを作るという働きをすることになる。その意味で，リンクまでをまとめて〝アセンブルする〟ということがあるので混乱しないように。

初代X68000(CZ-600C)のシステムディスクには福袋というディレクトリがあって，その中にこっそりとAS.XとLK.Xが入っていたのだが，ACE以降では残念ながら削られてしまった。いまアセンブラ/リンカを手に入れようと思ったら，市販されているものを買ってくるしかない。「C compiler PRO-68K」か，「THE福袋V2.0」のどちらかが必要だ。

C compiler PRO-68KはCコンパイラを中心とした総合開発セットで，'90年9月にバージョンアップされ，Ver.2.0となった。メインはCコンパイラとそのライブラリ一式だが，AS.X，LK.Xも含まれており，マシン語プログラムのデバッグを助けるデバッガDB.X，CプログラムをソースレベルでデバッグできるソースコードデバッガSCD.X，プログラムの開発/保守をサポートするMAKE.X，その他の有用なツール類が含まれる。値段は44,800円(税別)とちょっとしたものだが，コストパフォーマンスは高い。また，ライブラリのソースリスト(アセンブリ言語で書かれている)がついてくるのも見逃せない。マシン語の学習用途にももってこいだ。

マニュアルとしては，『Cユーザーズマニュアル』，『Cリファレンスマニュアル』，2分冊の『ライブラリマニュアル』，『ソースコードデバッガマニュアル』に加え，アセンブラなどの使い方やモトローラMPU68000の命令の解説などが載っている『アセンブラマニュアル』と，Human68kのDOSコール，X68000のIOCSコールの一覧や，X-BASICの外部関数の作り方，デバイスドライバの作り方などなどの情報がまとめられている『プログラマーズマニュアル』の計7冊が添付されている。

THE福袋V2.0は，旧版のC compiler PRO-68KからCコンパイラ関係をばっさり削ったマシン語開発専用セットだ。AS.X，LK.X，DB.X等のツール類と，『アセンブラマニュアル』，『プログラマーズマニュアル』が入って，9980円(税別)という値段は，安い。マシン語を志す人には無条件でおすすめできるアイテムだ。しかし，発売がかなり前なので，すでに入手は困難になっていると思う。

●環境設定

こうして必要なツールが揃ったら，プログラミング開発環境を整えておこう。まず，アセンブラやエディタなどは1つのディレクトリにまとめておきたい。BINに突っ込んでおけばいいだろう。また，ハードディスクを持っている人は作業用のディレクトリを作っておくといい。ハードディスクがなければ，作業用のシステムディスクを1枚作って，必要なツールや作りかけのプログラムを入れておくことをすすめる。この本では，特別なデバイスドライバの組み込みなどは必要としない予定でいるから，そのあたりはあまり気にすることはない。ただ，作業はCOMMAND.X上で行うことになるので，コマンドモードでのオペレーションにはある程度慣れておいてもらったほうがいい。

参考書について

プログラム作りにはなんらかの資料が欠かせない。BASICを使うときにはBASICのマニュ

アル，Cを使うときにはCのライブラリマニュアルといったぐあいにだ。マシン語プログラミングをするときにも，それなりの資料を用意しなければならない。X68000上でマシン語プログラムを作るとなると，プロセッサであるモトローラ68000の機能や命令の働きをまとめた参考書とHuman68kのDOSコール，IOCSコールの一覧がいるだろう。ゆくゆくは，X68000のハードウェアの解析書もほしくなるかもしれない。

●マニュアル

まず手近なところでマニュアルがある。C compiler PRO-68KやTHE福袋V2.0に含まれる『アセンブラマニュアル』と『プログラマーズマニュアル』の2冊があれば，MPU68000のマシン語からDOSコール・IOCSコールの一覧までの情報がほとんど得られる。また，『Human68kユーザーズマニュアル』にあるコントロールコードやエスケープシーケンスの表，取扱説明書などに見られる漢字コード表，なにかを見れば必ず載っているASCIIコード表など，マニュアルを参照する機会は多い。

●MPU68000の解説書

68000は昔から解説書の数だけは多いから，ちょっと大きな書店にいけば何かしらみつかるだろう。ここでは，とりあえず『M68000マイクロプロセッサ・ユーザーズ・マニュアル』(CQ出版社)，『68000プログラマーズ・ハンドブック』(技術評論社)をすすめておく。

●X68000の資料

さて，68000の解説書の次は，X68000本体の資料だ。プログラマーズマニュアル以外でDOSコールやIOCSコールがまとめられているものというと，『X68000環境ハンドブック』(工学社)がある。ある程度は独自解析して未公開のシステムコールなども拾ってあるようで，リファレンスとしてはまとまっているような印象を受けた。また，少し古くなるが，『X68000データブック』(小学館)もある。この本はおもにX68000のハードウェアの資料を並べたもので，システムコールの解説はどちらかというとオマケだが，1冊でハード・ソフト両方の資料が載っているというのはお得かもしれない。さらには，元祖X68000の『Human68kユーザーズマニュアル』には，付録としてHuman68kのDOSコール一覧が載っており，リファレンス的な使い方には十分役に立つ。

後は将来，ハードウェアの解説が必要になれば，さきほどの小学館の本か，『X68000テクニカルデータブック』(3000円，アスキー)ということになるのだが，どちらも古い本だから手に入れるのは難しいかもしれない。古本屋で探してみるといいだろう。

COLUMN

僕がマシン語にこだわるわけ

熱力学の第2法則ではないが，「ほんとは誰でも楽をしたい」の法則により，多くの人はCなどのコンパイラ言語へと流れていく。

ところが，時代に逆らって「ヤだ，ヤだ，ヤだ。マシン語じゃなきゃヤだ」と意地を張る，僕のような人間もいるわけだ。僕自身，それがなぜなのかは，あまり考えてみたことがない。はっきりしているのは，たんに速度が速いとか，プログラムの大きさが小さくな

るといった問題ではないということだ。強いていうなら，自分のマシンを直接ドライブすることに魅力を感じているのかもしれない。

マシン語プログラムは，プロセッサの命令と１対１に対応するかんたんな略語からなるアセンブリ言語を用いて開発される。このアセンブリソースプログラムをアセンブラというプログラムに通すことで，実行可能なマシン語プログラムができあがる。手順そのものはコンパイラ言語での開発とほとんど変わらないし，最終的にマシン語プログラムになるのも表向き同じだ。しかし，コンパイラがソースプログラムをいくつもの命令の組み合わせに「翻訳」するのに対して，アセンブラはアセンブリソース上の一命令をマシン語の一命令に単純に「変換」するだけだから，ソースプログラムの一命令に対するプログラマの思い入れ(のようなもの)は，かなり違ってくる。

自作のマシン語プログラムが走るとき，プロセッサの動作は完全に僕の手のうちにある。どの瞬間を切り出してみても，プロセッサは僕がプログラムに書いたとおりの動作をし，そして，それ以上のことはなにもしない。言葉ではどうもうまくいえないけれど，僕はマシン語プログラミングのそんなところに惚れちゃってるのだ。

CHAPTER 1

マシン語プログラミングの流れ

CHAPTER 1

マシン語プログラミングの流れ

ASSEMBLER

早いもので，X68000が発売されてからもう４年近くになる。凄まじいほどの勢いで技術が進歩し，ピカピカの新機種も，あっという間に旧機種になってしまうというパソコン界の常識に照らしてみれば，そろそろ古臭さや限界が見えはじめてもいいころだ。

が，X68000はいまだ新鮮で，マシンのパワーは底知れず，ことあるごとに僕らユーザーをウキウキさせたり，ワクワクさせたり，ドキドキさせたり，ゾクゾクさせたりしてくれている。これは，よいことだ！　と思う。「さすがは夢を超えたマシン」とでもいって喜ぶべきことなのだろう。

ただ，新鮮さは未発達であることの裏返しでもある。圧倒的なマシンパワーという言葉の影には，おいてきぼりにされたユーザーの姿が目に浮かぶ。これは，あまりよいことではない，ような気がする。もしかすると，ちょっと慌てなければならない事態なのかもしれない。

X68000が発売された当時，本体付属のソフト以外には１本のソフトもないような状況だったにもかかわらず，X68000ユーザーはなんの心配もしていなかった。なぜなら，X68000は自分たちの手で育て上げるマシンだ，という思いがあったからだ。こういう思い入れ(ないしは思い込み)こそがマシンを発展させる原動力といえる。

残念なことに市販ソフトがある程度出揃った時点で，初期ユーザーの持っていた熱意も薄れてしまったように思う。安心してしまったのかもしれない。そのせいかどうか，X68000のユーザーは重箱の隅をつつき回すマニア中のマニアと，テクニカルなこととは無縁な人とに極端に二分化したように見える。

そろそろ原点に戻るときじゃないか。

いまこそマシン語に取り組もう

マシンを自分たちの手で育てるということが，必ずしもプログラミングに結びつくものではないことは承知している。ユーザーの声をメーカーやソフトハウスに伝えることも，マシンを育てる意義を持つだろう。しかし，他人を自分の思うように動かすのは，自分で何から何までやることと同じくらい難しいことでもある。

どちらも同じように難しいのであれば，他力本願よりも自分たちでバンバンとプログラムを作って，X68000をもっともっと育ててあげようというのが親心というものだ。

マシン語というと，「まだちょっと……」と尻込みする人も多いだろう。けれど，もし，その気があるのであれば，非力な僕でも少しくらいのお手伝いはできると思っている。

というわけで，かなり大上段に構えた気配があるし，なかばゴリ押しっぽい部分もあるが，以上，マシン語プログラミングのすすめであった。

X68000と68000

X68000の名前には，プロセッサであるモトローラ68000の名が誇らしげに埋め込まれている。68000[1]というマイクロプロセッサは，16Mバイトにおよぶ広大でリニアなアドレス空間を持ち，その命令セットは強力で，豊富なアドレッシングモードを備え……，というような話は何かを見れば書いてあるだろうから，ここでわざわざ広報活動をする必要もないだろう。そんなことはいわなくたって，68000が優秀なマイクロプロセッサであることは誰もが認めている。68000にめぐりあったあなたは，めぐりあったこと自体，幸せなことなのだ。

■ 1) モトローラが1979年に発表した16ビットマイクロプロセッサユニット。X68000には，クロック10MHzのCMOS版68000が使用されている。

……と，ここまで書いた僕は，X68000のネーミングの重大な欠点を見つけてしまった。「こういう文章を書くときに，パソコン本体の名前とプロセッサの名前が紛らわしい」のだ。そこで読者の無用な混乱を避けるために，ここで本文中での使い分けを決めておこう。今後，X68000はフルスペルで「X68000」と書き，プロセッサ名はたんに「68000」と書くこととする。

いきなりプログラミング

さて，マシン語入門の事始めだ。さっそくだが，プログラムを作成してみることにしよう。

プログラムの開発手順は，だいたい図1のようになる。これはもっとも一般的かつ単純な例であり，実際にはこれ以外のステップが必要な場合もあるが，基本的な流れはこの図のとおりだ。

以下，この図にそってプログラミングの過程を追いかけてみよう。

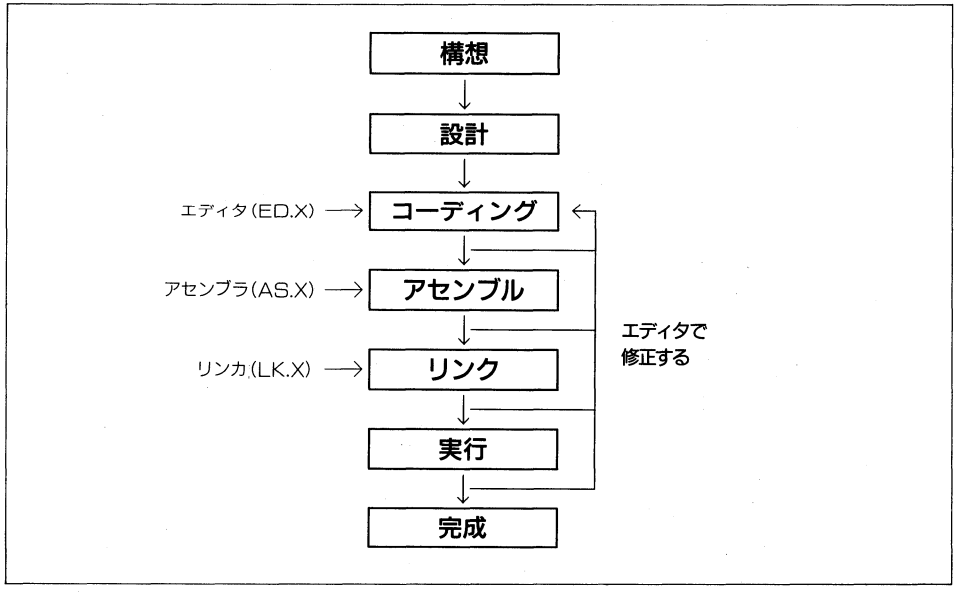


図1
プログラム開発の
基本的流れ

1 構想

まず，どんなプログラムを作るかを考える。うーん，当たり前すぎて，これ以上何も話すことがない。ここでは，「画面に1文字の〝a〟を表示する」というありがちな例を取り上げよう。

2 設計

プログラムの細かな仕様を決め，その処理を具体的にどのように行うか，そのアルゴリズムや必要な要素を検討する。ステップとしては，プログラム全体を大まかな処理に分解し，それぞれの処理をさらに細かな処理に分解する……というようにプログラムを細分化する方向で考えるとわかりやすいだろう。プログラムをサブルーチンに分け，サブルーチンをさらに小さなサブルーチンに分解するといってもいい。

最終的には，プログラムは小さな処理単位にまで分解される。ここまでやってしまえば，プログラムの現物は影も形もないが，事実上プログラムは完成してしまうことになる。

設計は頭の中でやるもよし，紙に書くのもよし。ED.Xなどのフルスクリーンエディタを使ってメモを書いてもいいだろう。フローチャートを書く……必要は全然ないが，書けといわれたら書ける程度には煮詰めたい。

ここでサンプルとするプログラムの場合だと，あまりにプログラムの規模が小さすぎるので，

これ以上の細分化はほとんどできない。あえて分けるとすれば,「画面に１文字表示する処理」と，忘れてはならない「プログラムの実行を終了する処理」に分解される。

この程度の処理なら，Human68kに用意されているDOSコールを利用するだけでできる。前章でも紹介した『プログラマーズマニュアル』などで調べれば，それぞれの処理はDOSコール＄FF02のputchar，＄FF00のexitで行えることがわかり，幸いにも，具体的な呼び出し方まで載っている。後はマニュアルを写すくらいのつもりでも，なんとかプログラムが書けそうだ。

COLUMN

DOSコール

Human68kにはDOSコールと呼ばれるサービス群がある。DOSコールには主として画面やキーボード，ファイルなどの入出力を行う機能が詰め込まれており，どんなプログラムにも必ずといっていいほど必要になる文字の入出力やファイル処理などを手軽に行う手段として用意されている。

もしDOSコールがなければ，ファイルから１文字読み込むような処理を行うにも膨大な手間がかかるし，それなりの知識も必要になる。しかし，DOSコールを利用すれば，ファイルをオープンするDOSコールを呼び出してから，１文字入力のDOSコールを呼び出すという，それこそ高級言語の関数を使う程度の労力ですみ，プログラマはプログラムの別の部分に専念できるというわけだ。

また，DOSコールには入出力の処理を統一的に記述するという意味もある。DOSコールを使って作成したファイルは，当然他のプログラムで読むこともできる。もっとも，それこそがOS(というかDOS)の存在意義でもある。逆にいうと，OSの上でプログラムを動かすからには守らなければならない最低限の約束事があるということだ。

DOSコールはプログラムとOSのインタフェイスであり，また，プログラムどうしのインタフェイスでもあるのだ。

そういう理由もあって，この本では，DOSコールを使った入出力処理をみっちりとやることにしている。X68000のマシン語プログラムというと，すぐにグラフィックなどの派手な部分に目がいき，高速化のために(もしくはたんなる気分で)DOSやIOCSを無視してハードウェアに直接ふれるようなプログラムが題材として取り上げられるが，それに対するささやかな抵抗のつもりも少しある。

3 コーディング

ここからがプログラムを実体のあるものにしていく作業だ。マシン語プログラムのソースプログラムをエディタで作成する。設計がしっかりしていれば，コーディングはごく単純な作業になるだろう。逆に，この段階で四苦八苦するようであれば，設計が甘かったということだ。

さて，画面に〝a〟を表示させるプログラムは，とりあえずリスト１-aのような感じになる。

リスト1-a
SAMPLE1.S

```
1:          move.w  #'a',-(sp)
2:          .dc.w   _PUTCHAR
3:          addq.l  #2,sp
4:          .dc.w   _EXIT
```

では，エディタを立ち上げて，試しにこのプログラムを打ち込んでもらいたい。新規ファイルを作成するので，適当なファイル名をつける。かりに〝SAMPLE1.S〟とでもしておこう。Human68kではアセンブリソースファイルの拡張子は〝.S〟にする習慣なので，それにしたがっている。このファイルをエディットするには，本体付属のED.Xの場合だと，

```
A>
```

などのプロンプトの後に，

```
A>ED SAMPLE1.S
```

とコマンドを与える。

エディタが立ち上がったら，リストどおり正確に打ち込む。空欄の部分はスペースを適当な数入れてもいいが，TABを使えば桁位置をかんたんに揃えることができる。

中には意味不明のリストを打ち込むことに抵抗を感じる人もいるだろうから，ここで，かんたんにプログラムの解説をしておく。最初の3行は，Human68kのDOSコールを使って1文字表示するときの決まった手順で，表示したい文字コードをスタックに積み，DOSコールputcharを呼び出し，その後，もういらなくなったスタック上のデータを捨てている。4行目は，これまたHuman68k上のプログラムを終了するときの決まりで，DOSコールexitを呼び出している[2)]。

■ 2) リストはPR.Xを使って打ち出してあるので，行頭に行番号がついているが，打ち込む際には行番号の後ろ以降，枠線内のみを入力すればよい。これは以後の掲載リストでも同様だ。

入力がすんだところで，テキストをセーブし，エディタを抜ける。ED.Xの場合，ESCキーを押して(一度離してから)〝E〟キーを押せば，テキストがセーブされ，コマンドモードに戻る。

COLUMN

スタック(1)

スタック(stack)とは，「データを積み上げるようなデータ構造」，または「そのようにしてデータを積み上げておく場所」のことをいう。というと，なにやら複雑そうに見えるが，考え方はかんたんだ。データを本に見立てると，本を1ヵ所に積んだり下ろしたりするようなものと思えばよい。

本を何冊か順番に積んでいくと，最初に積んだ本が底になり，いちばん上には最後に積んだ本がある。積んだ本の山を崩さずに本を取ろうとすれば，当然，いちばん上の本を手にすることになるだろう。また，底の本を取るためには，その上に積まれた本をすべて取り去る必要がある。

このように，スタックは最後に積んだデータが最初に出てくるので，「LIFO(Last In First Out)バッファ」とも呼ばれる。また，スタックにデータを積むことを「プッシュ(push)

する」といい，データを取り出すことを「ポップ(pop)する」という。

もちろん，〝積む〟とか〝下ろす〟というのは比喩的な表現で，パソコンの中に現実にデータを積み上げるわけにはいかないから，適当なメモリ領域をスタックとして使う。そして〝最後にスタックに積んだデータ(＝次に取り出されるデータ)のスタック上のメモリ番地(アドレス)を格納しておくスタックポインタ(stack pointer)と呼ばれる変数〟を用意しておき，この変数が〝積まれたデータの山のいちばん上(スタックトップ)〟を指していると考える。

後は，スタックポインタの値を減らして，その新たなスタックポインタの位置にデータを書き込めばプッシュしたことになり，スタックポインタの指す位置からデータを取り出してからスタックポインタの値を増やせばポップしたことになる(多くの場合，スタックはアドレスの小さいほうへ伸びていくように構築される)。

BASICなどでも，配列を仮想的なスタックとして使うことがたまにある。たとえば，

```
dim int STACK(100)
int SP=100
```

というように，配列と，スタックポインタとして使う変数を1つ用意しておく。この例では，最初スタックポインタSPの値は100だから，STACK(SP)はSTACK(100)のことだ。つまり，SPは配列STACKの100番目をポイントしている。

ここで，

```
・SP=SP-1:STACK(SP)=1
```

により，STACK(99)に1が格納され，SPは99になる。さらに，

```
SP=SP-1:STACK(SP)=2
```

によって，STACK(98)に2が格納され，SPは98になる。ここまでの手順で，この仮想的なスタックには1と2が順に積まれたことになる。

ここで，

```
A=STACK(SP):SP=SP+1
```

とやれば，Aにはスタックトップに積まれていた値である2が入り，スタックトップには1が残る。さらに，

```
A=STACK(SP):SP=SP+1
```

により，Aには1が入って，スタックは空になるわけだ。

また，上のようにスタックに1と2を順に積んだところで，

```
SP=SP+1
```

を実行すると，次に，

```
A=STACK(SP):SP=SP+1
```

によってポップされるデータは1だ。つまり，スタックポインタの値を強制的に増やしたので，後から積んだ2の値は捨てられてしまったことになる。

マシン語プログラムにおいては，データを一時的にどこかにしまっておくときにスタックが用いられる。スタックを使うと，データをしまう場所をとくに用意する必要もなく，取り出す順番さえまちがわなければ，複数のデータを待避することができるなど，いろいろ便利である。

さらにHuman68kのDOSコールでは，引数の受け渡しにスタックを用いることになっている。本文のリストに見られるDOSコールputcharの呼び出し方と，ここでのスタックの説

明を見比べてみると，何かが見えてくると思うが，どうだろうか。

4 アセンブル

エディタで作成したソースプログラムはたんなるテキストファイルであり，直接実行できるプログラムではない。そこで，ソースをマシン語(オブジェクト)に変換する「アセンブル」という作業を行う必要がある。

アセンブルを行うプログラムがアセンブラで，Human68k上での標準的なアセンブラがAS.Xだ。AS.Xは，コマンドモードからコマンド名に続いて，アセンブルしたいファイル名を入力することで起動する。この例の場合，

```
A>AS SAMPLE1.S
```

となる。ここで，アセンブルするファイルの拡張子が〝.S〟の場合は，次のように拡張子を省略してもよい[3]。

■ 3) AS.Xでは，アセンブルするファイルの拡張子.Sが省略でき，LX.Xではリンクするファイルの拡張子.Oが省略できる。

```
A>AS SAMPLE1
```

エラーなく，アセンブルが終了すると，

```
No Fatal error(s)
```

というメッセージが表示され，ソースファイル名の拡張子を〝.O〟にしたオブジェクトファイルが生成される。

ところが，実際にはミスタイプや勘違い，見落としなどにより，正常にアセンブルされず，かわっていくつかのエラーメッセージが表示されることがある。というより，エラーなしにアセンブルできることのほうがめずらしい。じつはさきほどのプログラムも見事にエラーを出してくれる。たぶん，次のようなエラーメッセージが現れるはずだ。

```
line     2 undefined symbol error
line     4 undefined symbol error
undefined symbol
_PUTCHAR
_EXIT
    2 Fatal error(s)
```

2行目と4行目に未定義の「シンボル」があり，その未定義のシンボルは_PUTCHARと_EXITで，計2つの致命的なエラーがあった，という意味だ。

このエラーメッセージを頼りにソースを修正すると，リスト1-bのようになった。もちろん，

リスト1-b

```
1: _PUTCHAR         equ     $ff02
2: _EXIT            equ     $ff00
3: *
4:          move.w  #'a',-(sp)
5:          .dc.w   _PUTCHAR
6:          addq.l  #2,sp
7:          .dc.w   _EXIT
```

ソースの修正にはふたたびエディタを使う。

リスト1-aでは未定義のシンボルがあったわけだから，それらのシンボルを定義してやればよい。DOSコールの名前は68000のマシン語の命令ではないので，定義してはじめてアセンブラに理解してもらえるようになるわけだ。1～2行目がプログラム中で使用するDOSコール名の定義だ。ちなみに，3行目の〝*〟は，コメント行の印だ。

修正がすんだら，再度アセンブルする。今度は，

No Fatal error(s)

のメッセージとともに無事アセンブルが完了し，〝SAMPLE1.O〟という名前のオブジェクトファイルが生成されているだろう。DIRコマンドでディレクトリをとって確認しておこう。

5 リンク

小規模なプログラムならどうということはないが，プログラムの規模が大きくなるにつれ，プログラム全体の見通しが悪くなり，アセンブルする時間も長くなってくる。そこで，プログラムを複数のソースに分割し，それぞれをアセンブルしたうえで最終的につなぎあわせて実行ファイルを作成するという方法がとられることになる。

すでにアセンブルできた部分は何度もアセンブルする必要がなくなるから，トータルでの開発時間も短く押さえられる。また，以前作ったプログラムで動作が(バグがないことが)確認されているパーツを新しいプログラムに組み込んで使うといったこともできる。

このプログラムをつなぎあわせる作業のことを「リンク」という。ただ，実際には，開発手順を統一するなどの理由で，ソースを分割していなくてもつねにリンクを要求するアセンブラも多く，AS.Xの場合もそうなっている。この場合，リンクは，たんに実行形式(実行可能な)プログラムを作成するという程度の意味になる。

Human68kにおける標準的なリンカ(リンクを行うツール)がLK.Xだ。コマンド名に続いて，オブジェクトファイル名をコマンドラインに与えることによって起動する。例の場合，

A>LK SAMPLE1.O

だ。ここでも，次に示すように，オブジェクトファイルの拡張子が〝.O〟のときは拡張子を省略することが許される。

A>LK SAMPLE1

ソースを分割して開発しているのでなければ，リンクの段階でエラーが出ることはまずない。

プログラムがちょっと大きくなるとリンクするのにかなり時間がかかるようになるが，のんびり待ってあげよう。リンクが終了すれば，オブジェクトファイル名の拡張子を〝.X〟にした名前で，待望の実行可能ファイルができあがる。

6 実行

これまでの作業で作成された実行可能ファイルは，そのファイル名をコマンドラインに与えることによって実行される。では，

```
A>SAMPLE1
```

と打ち込んで，いま作ったばかりのプログラムを実行してみよう。うまく動かなかった場合はふたたびソースの修正に戻ることになるが，サンプルプログラムはちゃんと動作しているようだ。

以上の作業は，これからもプログラムを作るたびに行うことになる。作業に慣れる意味で，本章ではさらに同程度の規模のプログラムをいくつか作っていくことにする。

コントロールコードを出力する

リスト1-bを実行すると，改行されずに次のプロンプトが出て，

```
aA>
```

のように表示されるのがちょっと気になる。このままではあまり美しくないので，ちゃんと改行するようにしてみたい。

Human68kでは，$0D_H$，$0A_H$という2つのコードで改行が行われる。すなわち，〝a〟を表示した後，これら2つのコードをputcharを使って出力してやれば改行されるだろうことが想像できる[4]。

■ 4) 改行などの画面表示制御は，コントロールコードを使って行う。ある決められたコントロールコードを画面に出力することによって，さまざまな画面制御が行われる。

1文字表示するDOSコールputcharの使い方はもうわかっているから，〝a〟を表示したのと同様の手順を$0D_H$と$0A_H$に対して繰り返せばプログラムはすぐにできあがり，リスト2のようになる[5]。

■ 5) Human68kユーザーズマニュアルを開くと，付録1にASCII制御コードの一覧がまとめてある。これを見ると，$0D_H$（Hは16進数の意味。なお，$0D_H$を10進数になおすと13になる）でカーソルを行の先頭に移動し，$0A_H$（10進数では10）によってカーソルが1行下に移動することがわかる。

リスト2

```
 1: _PUTCHAR        equ     $ff02
 2: _EXIT           equ     $ff00
 3: *
 4:          move.w  #'a',-(sp)
 5:          .dc.w   _PUTCHAR
 6:          addq.l  #2,sp
 7:          move.w  #$0d,-(sp)
 8:          .dc.w   _PUTCHAR
 9:          addq.l  #2,sp
10:          move.w  #$0a,-(sp)
11:          .dc.w   _PUTCHAR
12:          addq.l  #2,sp
13:          .dc.w   _EXIT
```

この要領で,

ab

の2文字を表示し，改行するプログラムとか,

a

b

と縦に表示するプログラムなどのバリエーションがいくらでも作れるだろう。いろいろ試してみてもらいたい。

複数の文字を表示する

表示する文字が2文字，3文字と増えるにしたがって，putcharをだらだらと並べるのが面倒になってくる。それなら，まとめて表示する方法はないのかと『プログラマーズマニュアル』で調べてみれば，DOSコール＄FF09にprintという文字列を表示する機能がある。さっそく，このDOSコールを使ってみようとマニュアルをさらに読むと，いきなり「MESPTRで指定したポインタ以降，ヌル文字(0)までの文字列を表示します」などと書いてあって挫折しそうになるかもしれない。それでも，呼び出し例が書いてあるから，なんとかこれを頼りにプログラムを書いてみることにする。きっとリスト3-aのようになるだろう。

リスト3-a

```
 1: _PRINT          equ     $ff09
 2: _EXIT           equ     $ff00
 3: *
 4:          pea     mes
 5:          .dc.w   _PRINT
 6:          addq.l  #4,sp
 7: mes:     .dc.b   'PRINT TEST',$0d,$0a,0
 8:          .dc.w   _EXIT
```

アセンブル，リンクはエラーもなく終わる。ところが，いざ実行してみると，たしかに文字列は表示されるのだが，その直後に画面中央にどーんと，

```
おかしな命令を実行しました
```

という見慣れないメッセージが出て驚くことになる(メッセージにしたがって〝A〟のキーを押せばコマンドモードに戻る)。

表示自体はうまくいっているが，どうもその後がいけないようだ。

「おかしな命令」の謎を探るため，マシン語プログラムを実行するとはどういうことか，という話をしておこう。

僕たちがアセンブリ言語で書いたソースプログラムは，アセンブル&リンクを経てマシン語プログラムに変換される。マシン語プログラムの実体は，よくいわれるように人間には解読できそうもない意味不明の数字の列だ。68000は，この数字列を順にメモリから読み出してはそれがどんな命令か調べ，実行する。

ここで，リスト3-aを見直してみよう。頭の2行はシンボルの定義というアセンブラに対する情報であり，3行目はコメント行だから，プログラム本体は4行目から始まる。

68000は4～6行で文字列を表示した後，7行目の表示する文字列データにさしかかる。人間にはこれがデータだということがわかっているが，マシン語プログラムの中ではすでに命令もデータもただの数字になってしまっているから，68000には区別がつかない。そこで，ともかく順番だからというので，データを命令だと思って読み込み，実行しようとするわけだ。

運よくそのデータを命令として解釈できれば，68000はそしらぬ顔で実行し，次に進む。しかし，すべての数字の組み合わせが命令として定義されているわけではなく，解釈不可能な場合もある。すると，68000は「未定義の命令じゃないか」といって怒るわけだ。その結果がさきほどのエラーメッセージである[6]。

■ 6)プログラムとデータを区別できないというのは,現在のコンピュータが生まれたときから背負っている欠点といえ,マシン語プログラムが暴走する原因の1つにもなっている。

これが68000ではなく，Z80など，他のマイクロプロセッサの場合だと，たんにスキップしてさらに危険な領域に突っ込んでいき，暴走することもある。が，68000は，ある程度のチェックを行う機能を持っているので，マシン語プログラムがかんたんに暴走するという常識を覆してくれる。

ただし，このチェックは万全ではない。データが偶然にも命令として解釈され，その命令によりプログラム本体を破壊してしまう可能性もあるからだ。

さて，リスト3-aがうまく動かなかったのはプログラムの途中にデータを挟んだのが原因だったのだから，データをプログラムから分離してしまえばよいだろう。リスト3-bのように，データをプログラムの最後に移動させれば正しく動くようになる。

リスト3-b

```
1: _PRINT           equ     $ff09
2: _EXIT            equ     $ff00
3: *
4:          pea     mes
5:          .dc.w   _PRINT
6:          addq.l  #4,sp
7:          .dc.w   _EXIT
8: mes:     .dc.b   'PRINT TEST',$0d,$0a,0
```

この文字列は，mesというシンボル(ラベル)によって位置がわかるようになっているから，命令の邪魔にならないところなら，どこにあってもよかったのだ。

キーボードからの入力

表示ができるようになったら，今度は入力を扱ってみよう。キーボードから1文字取り込み，入力された文字を表示するプログラムを作ってみる。表示の部分はいつものようにputcharを使うとして，入力を行うDOSコールを調べておこう。DOSコール$FF08にgetcというファンクションがあるので，これを使うことにする[7]。

■ 7) ところで，DOSコールにはgetc以外にも1文字キー入力を行うものがいくつかある。$FF01getchar，$FF07inkeyなどだ。マニュアルを見ると，getcharはエコーつき，inkeyはブレイクチェックつきと書いてあるが，これらが具体的にどう違うかは自分で試してみてほしい。getcとgetcharの違いはすぐにわかると思う(エコーバックがあるかないか)。
getcとinkeyの違いは，入力待ちのときにBREAKキーを押すと見えてくるはずだ(ブレイクチェックがあるかどうかの違いだ)。

マニュアルによると，入力された文字はd0レジスタに「キーコード」として返されるらしい。キーコードなどと書いてあるが，これはどうやらASCIIコードのことを指すようだ。「d0レジスタ」というのも，キーコードを格納するいれものと考えればよい。高級言語の変数のようなものと思っても差し支えないだろう。

と，比較的いいかげんに想像をめぐらせておき，とにかくプログラムにしてみる。リスト4だ。

リスト4

```
1: _GETC            equ     $ff08
2: _PUTCHAR         equ     $ff02
3: _EXIT            equ     $ff00
4: *
5:          .dc.w   _GETC
6:          move.w  d0,-(sp)
7:          .dc.w   _PUTCHAR
8:          addq.l  #2,sp
```

```
9:          .dc.w   _EXIT
```

実行すると，カーソルが点滅し，キー入力待ちになるから，適当なキーを押す。

```
aA>
```

のように，入力した文字が表示されれば成功だ。表示の後の改行を行っていないので，気分によっては改行処理を付け加えてもいいだろう。

本章はここまで。ついに用語や個々の命令の解説をろくにせずに終わってしまった。文中にはずいぶん未定義の用語が氾濫しているし，プログラムにもまだまだ不明確な部分が多い。それでも，プログラム全体の流れはつかめているだろうし，用語の意味だって前後から想像することはできるだろう。

たとえば，putcharに与える文字コードを#'a'と書いたり，#$0dと書いたりしているのを見れば，「'a'のように文字をシングルクオーテーションで囲むと，その文字の文字コードになる」とか，「アセンブラでは16進数は頭に〝$〟をつけて表す」ということがわかるにちがいない。

きっと読者はそうやっていろいろ考えながら，ここまで読み進んできたものと思う。そして，これからもそうであることを期待する。

自分で考えること，試すことを怠らなければ，マシン語といえどもそれほど難しいものではない。最初のうちは考えたり，試したりすることが面倒に思えるかもしれないが，いつの間にか面倒どころか楽しくなってくるはずだ。そうなれば，後はとんとんと上達していくことだろう。

さて，次章ではもう少しプログラムらしいプログラムを作成し，あわせて68000のマシン語の命令のいくつかを紹介しよう。

COLUMN

標準入力・標準出力

本文では，画面に表示するとか，キーボードから入力するという表現を使っているが，じつはこれは正しくない。正確にはputcharは「標準出力に１文字出力する」，getcは「標準入力から１文字入力する」という働きをする。ただし，ふつうの状態では，標準入力はキーボードに，標準出力は画面にと割り当てられているというのが一般的だ。実際にはリダイレクトすることによって，標準入出力をファイルやプリンタなどに切り替えて使うことができる。

試しに，リスト３-bをアセンブルしたプログラムを，

```
A>SAMPLE3 >TEMP
```

のように(ファイル名は，かりにSAMPLE3としてある)，コマンドラインの最後に〝>TEMP〟をつけて実行してもらいたい。画面には何も表示されないが，ディレクトリをとってみると，〝TEMP〟というファイルができているのがわかるだろう。このファイルを，

```
A>TYPE TEMP
```

のようにTYPEコマンドを使って表示してみると，〝PRINT TEST〟という文字列が格納されていることがわかる。つまり，〝>〟を使ってリダイレクトすることにより，標準出力

が〝TEMP〟というファイルに切り替えられたのだ。

また，いまできたファイル〝TEMP〟を残したまま，リスト4をアセンブルしたプログラムを，

```
A>SAMPLE4 <TEMP
```

として実行してみよう。今度は画面に〝P〟の文字が表示される。これは，ファイル〝TEMP〟の先頭の1文字目だ。このように〝<〟は標準入力を切り替える働きをする。

Human68k上では，CONやPRN，AUXなどのデバイス名もファイル名のかわりに使える。たとえば，

```
A>SAMPLE3 >PRN
```

とすれば，プリンタに印字することができる。

CHAPTER 2

68000の基本命令を覚えよう

CHAPTER

68000の基本命令を覚えよう

ASSEMBLER

本章では，よく使われる68000の命令に加えて，以後の話の基礎となるマシン語特有の概念・用語がいくつか登場する。じわじわとマシン語のおもしろい部分，いやらしい部分が見えてくるだろう。

たぶん，この章は本書の，そしてマシン語の〝最初にして最後，かつ最大の〟山場となるだろう。一読した程度ではすべてを摑みきれないかもしれない。が，それはそれでよい。むしろ，最初から全部詰め込もうとしてパンクしてしまうよりはずっとましだ。時間をかけて，ゆっくり消化していってもらいたいと思う。

ここで，1つだけアドバイスしておこう。無理に用語を覚えようとしてはいけない。そんなものは何度も出てくればそのうち自然に頭に入る。それよりも，〝それ〟がどんなものなのかという概念・イメージ，なぜ〝それ〟が必要なのかという存在理由を考えながら読み進んでほしい。

では，まずは2つのキーワード，「アドレス」と「レジスタ」を軽く押さえておく。

アドレス

〝箱〟がいっぱいある。これらの箱はどれもみんな同じ大きさ形をしているので，たがいに区別することができない。これを区別できるようにしたいとする。誰でも考えつくように，こんなときは，箱を1列に並べ，端から順に番号を振ればよい。こうして振られた番号は，ある特定の箱1つを指定する手段として，ひいては間接的に〝箱の中身〟を指定する手段として利用できるだろう。

このたとえは，コンピュータのメモリ(memory)とアドレス(address：番地)の関係を表している。〝箱〟はメモリの一要素であり，〝箱につけた番号〟がメモリ上の位置，すなわちアドレスである。そして，〝箱の中身〟がメモリに格納されたデータにあたる。なお，ふつうの感覚では番号は1から数えるが，コンピュータの世界では数字は0から数える慣例になっており，アドレスも0番地から始まる。

いま一度，箱のたとえに戻る。箱のうちのどれか1個に注目しよう。その箱を基準と考えると，〝基準の箱から3個手前〟とか〝5個後〟というように〝基準となる箱の番号との差〟によ

っても，特定の箱を指定できることがわかるだろう。アドレスも〝ある基準となるアドレスとの差〟で表現することがあり，これを「相対(relative)アドレス」と呼ぶ。また，〝箱につけられた番号〟のほうは固定的，絶対的であるので「絶対(absolute)アドレス」と呼ぶことがある。

レジスタ

「レジスタ(register)」は，プロセッサ内部に用意された小規模なメモリだ。感覚としては，高級言語でいう変数のようなものと思ってもよいかもしれない。ただし，高級言語の変数とはっきり違うのは，名前や，ときには用途までもが決まっているということだ。

プロセッサが動作するためには，ある程度の情報が必要だ。たとえば，いまメモリのどの位置の命令を実行しているのかがわからなければ迷子になってしまうし，自分(プロセッサ)が現在どんな状況に置かれているのかという細々とした情報もいるだろう。そういった情報をしまっておく場所として，プロセッサ内部には特定用途のためのレジスタが用意されている。実行すべき命令の位置(アドレス)はプログラムカウンタ(pc：Program Counter)と呼ばれるレジスタに，現在状況はステータスレジスタ(sr：Status Rèister)に，というようにだ。

また，特定用途用ではなく，汎用の(プログラムで自由に使える)レジスタもある。これら汎用レジスタは，プロセッサの外部に存在するメモリよりも，内部にある分，高速にアクセス(access：読み書き，と思っておけばいい)することができるという利点がある。

次ページの図2に68000の全レジスタを示そう。すでに説明したpc，srに加え，d0～d7，a0～a7の16本の32ビット長汎用レジスタが用意されている。このうち，d0～d7はデータレジスタと呼ばれ，主として〝たんなるデータ〟を格納するのに用いられる。また，a0～a7はアドレスレジスタと呼ばれ，おもにアドレスを格納するのに用いられる(とくに，a7レジスタは通常システムスタックポインタとして用いられ，アセンブリソースの中ではその略であるspと書くことが許されている)。このような汎用レジスタの性格分けの陰には68000の設計思想がある。つまり，データとアドレスを分離することで，プログラムの信頼性向上を狙っているのだ。データとアドレス(ポインタ)を混同して使うバグがどれほど致命的かは，他プロセッサやC言語を知っている人にはよくわかると思う。

ところで，68000のメモリ空間は16Mバイトであり，アドレスでいうと000000_H～$FFFFFF_H$の範囲になるから，24ビットに収まる。実際，68000の外部アドレスバス(アドレスを指定する信号線)は24ビット分(24本)しかない。ということは，本当はアドレスレジスタやプログラムカウンタは24ビット長で足りてしまうわけだ。それをあえて32ビット長にしてあるのには設計上の都合ということもあるだろうが，68000を作った時点ですでに68020以降の完全32ビットマイクロプロセッサのことが念頭にあったにちがいない。

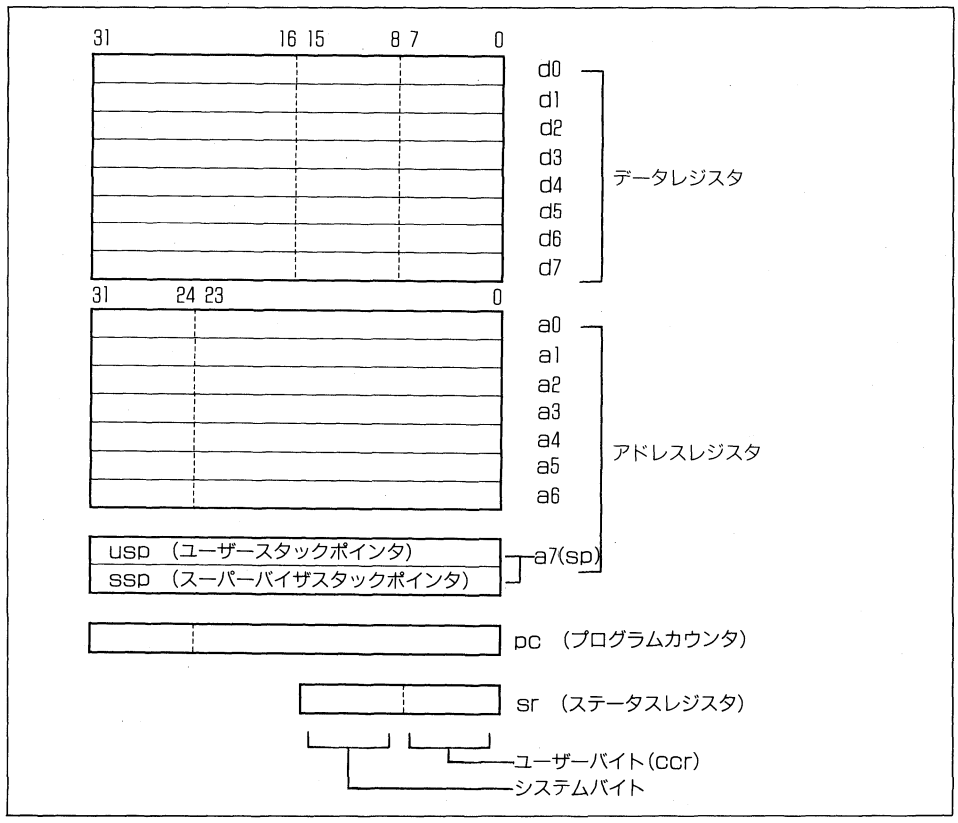


図1
68000のレジスタ

COLUMN

16ビットプロセッサとは

68000は16ビットのマイクロプロセッサである。モトローラもそういっている。何が16ビットなのかというと，外部データバス(プロセッサが外部とデータをやりとりする信号線)が16ビット分(16本)ある。8ビットプロセッサはデータバスが8本，32ビットプロセッサは32本というわけだ。

この信号線の本数が多ければ多いほど，一度にまとめてたくさんのデータをやりとりできることになり，マイクロプロセッサの能力(速度)を測る1つの目安になっている(ただし，〝16ビットは8ビットの2倍のバス本数だから，それだけで2倍速い〟といったかんたんな話ではない)。

ところが，これは定義と呼べるほど厳密なものではないらしい。たとえば，8086のデータバスだけを8ビットにした8088を，インテルは16ビットプロセッサだと称している。最近では32ビットプロセッサの80386の外部バスを16ビットにした80386SXとかいうプロセッサも32ビットプロセッサということになっている。この場合，プロセッサの内部バスの

本数を数えて何ビットプロセッサと呼んでいるようだ。つまり，プロセッサメーカーが16ビットプロセッサといえば，16ビットプロセッサなのである。

ちなみに，68000には外部データバスを8ビットにした68008という弟分がいて，モトローラはこれを8ビットプロセッサであるとしている(はず)。

さて，68000はレジスタが32ビット長であることからもわかるように，内部バスは32ビットだ。このあたりが〝68000は16ビットの仮面を被った32ビットプロセッサである〟といわれるゆえんだろう。もし，68000を作ったのがインテルだったら，きっと32ビットプロセッサといって売り出したにちがいない。もっとも，インテルの思想では68000は生まれないだろうけど。

アセンブリ言語の書式

そろそろ68000の命令に登場願いたいのだが，その前にちょっとだけ寄り道をして，ここで，僕たちがこれからマシン語プログラムの開発に用いるアセンブリ言語の文法というか書式についてかんたんにまとめておく。

アセンブリ言語のプログラムは，基本的に1行1命令の形式になっている。アセンブラによっては，マルチステートメントが許されているものもあるが，AS.Xを含むほとんどのアセンブラでは1行に複数の命令を書くことはできない。

各行は〝空行〟か〝コメント行〟か，そうでなければ何か意味のある命令などが置かれた行だ。空行は何もない行で，当然，何の意味も持たない。プログラムを読みやすくするために処理の切れ目に空行を挟んだりする。また，コメント行は注釈であり，これもアセンブル後のマシン語プログラムには影響しない。AS.Xでは，行頭に〝*〟を置くとその行はコメント行とみなされ，アセンブル時にはたんに読み飛ばされる。

空行とコメント行以外の意味のある行は，図1のようなフィールドに分けられる。ただし，必ずしもこれらのフィールドがすべて揃っている必要はない。

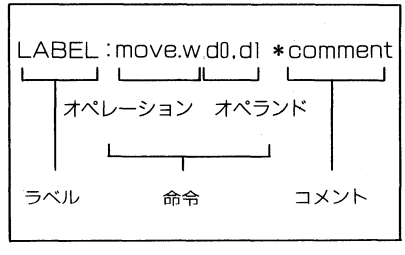


図2
1行のフォーマット

●ラベル

ラベルフィールドには，分岐命令の飛び先などの目印として使う任意の文字列をラベルとして書くことができる。ラベルは行頭から書きはじめ，最後に〝：〟を置く。なお，ラベルは1本のプログラムの中に同じものが複数あってはならない。

ここで，AS.Xはラベル中の英字の大文字・小文字を区別するので，

```
LABEL
Label
label
```

などはみんな別のラベルと見なされる。

●命令

命令のフィールドに実際の命令を書く。一般に，命令は行頭から（ないしはラベルの後ろから）いくつかのホワイトスペース（空白：スペースかタブ）を置き，その後から書きはじめる。AS.Xでは，このホワイトスペースがなくてもかまわないようだが，一般的ではないし，プログラムが読みにくくなるので，あまりすすめられない。

命令フィールドは，さらにオペレーション（operation）部とオペランド（operand）部に分けられる。オペレーション部は〝～する〟という動詞に相当する部分だ。オペレーションは命令の機能・意味を表す英単語を略した〝ニーモニック〟で書き表す。AS.Xでは，ニーモニックは大文字で書いても小文字で書いてもかまわない[1]。

■ 1) 本書ではニーモニックは一貫して小文字で表記する。

一方，〝～を〟とか〝～へ〟にあたるのがオペランドで，命令によってその数が決められている（68000では0～2個）。オペランドが複数ある場合は〝,〟で区切って記述する。また，オペランドを左から順に〝第1オペランド〟，〝第2オペランド〟と呼ぶことがある。さらに，〝～を〟に相当するオペランドを〝ソース（source：源の意味）オペランド〟，〝～へ〟にあたるものを〝デスティネーション（destination：行き先とか目的地の意味）オペランド〟という。68000のアセンブリ言語では，わずかな例外を除き，第1オペランドがソース，第2オペランドがデスティネーションとなる[2]。

■ 2) 68000のアセンブリ言語ではオペランドの順序がインテル/ザイログ系のプロセッサとは逆なので，8086, Z80などに慣れている人は注意すること。

●コメント

命令フィールド以降行末までがコメントフィールドだ。ここには任意の注釈を書くことができる。AS.Xでは命令フィールドの後ろは無条件にコメントとみなすようだが，〝*〟または〝;〟を1個置いてから注釈を書きはじめるのが一般的だ。

ところで，AS.Xではコメントなどに漢字を使うことができるお陰で，いいかげんな和製英語を使わなくてすむ。コメントの有無や質・量でプログラムの読みやすさがだいぶ違ってくるか

ら，コメントは過度なくらい書くようにしたい。とはいうものの，僕は日頃コメントをあまり書かない悪いプログラマなのだが。

12語でできる68000

読者諸氏がBASICを覚えたとき，まさかはじめからステートメントを全部暗記しようなんてことは考えなかったにちがいない。最初はFOR～NEXTとかPRINTなどの命令をほんの一握り覚え，後は必要に応じて命令の数を増やしていく，という段階を踏んだことだろう。また，各命令は覚えようとして覚えたわけではなく，使っているうちに〝覚えてしまった〟んじゃないだろうか。

マシン語の場合もやっぱり同じだと思う。最初に覚える命令はごく基本的なものにとどめ，とにかくそれらを組み合わせてプログラムを書いてみる。それが常道というものだ。

というわけで，ここでは数ある68000の命令の中から比較的使用頻度の高い12語を選んで紹介してみる。この12語を知っていれば，ある程度はプログラムが書けるはずだ。ただ，これには若干のトリックがないでもない。そのあたりのカラクリは本書を読み進むにつれて徐々に明らかになるだろう。

●move

マシン語にかぎらず，プログラムの基本の基本がデータ転送，いわゆる代入操作だ。

moveはそのデータ転送を行う命令で，アセンブリ言語の書式では，

```
move    転送元，転送先
```

のように書く。

●add, sub

add，sub[3]は，それぞれ加算，減算を行う命令だ。書式は，

```
add     B, A
sub     B, A
```

のようになり，それぞれ〝BをAに足し，結果をAに格納する〟，〝BをAから引き，結果をAに格納する〟という働きをする。

■ 3) sub：SUBtractの略。

BASIC風に書けば，

```
A＝A＋B
A＝A－B
```

と同じような動作だ。

ここで，

```
C＝A＋B
```

を実現するには,

```
move    A,C
add     B,C
```

のように2つの命令を組み合わせなければならない点に注意したい。マシン語には本当に単純な命令しかないということが実感できるだろう。

●and, or, eor

マシン語では，加減算などの算術演算と同じくらい頻繁にビット単位での論理演算が登場する。and, or, eor[4]はそれぞれ，論理積，論理和，排他的論理和をとる命令だ。書式は例によって,

```
and     B,A
or      B,A
eor     B,A
```

のようになり，それぞれ，AとBの間でビット単位の論理演算を行い，結果をAに格納する。

なお，eorはそれほどよく使う命令というわけではないのだが，関連命令をまとめて覚えてもらいたかったので，and, orといっしょに紹介した。

■ 4) eor：Exclusive ORの略。

●cmp

プログラムの中では，ある条件が成り立っているかどうかに応じて処理を振り分けるという場面がよくある。このような処理を行うのに必要な命令が，比較命令cmp[5]だ。

cmpは,

```
cmp     B,A
```

のように使い，AとBとを比較して結果(等しいか等しくないか。また，どちらが大きいかといった情報)を〝コンディションコード(condition code)〟に反映する。

cmp命令は単独で使っても意味がなく，次に示す条件分岐命令と組み合わせて使われる。

■ 5) cmp：CoMPareの略。

●bra, beq, bne

マシン語プログラムは通常，命令の並んだ順番に実行される。この処理の流れを変えるのが分岐命令bra[6]だ。BASICなどでいうGOTOにあたる。書式は,

```
bra     分岐先
```

のようになる。

braは無条件で分岐するのに対して，beqとbneはある条件が成立していれば分岐し，そうでなければ次の命令の実行に移るという条件分岐命令だ。

beq[7]は，直前の比較の結果が等しければ分岐し，等しくなければ分岐しない(そのまま次の命令の実行に移る)。bne[8]はその逆で，直前の比較の結果が等しくなければ分岐し，等しければ分岐しない[9]。

この過程でプロセッサは〝コンディションコード〟を調べて，直前の比較結果を知る。

■ 6) bra：BRanch Alwaysの略。つねに枝分かれするという意味。
■ 7) eq：EQualの略。
■ 8) ne：Not Equalの略。
■ 9) ここではbra，beq，bneを別々の命令として紹介したが，本来は1個の命令のバリエーションであり，68000の解説書では〝Bcc〟という1つの命令にまとめられている。この〝cc〟の部分に各種の条件が入る。条件としては，いま紹介したもの以外にも数多くあるが，残りは今後出てきたときに解説したい。

●bsr, rts

サブルーチンを実現するのがbsr[10]とrts[11]だ。それぞれ，BASICでいうGOSUB, RETURNにあたる働きをする。

```
bsr     分岐先アドレス
```

により，サブルーチンを呼び出し，サブルーチンの中の，

```
rts
```

により，呼び出し元に戻る。

■10) bsr：Branch to SubRoutineの略。
■11) rts：ReTurn from Subroutineの略。

以上で12語だ。それぞれの命令の働きは単純なものだから，動作を理解するのは容易だろう。

ところで，各命令は〝シーエムピー〟とか〝ビーエスアール〟などと棒読みにしないで〝コンペア〟，〝ブランチ・トゥ・サブルーチン〟のように，元の英語どおり(カタカナ読みだけど)に発音するようにしたい。これを，お経を唱えるように〝ビーエスアール，ビーエスアール〟などと読んでいると，いざというときに思い出せなくなって，〝マシン語はハナモゲラだから嫌い!〟とか〝そもそも私は日本人なんだー〟と言い訳しつつ挫折するのがオチだ。

アドレッシングモード

今度は，命令の後に続く部分(オペランド)に目を向ける。上の説明ではオペランドを〝転送元〟とか〝A〟などの言葉で表現したが，実際のプログラムでは，この部分に特定のメモリないしはレジスタなどが入る。

この〝オペランドの種類の指定方法・指定形式〟を「アドレッシングモード(addressing mode)」という。アドレッシングモードは，言葉どおりにとれば，〝アドレスを指定する方法〟のことだ。しかし，オペランドにメモリではなくレジスタや定数が現れるようなケースも〝アドレッシング〟の範疇に含むと考える。

高級言語にはアドレッシングモードに相当する概念はないか，もしあったとしても強く意識されることは少ない。たとえば，BASICのPRINT文では，

```
PRINT  1
PRINT  I
```

```
PRINT A(0)
PRINT INT(3.14*2*R)
```

といったぐあいに，命令の後ろに定数，変数，配列，関数などのいずれが来てもかまわないし，なんならもっと長く複雑な式でも書ける。そして，たんにポツンと定数を1個書く場合と長い式を書く場合とを区別する必要はないわけだ。

ところが，マシン語ではこのあたりの事情はかなり違う。基本的にはオペランドとなりうるのは，メモリの1アドレスか，1つのレジスタか，定数値だけだ。しかも，命令によっては，「オペランドはデータレジスタにかぎる」といった制限がつく場合もある(後でいいから，気が向いたら『アセンブラマニュアル』などの命令一覧を見てみるとよいだろう)。

もっとも，68000では同じアドレスを指定するのにも，

・直接数字で絶対アドレスを指定する。

・何かを基準にした相対的なアドレスで指定する。

・レジスタにアドレスを入れておき，そのレジスタ名で間接的にアドレスを指定する。

といったように(他にもまだある)，数通りの形式(アドレッシングモード)が用意されており，かなりプログラムは書きやすくなっている。

また，他のプロセッサと比べれば，命令の直交性(命令とアドレッシングモードの組み合わせの自由度)も高いほうで，「え？　この命令でこのアドレッシングモードが使えないの？!」と憤慨する場面も〝そう多くは〟ない。

さて，68000は細かく分ければ10数種類のアドレッシングモードを備えている。もちろん，全部知っていなければプログラムが書けないわけではない。比較的シンプルで，かつ，重要度の高いアドレッシングモードをさきに押さえ，後は必要に応じて手を広げていくというのが賢明というものだ。

以下，いくつかピックアップして紹介する。

レジスタ・直接(ダイレクト)・アドレッシング

データレジスタ直接形式と，アドレスレジスタ直接形式がある。長い名前がついているが，結局，レジスタの名前で直接指定する形のことだ。

たとえば，

```
move    d0,a0
```

のように書けば，d0レジスタの内容をa0レジスタにコピーするという意味で，

```
add     d1,d0
```

なら，d1の内容をd0に加えるという意味になる。

イミディエイト・アドレッシング

これも名前の長いわりには単純なもので，オペランドに定数(即値，イミディエイト値)をそのまま書く形式のことだ。

```
move    #2000,d0
```

のように書けば，d0に2000という値が入る。

この例でもわかるように，定数の前には〝#〟を置くのが68000のアセンブリ言語での決まりになっている。なお，アセンブリ言語上では，

```
move    #10＊20,d0
```

のような「定数式」を使ってもよい。この式は，アセンブル時にアセンブラが計算してくれる。

10進数以外にも，アセンブラは数の表現方法を何通りか許している。

前章のプログラムにもあったように，16進数は頭に〝$〟をつけて表し，

```
move    #$1234,d0
```

によって，d0には1234_H(10進数では4660)という値が入る。

また，AS.Xでは頭に〝%〟をつけると2進数と解釈され，

```
move    #%1011010,d0
```

ならば，d0に1011010_B(10進数の90)が入る。

ここで，10進数，16進数，2進数ともに読みやすくするために，任意の位置に〝_〟(アンダーバー)を挿入することが許されている。とくに2進数は桁が増えるとわけがわからなくなるので，

```
move    #%1010_0101_1100_0011,d0
```

のように，4ないし8桁ごとに〝_〟を入れるとよいだろう。

さらに，任意の文字を「″」または「'」のクオーテーションマークで囲むことによって，その文字の文字コードを表すこともできる。ここでいう文字コードとは，半角文字ならASCIIコード，漢字などの全角文字ではシフトJISコードだ。たとえば，

```
move    #'A',d0
move    #'全',d0
```

のように書けば，それぞれ，〝A〟のASCIIコードである41_H，〝全〟のシフトJISコードである9153_Hを意味するようになる。AS.Xでは複数の文字をクオーテションマークで囲む形も許され，

```
move    #'AB',d0
```

なら，d0には4142_Hという値が入る。

アブソリュート(絶対)・アドレッシング

これは，メモリの絶対アドレスを数字で直接指定する形式だ。

```
move    $80000,d0
```

なら，メモリの80000_H番地から格納されているデータをd0レジスタに取り出すという意味になる。また，

```
move    d0,$80000
```

なら，逆にd0レジスタの値を80000_H番地へ格納するという意味になる。

ここで，

```
move    #$1234,d0
```

と，

```
move    $1234,d0
```

の違いに注意したい。前者はイミディエイト・アドレッシングだから，〝1234_Hという値〟をd0に入れるという意味であり，後者はアブソリュート・アドレッシングだから，〝メモリの1234_H番地から格納されている値〟をd0に入れるという意味で，まったく働きが違う。

データのサイズ

アドレッシングモードの紹介をしている途中だが，このあたりで68000のマシン語でアドレッシングモードと並んで重要な概念である〝サイズ〟について話しておきたい。

68000のアセンブリリストを見てみると，moveなどの命令の直後に〝.b〟，〝.w〟，〝.l〟などのオマケがついていることがある[12]。これがサイズの指定だ。これは〝操作対象となるデータの長さ(何ビットか)〟を表している。高級言語における〝データ型〟のようなものと見ることもできるだろう。

■12) 〝.b〟，〝.w〟，〝.l〟は，それぞれ，バイト(Byte)，ワード(Word)，ロングワード(Long word)の略。

68000は，内部的には32ビットのプロセッサだ。これは32ビット単位で(まとめて)データを処理できることを意味する。が，いつも32ビット単位でデータを処理するとなると，もっと小さな(たとえば8ビットの)データを扱うときなどには(プロセッサの内部で)必要以上の手間がかかってしまうことになり，無駄が生じる。

そこで，68000では操作の対象となるデータをバイト(8ビット)，ワード(16ビット)，ロングワード(32ビット)の中から選んで指定することができるようになっているわけだ。

たとえば，

```
move.b  d0,d1
```

は〝d0レジスタの下位8ビットをd1レジスタの下位8ビットに転送する〟という意味になる。このとき，d1レジスタの全32ビットのうち上位24ビットは変化しない。ここで，d0＝01234567_H，d1＝$89ABCDEF_H$だとすると，命令の実行後のd1レジスタの値は$89ABCD67_H$になる。

同様に,

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

は〝d0レジスタの下位16ビットをd1レジスタの下位16ビットに転送する〟ことになり(さきの例だと，d1レジスタは89AB4567Hとなる),

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

なら〝d0レジスタの全ビット(32ビット)をd1レジスタにコピーする〟ことになる(d1はもちろん，01234567Hだ)。

ここで，サイズがワードのときは〝.w〟を省略することが許され,

```
move    d0,d1
```

は

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

と同じ意味になる。

COLUMN

データ長の単位

コンピュータにおける，もっとも小さなデータの単位は〝ビット(bit)〟と呼ばれ，1ビットは2進数1桁に相当する(bitはBinary digITの略)。つまり，1ビットでは〝0か1か〟しか表現することができない。

ビットをある決まった数だけまとめたものを〝バイト(byte)〟という。68000を含む多くのマイクロプロセッサでは8ビット＝1バイトと定義されている(本書でも，この定義にしたがう)。バイトはメモリ量を表すときにおなじみの単位だろう。一般に，メモリは1バイトごとに区切られ，それぞれに通し番号である〝アドレス(address)〟が振られている。

1バイトでは,

00000000_B〜11111111_B

までの2進8桁の数を表現することができる(ここで，2進数で表したときの各桁を右から順に，第0ビット，第1ビット，……，というように数える。とくに，第0ビットのことを「最下位ビット」，また，いちばん左端の桁を「最上位ビット」と呼ぶ)。16進数で考えると,

00_H〜FF_H

であり，1バイトではちょうど16進数2桁の数を表現できることがわかる。これは，10進数に直せば,

0〜255

になる。

さらに，68000では16ビット＝2バイトをまとめて「ワード(word)」，32ビット＝4バイト＝2ワードを「ロングワード(long word)」と呼ぶ。ワードの下(右)半分を「下位バイト」，上(左)半分を「上位バイト」，さらに，ロングワードの下半分を「下位ワード」，上半分を「上位ワード」と呼んだりもする。

ワードというのは〝そのマイクロプロセッサにとって扱いやすい(と設計者が考えた)データの長さ〟のことであり，コンピュータによって1ワードが何ビットを意味するかはまちまちだが，68000系，8086系のマイクロプロセッサを使う分には1ワード＝16ビットと覚えていても差し支えはないだろう。

なお，16ビットでは，
0000_H～$FFFF_H$　（16進）
0～65535　（10進）
の数を，32ビットでは
00000000_H～$FFFFFFFF_H$（16進）
0～4294967296　（10進）
の範囲の数を表現できる。

サイズとメモリ上のデータ

サイズの考え方はレジスタどうしであればわかりやすいが，メモリ上のデータを扱うときには少しばかり注意が必要だ。

かりに，80000_H番地から次のようにデータが並んでいるとする。

80000_H　12_H
80001_H　34_H
80002_H　56_H
80003_H　78_H

また，d0レジスタには00000000_Hが入っているものとしよう。

ここで，

```
move.b  $80000,d0
```

を実行すると，〝80000_H番地に格納された1バイトのデータをd0レジスタの下位8ビットにコピーする〟わけだから，d0の内容は00000012_Hになるのは明らかだ。では，

```
move.w  $80000,d0
```

のようにメモリからワードデータを取り出す場合にはどうなるだろうか。

68000ではこのような場合は，80000_H番地のデータを上位バイト，80001_H番地を下位バイトとするようなワードデータがd0に転送される。いまの例ではd0は00001234_Hになる。

さらに，

```
move.l  $80000,d0
```

のようにサイズがロングワードの場合は，指定アドレスからの連続する4バイト（この場合，80000_H～80003_H番地）のデータがd0に取り出される（その結果，d0は12345678_H）。

メモリへデータを書き込むときも同じようなぐあいで，d0が12345678_Hのとき，

```
move.b  d0,$80000
```

によって，80000_Hにはd0の下位8ビットである78_Hが書き込まれ，80001_H番地以降は変化しない。

以下，

```
move.l  d0,(a0)
```

などとやろうものなら，その〝どこだかわからないアドレス〟にデータが書き込まれることになり，プログラムが誤動作することにもなりかねない。

もっとも，68000ではプロセッサ自身がある程度のチェックをすることで，変なアドレスにデータが書き込まれるのを防ぐ機能を持っている。運よくそのチェックに引っかかれば，例によって68000は〝怒って〟，プログラムの実行を停止する。

ポストインクリメント・アドレスレジスタ・インダイレクト(間接)・アドレッシング

アドレスレジスタ間接アドレッシングのバリエーションだ。アセンブリ言語では，アドレスレジスタ名をカッコでくくり，その後ろに〝+〟を1つ書くことで表す。

```
move.b    (a0)+,d0
```

このアドレッシングモードの動作は，なかなかおもしろい。まず，アドレスレジスタのポイントするメモリとの間でデータのやりとりや演算を行い(ここまではアドレスレジスタ間接アドレッシングと同じ)，〝その後で(post)〟指定したアドレスレジスタの値が自動的に〝増えて(increment)〟，次に続くデータをポイントするように変更される。

上の例でa0が最初80000_Hだったとすると，80000_H番地の1バイトデータがd0の下位バイトに転送された後で，a0は自動的に80001_Hになる。一命令で，

```
move.b    (a0),d0
add.l     #1,a0
```

に相当する処理をいっきに行えるわけだ。

ここで，アドレスレジスタは〝次のデータをポイントするように変更される〟というのがミソで，

```
move.w    (a0)+,d0
```

のようにサイズがワードの場合はワードデータの長さ分，つまり2増え，

```
move.l    (a0)+,d0
```

のようにロングワードの場合は4増える。

ポストインクリメント・アドレスレジスタ間接アドレッシングは，配列状に並んだデータに対して次々に処理を行う場合などに便利に使える。データを1個処理し終わったときには，アドレスレジスタはもう次のデータを指していてくれるのだ。

プリデクリメント・アドレスレジスタ・インダイレクト(間接)・アドレッシング

これは，ポストインクリメント・アドレスレジスタ間接アドレッシングと対になるものだ。

アドレスレジスタ名をカッコでくくり，その直前に〝−〟(マイナス)をつけることで表す。

```
move.b    d0,−(a0)
move.l    −(a0),d0
```

このアドレッシングモードでは，命令を実行する〝前に(pre)〟指定のアドレスレジスタの値を〝減じ(decrement)〟，その後，アドレスレジスタ間接アドレッシング同様の動作を行う。

a0が80000_Hをポイントしているときに，

```
move.b    −(a0),d0
```

を実行すると，a0レジスタは$7FFFF_H$になり，d0にはその時点でアドレスレジスタによってポイントされているアドレス(いまの場合，$7FFFF_H$番地)のデータが取り込まれる。これは，

```
sub.l     #1,a0
move.b    (a0),d0
```

の2命令分に相当する動作だ。また，a0＝80000_Hのときに，

```
move.l    −(a0),  d0
```

を実行すれば，a0は4減じられて$7FFFC_H$となり，d0には$7FFFC_H$からの4バイトデータが入ることになる。

プリデクリメント・アドレスレジスタ間接アドレッシングは，ポストインクリメント・アドレスレジスタ間接アドレッシングと組みにして，〝スタック〟を形成するのに使われることが多い。具体的に，どうやってスタックを実現するかについてはコラムを参照されたい。

COLUMN

スタック(2)

スタックはたんなるデータ構造の一種だが，コンピュータにとっては，それだけではすまない重要な意味を持つ。そのため，ふつうマイクロプロセッサにはスタックを操作する命令が用意されている。一介のデータ構造をハードウェアでサポートしているのだ。

スタックがこれほどまでに重要なのは，サブルーチンの呼び出しをするのにスタックがあると非常に便利だからだ。この話は別項に譲るとして，ここではスタックのもう1つの用途である，データの一時的な待避場所という見方から，68000でスタックを構成する手法について説明する。

まず最初に，68000にはスタック操作専用の命令は用意されていないことを断っておく。68000でのスタック操作はアドレッシングモードをうまく活用することで行う。スタック操作に使うアドレッシングモードは，プリデクリメント・アドレスレジスタ間接形式，ポストインクリメント・アドレスレジスタ間接形式だ。これらのアドレッシングモードをmove命令に適用すれば，プッシュ動作，ポップ動作を行うことができる。

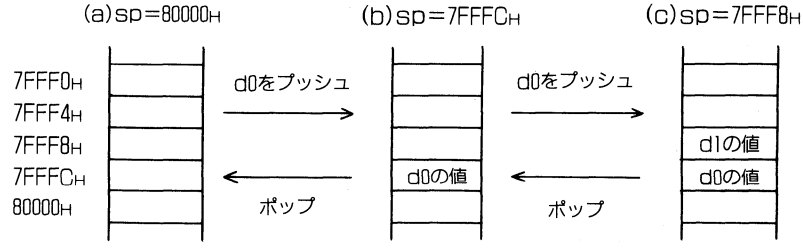
以下は，前章の「本」にたとえたスタックのイメージを思い出しながら読んでもらいたい。

図Aの(a)のように，最初スタックポインタ(sp＝a7レジスタ)が80000_H番地を指しているものとする。ここで，

```
move.l  d0,−(sp)
```

によって，プリデクリメント・アドレスレジスタ間接アドレッシングを適用し，d0レジス

図A スタック



タの内容を転送すると，spは7FFFC$_H$番地を指すようになり，d0の値は7FFFC$_H$番地以降に格納される(b)。

ここでさらに，

```
move.l  d1,－(sp)
```

を実行すると，(c)のようになり，spは7FFF8$_H$になる。スタックにはd0, d1レジスタの内容がプッシュされたことになる。つまり，プリデクリメント・アドレスレジスタ間接アドレッシングを使えば，データ転送命令であるmove命令を使って，スタックへのプッシュを行うことができる。

この状態から，

```
move.l (sp)＋,d1
```

を実行すれば，(b)の状態に戻り，d1にはスタックからデータが取り出される。

さらに，

```
move.l (sp)＋,d0
```

により，d0レジスタにデータがポップされ，(a)と同じ最初の状態に戻る。結局，ポストインクリメント・アドレスレジスタ間接アドレッシングにより，スタックからのポップ動作が実現される。

また，

```
move.l (sp),d0
```

という命令は，スタックトップを覗き見るという働きをすることがわかると思う。

ところで，ポストインクリメント，プリデクリメントのアドレッシングモードは，なにもmove命令にしか使えないわけではないから，

```
add.l (sp)＋,d0
```

のように，スタックから取り出したデータを直接演算に使うといった芸当もできる。スタック専用の命令を用意せず，アドレッシングモードに含めた68000の設計は，ソフト屋好きのする美しい設計といえるだろう。

COLUMN スタック(3)

プロセッサにとって，サブルーチンの呼び出しでやっかいなのは〝サブルーチンが終わったらメインルーチンに帰ってこなければならない〟という問題だ。サブルーチンの呼び出しが1段階だけなら，専用のレジスタでも用意して，戻り先のアドレスをしまっておけばすむところだが，現実にはサブルーチンは何重にも重ねて呼び出される。

そこで，スタックの登場である。

スタックは最後に入れたものが最初に出てくるし，スタックポインタを１つ用意するだけで管理できる。サブルーチンの呼び出しにはうってつけのデータ構造だ。

スタックがあれば，サブルーチンの呼び出しは〝スタックに戻りアドレスを押し込んで(プッシュして)から，サブルーチンの先頭へ分岐する〟ことで行え，サブルーチンからのリターンは，〝スタックトップの値(最後にスタックに積んだデータ)をアドレスとみなして，そこへ分岐する〟ことで実現できる。

さて，68000は，プログラムカウンタの指すメモリからデータを命令を読み込み，プログラムカウンタが次の命令を指すように更新してから命令の実行に移る(そうなのだ)。ということは，サブルーチンコールの際にスタックに積む戻りアドレスは，その時点でのプログラムカウンタの値にほかならない。プロセッサ内部で，サブルーチンからの戻りアドレスをいちいち計算する必要もないわけだ。

サンプルプログラム

本章のまとめとして，最後にかんたんなサンプルプログラムを示す。

リスト１は，〝A〟から〝Z〟までの文字を順に表示するプログラムだ。文字の表示には，前章と同様にDOSコールputcharを使っている。

リスト1
A_Z,S

```
 1: *       'A'～'Z'を表示するプログラム
 2:
 3: _EXIT           equ     $ff00   *DOSコールexitのシンボル定義
 4: _PUTCHAR        equ     $ff02   *DOSコールputcharのシンボル定義
 5:
 6:         .text
 7:         .even
 8: *
 9: start:
10:         move.w  #'A',d1         *表示を始める文字コードをd1へ
11:
12: loop:   move.w  d1,-(sp)        *スタックに文字コードを積み
13:         .dc.w   _PUTCHAR        *DOSコールputcharを呼び出す
14:         add.l   #2,sp           *スタックポインタを補正する
15:
16:         add.w   #1,d1           *d1=次に表示する文字コード
17:         cmp.w   #'Z'+1,d1       *d1は'Z'+1と等しいか？
18:                                 *（'Z'をすぎたか？）
19:         bne     loop            *そうでなければ処理を繰り返す
20:
21:         bsr     crlf            *改行サブルーチンを呼び出す
22:
23:         .dc.w   _EXIT           *実行終了
24:
25: *       改行処理サブルーチン
26: *
27: crlf:
28:         move.w  #$0d,-(sp)      *CRコードを
29:         .dc.w   _PUTCHAR        *  出力
```

```
30:             add.l   #2,sp           *スタック補正
31:
32:             move.w  #$0a,-(sp)      *LFコードを
33:             .dc.w   _PUTCHAR        *  出力
34:             add.l   #2,sp           *スタック補正
35:
36:             rts                     *メインルーチンへ戻る
37: *
38:             .end
```

プログラムは単純なループ構造をしている。最初にd1レジスタに〝A〟のASCIIコードを入れておき(10行)，DOSコールを使って1文字表示する(12～14行)。1文字表示したら，d1に1を足す(16行)。これでd1は次に表示すべき文字の文字コードになる。この時点で，d1を〝Z〟の文字コード+1と比較し(17行)，等しくなければ，〝loop〟というラベルのついた行に分岐し(19行)，処理を繰り返す。等しければ，26文字分の表示が終わったことを意味するからループを抜け，改行して，DOSコールexitによりプログラムを終了する(23行)[13]。

■13) DOSコールへのパラメータの引き渡し方も，本章のスタックに関する知識があれば，move.w d1,－(sp)によって，スタックに1ワードのデータが積まれ，DOSコール実行後にスタックに積まれたままになっているデータを捨てる意味で，さきほど積んだデータサイズ(2バイト)を足し，スタックポインタを補正していることがわかるだろう。

改行の処理は，bsrでサブルーチン〝crlf〟を呼び出すことで行っている(21行)。サブルーチンの中では，おなじみのコントロールコード2つ(CRとLF)をputcharしているだけだ。

なお，〝beq〟，〝bsr〟の分岐先は〝ラベル〟で指定している。見てもらえばわかるように，行頭に適当な文字列＋〝:〟を書くことで，その文字列はラベルとして定義され，置かれた行のアドレスを意味する定数として使えるようになる。

後，プログラムの冒頭にある

```
.text
.even
```

という2行と，最後の

```
.end
```

だが，これらはアセンブラに対する指令(疑似命令)だ。〝.end〟は文字どおり〝これでプログラムが終わりますよ〟の印であり，〝.text〟，〝.even〟は〝ここからプログラムが始まりますよ〟という〝おまじない〟のようなものだと思ってもらってかまわない。

この他，リスト中ではDOSコールを呼び出す部分で，〝.dc.w〟という命令を使っている。これも疑似命令の一種で〝続くワードデータをそのまま(数字として)オブジェクトプログラムの中に埋め込みなさい〟という意味だ。しかし，いまはDOSコールを呼び出す際の決められた形だと思っていればいいだろう。

以上，68000のマシン語の基礎について話してきた。これだけ身につければ，とりあえずマシン語プログラマの仲間入りをしたといってしまってもいいだろう(ちょっといいすぎだな)。後

はプログラムを書くときの考え方やコツを身につけることだ。そのためには，どんどんプログラムを読み，また，書くのがいちばん。

次章以降では，その〝コツ〟をつかんでもらうために多くのプログラム例を示していくことになる。

CHAPTER 12

12語の68000実習プログラム

CHAPTER 3

12語の68000実習プログラム

ASSEMBLER

前章では，68000の基本的な12語の命令といくつかのアドレッシングモードを紹介した。「これはこうだから，覚えてね」攻撃に終始してしまったきらいはあるが，僕の気持ちを察してくれた人なら，試しにプログラムを何本か書いてみたんじゃないかと期待している。あれだけの命令と文字の入出力程度だけでも，かなりいろいろなプログラムを作れるものだ。

本当は，もう少し命令とアドレッシングモードの種類を増やさないと68000らしさが見えてこないのだが，12語でできるといってしまった手前もあって，この章では新しい命令を登場させずに前章で登場した命令だけを使って短いプログラムをいくつか作ってみる。

"Y"と"N"のキー入力を判別する

例題の常として，実用的でもなければ，おもしろくもないプログラムだが，最初の例題はこんなところでどうだろう。

> DOSコールgetcを使ってキーボードから大文字の〝Y〟か〝N〟が入力されるのを待ち，その文字をDOSコールputcharにより表示するプログラム「YN.X」[1)]

■ 1) getcを使って入力しているのだから，正確には「標準入力から～」というべきだが，ここではリダイレクションは行われない前提で標準入力＝キーボードとして話を進めている。

すでにDOSコールを使って１文字入出力する方法は知っているはずだ。入力された文字が〝Y〟や〝N〟かどうかはcmpで比較してみればわかる。そして，比較後の処理の振り分けはbeqを使えばよい。

「〝Y〟か〝N〟が押されるのを待つ」というのは，入力された文字が〝Y〟または〝N〟でなければ，ふたたび１文字入力に戻るようにすればいいのである。

結局，プログラムは，ループがあって，その中でキー入力を行い，入力された文字が〝Y〟か〝N〟かどうかを調べて，一致したらループから飛び出すという形になる(リスト１：YN.S)。

リスト1
YN.S

```
 1: *        キー入力に応じてYかNを表示する
 2:
 3: _EXIT            equ     $ff00
 4: _PUTCHAR         equ     $ff02
 5: _GETC            equ     $ff08
 6:
 7:          .text
 8:          .even
 9: *
10: start:
11: loop:    .dc.w   _GETC           *1文字入力
12:          cmp.b   #'Y',d0         *'Y'か?
13:          beq     yes             *そうなら'Y'の表示処理へ
14:          cmp.b   #'N',d0         *'N'か?
15:          beq     no              *そうなら'N'の表示処理へ
16:          bra     loop            *どちらでもなければやり直し
17: *
18: yes:     move.w  #'Y',-(sp)      *'Y'を表示
19:          .dc.w   _PUTCHAR
20:          add.l   #2,sp
21:          .dc.w   _EXIT           *終了
22: *
23: no:      move.w  #'N',-(sp)      *'N'を表示
24:          .dc.w   _PUTCHAR
25:          add.l   #2,sp
26:          .dc.w   _EXIT           *終了
27: *
28:          .end
```

さて，リスト1では，入力された文字が〝Y〟であればラベル「YES」で示される行へ，〝N〟であれば「NO」の行へ分岐し，それぞれの文字を表示して実行を終えるようになっている。しかし，よく考えてみると，ループを抜けた時点でd0レジスタには表示すべき文字が入っているわけだから，これを利用すればプログラムはもっとシンプルになる(リスト2)。

リスト2

```
 1: *        キー入力に応じてYかNを表示する　2
 2:
 3: _EXIT            equ     $ff00
 4: _PUTCHAR         equ     $ff02
 5: _GETC            equ     $ff08
 6:
 7:          .text
 8:          .even
 9: *
10: start:
11: loop:    .dc.w   _GETC           *1文字入力
12:          cmp.b   #'Y',d0         *'Y'か?
13:          beq     print           *そうなら表示処理へ
14:          cmp.b   #'N',d0         *'N'か?
15:          bne     loop            *そうでなければやり直し
```

```
16: *
17: print:  move.w  d0,-(sp)        *'Y'か'N'を表示
18:         .dc.w   _PUTCHAR
19:         add.l   #2,sp
20:         .dc.w   _EXIT           *終了
21: *
22:         .end
```

また，リスト2は15行あたりにも小さな変更が加えられている。この変更から「条件が成り立ったら～する」のと，「条件が成り立たなかったら～しない」ことが同じだということをつかんでもらいたい。

このYN.Xの応用として，リスト3に示す「ASKYN.X」を作ってみた。このプログラムは，キーボードから〝Y〟か〝N〟が押されるのを待って，〝Y〟が押されたら0，〝N〟が押されたら1を終了コードとして実行を終えるようになっている。

リスト3
ASKYN.X

```
 1: *       YかNのキー入力に対して
 2: *               Y ・・・ 0
 3: *               N ・・・ 1
 4: *                       の終了コードを返す
 5:
 6: _EXIT           equ     $ff00
 7: _GETC           equ     $ff08
 8: _EXIT2          equ     $ff4c
 9:
10:         .text
11:         .even
12: *
13: start:
14: loop:   .dc.w   _GETC           *1文字入力
15:         and.w   #%1101_1111,d0  *英小文字→大文字変換
16:         cmp.b   #'Y',d0         *'Y'か？
17:         beq     yes             *そうならyesの処理へ
18:         cmp.b   #'N',d0         *'N'か？
19:         bne     loop            *どちらでもなければやり直し
20: *
21: no:     move.w  #1,-(sp)        *終了コード1を返す
22:         .dc.w   _EXIT2
23: *
24: yes:    .dc.w   _EXIT           *終了コード0を返す
25: *
26:         .end
```

COLUMN

プログラムの終了コード

終了コードというのは，プログラムから親プロセス(一般にCOMMAND.X)へ返す戻り値のようなものといえる。この値を調べることで，プログラムが正常に終了したのか，エラーがあって中断したのかといった終了状況を知ることができる。ただし，プログラムがそのような終了コードを返すように作られていれば，の話だが。

もちろん，終了コードはエラーの有無を返したりするだけではなく，もっと他の，意味のある値を返すようにすることもできる。本文のASKYN.Xは，終了コードを積極的に利用したプログラムの一例だ。

終了コードを持ってプログラムの実行を終えるには，DOSコールexitのかわりにexit2を使う。このDOSコールは，

```
move.w  終了コード,－(sp)
.dc.w   _EXIT2
```

のようにして利用する。見てのとおり，終了コードはワードでスタックに積むが，一般には上位バイトは0にして0～255の値だけを使う。このあたりはCOMMAND.Xとの兼ね合いだ。

また，表には出てこないが，DOSコールexitは無条件に終了コード0を返す。つまり，exitは，

```
move.w  #0,－(sp)
.dc.w   _EXIT2
```

と同じ働きをする。

Human68k上のプログラムでは，正常終了したときは0，エラー終了したときは0以外の値(通常は1)の終了コードを返すのが慣例になっているので，今後はそれにしたがうことにしよう。

プログラムが返した終了コードは，COMMAND.XのIFの中でERRORLEVELまたはEXITCODEによって参照することができる。詳細は『Human68kユーザーズマニュアル』を参照してもらうことにして，リストAにかんたんな例を示しておく。

リストA　ASM.BAT

```
 1: echo off
 2:
 3: :LOOP
 4: as %1
 5: if errorlevel 1 goto ERROR
 6:
 7: lk %1
 8: if errorlevel 1 goto ERROR
 9:
10: del %1.o/y
11: goto END
12:
13: :ERROR
14: echo エラーが発生しました
15:
16: :END
```

このバッチファイルASM.BATは，

```
A>ASM ファイル名
```

の形で使い(ただし，ファイル名に拡張子はつけないこと)，指定したファイルをアセンブル，リンクするものだ。アセンブル時，リンク時にエラーが発生すると，メッセージを出してバッチ処理を中断するが，ここでAS.X，LK.Xの終了コードを利用している。

このバッチファイルを拡張すれば，アセンブルエラーが起きたら自動的にエディタが立

ち上がるようにもできるだろう。

15行のand命令がちょっと引っかかるかもしれないが，これはもっともかんたんな(手抜きかつ不完全な)英小文字→大文字変換処理になっている。これは，入力が〝y〟や〝n〟でも対応できるようにするための処置だ[2]。

■ 2) ASCIIコードの英大文字と小文字のコードを2進数で表してみると，第5ビットが0か1かというだけの違いしかないことがわかる。そこで，第5ビットを0にすれば，大文字は大文字のまま，小文字はしっかり大文字に変換されることになる。and演算の性質上，あるビットだけが0で他が1である数とandをとれば，元のデータの他のビットは変化させずに指定ビットをリセットする(0にする)ことができる。逆に，任意のビットだけをセットする(1にする)には，そのビットだけが1であるような数とorをとればよい。ちなみに，このような操作を「ビットマスク」とか，たんに「マスクする」という。

このASKYN.Xは，かんたんなものながら，作っておいて損のないプログラムだ。こんなちっぽけなプログラムがあるだけで，対話的なバッチプログラムを書くことができる。バッチの中でキー入力ができるというわけだ。きっと多くの人が似たようなプログラムをシステムディスクに忍ばせていると思うのだが，どうだろう。

ASKYN.Xを使ったかんたんな例「KILL.BAT」をリスト4に示しておく。このバッチは，DELコマンドの安全性を高めたもので，

A＞kill ファイル名

のようにして使うと，まず指定したファイルの一覧をDIRコマンドで表示し，本当に消去してもよいかを確認してからファイルを消去する(ファイル名には，当然ワイルドカードが使える)。ASKYN.Xの終了コードによって，DELを実行するかどうかを判断しているわけだ[3]。

■ 3) KILL.BATを使ううえでは，DIRコマンドとDELコマンドのワイルドカードの扱いの違いに気をつけなければならない。

A＞KILL TEST

を実行すると，バッチ内部では，

DIR TEST
DEL TEST

に展開されるわけだが，前者が，

DIR TEST.*

と同じ意味になるのに対して，DELで実際に消去されるのは拡張子がついていない〝TEST〟というファイルただ1つになる。

リスト4
KILL.BAT

```
1: echo off
2: dir %1/w
3: echo 以上のファイルを消去します
4: echo よろしいですか [Y/N]
5: askyn
6: if errorlevel 1 goto END
7: del %1/y
```

```
 8: :END
```

さて，ここで読者には，〝Y〟の代用としてスペースやリターンキー，〝N〟の代用としてESCキーも認めるようにASKYN.Xを拡張してみることをすすめる。かんたんすぎて頭の体操にもならないかもしれないが，運がよければ最初のバグ退治を経験できるかもしれない。完成したと思ったら，リスト5のTEST.BATでいろいろなキーを押して動作を確認してみてほしい。とくに，スペースの入力が正しく受けつけられるかどうかと，テンキーを適当に押してどういう結果が出るかを確認してみてほしい。

リスト5
TEST.BAT

```
 1: echo off
 2:
 3: :LOOP
 4: askyn
 5: if errorlevel 512 goto BREAK
 6: if errorlevel 1 goto NO
 7:
 8: :YES
 9: echo 終了コードは0です
10: goto LOOP
11:
12: :NO
13: echo 終了コードは1（以上）です
14: goto LOOP
15:
16: :BREAK
17: echo BREAKしました
```

運悪くバグが出なかった人は，このようにプログラムを拡張することの意味を考えてみるのもいいかもしれない。ユーザーインタフェイスというのは，なにもビジュアルなものだけを指すのではないということだ(ちょっと説教調だな)。

全角英大文字を表示する

つづいて，全角文字の表示を行ってみる。

```
1行に全角の〝A〟～〝Z〟を表示し，改行する
プログラム「ZEN_A_Z.X」
```

表示する文字が全角でもputcharを使うことには変わりはない。ただ，全角文字は2バイトのシフトJISコードで表されることを思い出す必要がある。

ここで，putcharは表示すべき文字コードをワードでスタックに積むので，

```
move.w   #'A', -(sp)     ＊Aは全角
.dc.w    _PUTCHAR
add.l    #2, sp
```

のようにすれば表示できるように思える。

ところが，実際に試してみるとわかるように，うまく表示されない。putcharはスタックに積まれたデータの下位バイトしか見ないのだ(つまり，上位バイトは無視される)。

こういうわけで，全角文字を表示するには，まずシフトJISコードの上位バイト，つづけて下位バイトというように，2度に分けてputcharを呼び出す必要がある。

さて，全角英大文字はシフトJISコード8260_H～8279_Hに割りつけられている。上位バイトは82_Hのまま固定だから，残る下位バイトにのみ集中すればよい。後は前章のリスト1の要領でループを組めば完成だ。結局，プログラムはリスト6のようになった。

リスト6
ZEN_A_Z.S

```
 1: *        全角の'A'～'Z'を表示する
 2:
 3: _EXIT            equ     $ff00
 4: _PUTCHAR         equ     $ff02
 5:
 6:          .text
 7:          .even
 8: *
 9: start:
10:          move.w  #$0060,d1       *シフトJISコード'A'の下位バイト
11:
12: loop:    move.w  #$0082,-(sp)    *上位バイトを
13:          .dc.w   _PUTCHAR        *  出力
14:          move.w  d1,(sp)         *下位バイトを
15:          .dc.w   _PUTCHAR        *  出力
16:          add.l   #2,sp           *スタックポインタ補正
17:
18:          add.b   #1,d1           *次の文字
19:          cmp.b   #$7a,d1         *最後まで表示したか？
20:          bne     loop            *そうでなければ繰り返す
21:
22:          bsr     crlf            *改行
23:
24:          .dc.w   _EXIT           *終了
25: *
26: crlf:
27:          move.w  #$0d,-(sp)      *CRコードを
28:          .dc.w   _PUTCHAR        *  出力
29:          move.w  #$0a,(sp)       *LFコードを
30:          .dc.w   _PUTCHAR        *  出力
31:          add.l   #2,sp           *スタック補正
32:          rts                     *リターン
33: *
34:          .end
```

このリストでチェックしておいてもらいたいのは，12～16行のputcharの連続呼び出しだ。通常は表示する文字をスタックに積んでからputcharをコールし，その後で積まれたままにな

っているスタック上のデータを捨てる意味でスタックポインタを補正するわけだが，いまのように2度続けて呼び出す場合には，「スタックに積んだデータを捨てて，新しいデータを積む」かわりに，「スタックトップのデータを直接新しいデータで置き換える」ことで同じ結果が期待でき，かつ，処理ステップを節約できることにもなる。

ただ，こういうことはやりすぎると，プログラムが読みにくくなる(＝修正がしにくくなる)危険があるので，あまりすすめられたことではない。いまの例では，表示が2バイト単位であることをはっきりさせる効果はあるだろうが。

文字に色をつける

今度は，地味な文字表示に多少なりとも華やかさを添える意味で文字色を変えてみる。ZEN_A_Z.Xを拡張して，次のようなプログラムを作ってみよう。

> 全角文字の〝A〟～〝Z〟を1文字ごとに色を変えながら表示するプログラム「COL_A_Z.X」

基本的には，リスト6のループの中で，文字の表示に加えて色の設定処理を挿入するだけだ。

文字色の設定は，DOSコール$FF23のconctrlを使って行う。このDOSコールは，いくつかの機能が複合していて，引数でどの機能を使うか指定するようになっている。文字色の設定は，そのモードの2にあたり，具体的な呼び出し方は次のようになる。

```
move.w   文字属性,－(sp)
move.w   #2,－(sp)        ＊モード2
.dc.w    _CONCTRL
add.l    #4,sp
```

文字の属性というのは，文字色や強調，反転などの指定をひっくるめたもので，X68000では0～15の値で指定する。それぞれの値の意味は，X-BASICのCOLOR(n)のnと同じ意味を持っている。

とりあえず作ってみたのがリスト7だ。ついでなので，ただ文字の色を変えるだけではなく，文字属性を0～15まで順に増やすことで，強調や反転と組み合わせて変化するようにしてみた。また，手を抜いて，最後の改行処理は省いてある。

リスト7
COL_A_Z.S

```
1: *        全角の'A'～'Z'を色（文字の属性）を変えながら表示する
2:
3: _EXIT            equ     $ff00
4: _PUTCHAR         equ     $ff02
5: _CONCTRL         equ     $ff23
6:
```

```
 7:          .text
 8:          .even
 9: *
10: start:
11:          move.w   #$0060,d1          *シフトJISコード'A'の下位バイト
12:          move.w   #1,d2              *文字属性=1（青ノーマル）
13:
14: loop:    move.w   d2,-(sp)           *d2=文字属性
15:          move.w   #2,-(sp)           *conctrlモード2
16:          .dc.w    _CONCTRL           *つぎに表示する文字の属性を設定
17:          add.l    #4,sp              *スタックポインタ補正
18:
19:          move.w   #$0082,-(sp)       *上位バイトを
20:          .dc.w    _PUTCHAR           *  出力
21:          move.w   d1,(sp)            *下位バイトを
22:          .dc.w    _PUTCHAR           *  出力
23:          add.l    #2,sp              *スタックポインタ補正
24:
25:          add.b    #1,d2              *次の文字属性（変な表現？）
26:          and.b    #$0f,d2            *d2=d2%16
27:
28:          add.b    #1,d1              *次の文字
29:          cmp.b    #$7a,d1            *最後まで表示したか？
30:          bne      loop               *そうでなければ繰り返す
31:
32:          .dc.w    _EXIT              *終了
33: *
34:          .end
```

12行が文字属性を保持するレジスタの初期設定だ。1から始めているのは，文字属性0が黒だということがわかっているからだ。

14～17行で文字表示に先立って文字属性を設定する。上で示したとおりの手順で行っているのがわかるだろう。

文字を表示した後で，次に備えて文字属性を1増加させる(25行)。ここでは26行のand命令の使い方がポイントだ。このandにはd2レジスタに入れておいた文字属性が15を超えないようにする働きがある。下位4ビットを残して，上位ビットが0になるようにマスクすることで，ちょうど16で割った余りを求めるのと同じ効果がある。納得できない人は，いろいろな数と$0F_H$(=15)とのandをとって確かめてみるとよいだろう。

さて，このプログラムを実行してみると，いくつかの穴が発見される。第1に，プログラムが終了した時点で文字属性が変更されたままになっていること，第2に，ところどころ文字が表示されていないことなどだ。そこで，このあたりを改良してみることにする。

文字属性の再設定

実行後の文字属性が変更されたままになってしまうという症状は，プログラムの最後で標準

の文字属性に再設定するだけで改善される。

ところどころ文字が表示されていないのは，文字属性が4や8などの4の倍数のときだ。考えてみれば，文字属性が4の倍数のときには，「黒の強調」とか「黒の反転」とかなわけで，見えないのは当然である。

そこで，文字属性を1→2→3→5→6→7→9……のように，4の倍数(0倍も含む)をスキップするような処理を付け加えることにする。

どうやって実現するかは，安直にcmpで比較し，0だったら強制的に1に，4だったら5にというぐあいに修正しまくるというのも1つの手だろう。しかし，せっかく4の倍数という共通点があるのだから，これをうまく活用したい。

4の倍数かどうかを調べるには，4で割ってみて余りが0かどうかで判断できる。さきほど16の余りを求めたときのように，2進数で表したとき，下2桁だけが1であるような数＝3でandをとれば，4の余りを求めたことになる。

さっそくプログラムにしてみると，リスト8のようになった。25行以下が文字属性が4の倍数かどうかを調べる処理だ。文字属性はd2レジスタに格納してあるわけだが，単純にd2と3のandをとってしまうとd2の値が変わってしまうので，一度d2を破壊してもよい(＝プログラムの他の部分で使っていない)d3レジスタにコピーし，d3と3のandをとるようにしてある。

リスト8
COL_A_Z.S (2)

```
 1: *       全角の'A'～'Z'を色（文字の属性）を変えながら表示する　2
 2:
 3: _EXIT           equ     $ff00
 4: _PUTCHAR        equ     $ff02
 5: _CONCTRL        equ     $ff23
 6:
 7:         .text
 8:         .even
 9: *
10: start:
11:         move.w  #$0060,d1       *シフトJISコード'A'の下位バイト
12:         move.w  #1,d2           *文字属性=1（青ノーマル）
13:
14: loop:   move.w  d2,-(sp)        *d2=文字属性
15:         move.w  #2,-(sp)        *conctrlモード2
16:         .dc.w   _CONCTRL        *つぎに表示する文字の属性を設定
17:         add.l   #4,sp           *スタックポインタ補正
18:
19:         move.w  #$0082,-(sp)    *上位バイトを
20:         .dc.w   _PUTCHAR        *　出力
21:         move.w  d1,(sp)         *下位バイトを
22:         .dc.w   _PUTCHAR        *　出力
23:         add.l   #2,sp           *スタックポインタ補正
24:
25:         add.b   #1,d2           *次の文字属性
26:         move.b  d2,d3           *一旦d2をd3にコピーして
27:         and.b   #$03,d3         *文字属性の下位2ビットは
28:         cmp.b   #0,d3           *　0か？
29:         bne     skip            *そうでなければそのまま
30:         add.b   #1,d2           *黒は飛ばす
```

```
31: skip:   and.b   #$0f,d2         *d2=d2%16
32:
33:         add.b   #1,d1           *次の文字
34:         cmp.b   #$7a,d1         *最後まで表示したか？
35:         bne     loop            *そうでなければ繰り返す
36:
37:         move.w  #3,-(sp)        *文字属性=3（白ノーマル）
38:         move.w  #2,-(sp)        *conctrlモード2
39:         .dc.w   _CONCTRL        *文字属性を標準の状態に戻す
40:         add.l   #4,sp           *スタックポインタ補正
41:
42:         .dc.w   _EXIT           *終了
43: *
44:         .end
```

3とのandをとった後，結果が0かどうかを調べ，もしそうであれば4の倍数だったことになるから，1を足す。

そして，37行以下が文字属性を標準の状態である3(白ノーマル)に戻している部分だ。もう，何もいうことはない。

とりあえず本章での課題はこれで終わりにする。しかし，読者諸氏にはさらにいくつかのプログラムを作ってみてもらいたいと思っている。最後のCOL_A_Z.Xだって，文字属性を逆順に変化させるようにするとか，奇数の文字属性だけを使うとか，2文字ごとに色を変えるようにするといったバリエーションが考えられるはずだ。

その他にも，ASCIIコードや50音の表を表示するとか，画面を消去する(スペースで埋める)プログラムのバリエーション(うずまき状に消していくとか)をいろいろ考えてみるのもおもしろい。自分の頭と体を使う人にはそれなりの発見が約束されるはずだ。

明日のために　その1

最後に，1つだけ話をしておこう。かりに読者諸氏がサンプルリストを見る前に自分で試しにプログラムを書いてみたり，自己流の例題を作って解いたり，少なくともサンプルを打ち込んで実際に動かすことくらいはしてくれていると仮定しよう(そう願っているんだけど)。いくつかプログラムを作ってみればすぐわかることだが，DOSコールの番号をマニュアルで調べるのは意外に面倒な作業だということに気づくだろう。

僕たちは，すでにequ疑似命令でDOSコール番号をラベルに定義するという技を知っている。ラベルを使えば，プログラムの中に$ff02なんて数字を埋め込まなくても，_PUTCHARという(比較的)わかりやすい表現で記述できるのである。それでも，ラベルは値を定義してやらなければ使えないわけだから，一度はDOSコール番号を調べる必要があることに変わりはない。ただ，こういう作業をプログラムを作るたびに行うのはうれしくない話なので，次なる技

を伝授しておこう。

話はかんたんだ。あらかじめすべてのDOSコール番号を定義したリスト9のようなファイルを1つ作っておく。先頭の数行が意味深だが，後はequがずっと並んだだけのファイルだ。ファイル名は〝DOSCALL.MAC〟としておこう。

リスト9
DOSCALL.MAC
(C compiler
PRO-68Kより
引用)

```
 1:         .nlist
 2: *
 3: * doscall.mac X68k XC Compiler v2.00 Copyright 1990 SHARP/Hudson
 4: *
 5: DOS      macro   callname
 6:          dc.w    callname
 7:          endm
 8: *
 9: * 注意
10: * RESERVED になっているファンクションコールの処理を変更したり、呼び出したり
11: * してはいけません。
12: * $fff0～$fff2までのファンクション番号は、終了、ＣＴＲＬ＋Ｃ、アボートの
13: * 処理を_INTVCSによって変更する場合に使用するもので、ファンクションコール
14: * ではありません。
15: * また、$fff3～$ffffまでのファンクションコールは、処理を変更できません。
16: *
17: _EXIT           equ     $ff00
18: _GETCHAR        equ     $ff01
19: _PUTCHAR        equ     $ff02
20: _COMINP         equ     $ff03
21: _COMOUT         equ     $ff04
22: _PRNOUT         equ     $ff05
23: _INPOUT         equ     $ff06
24: _INKEY          equ     $ff07
25: _GETC           equ     $ff08
26: _PRINT          equ     $ff09
27: _GETS           equ     $ff0a
28: _KEYSNS         equ     $ff0b
29: _KFLUSH         equ     $ff0c
30: _FFLUSH         equ     $ff0d
31: _CHGDRV         equ     $ff0e
32: _CHDRV          equ     $ff0e
33: _DRVCTRL        equ     $ff0f
34: _CONSNS         equ     $ff10
35: _PRNSNS         equ     $ff11
36: _CINSNS         equ     $ff12
37: _COUTSNS        equ     $ff13
38: *RESERVED               $ff14
39: *RESERVED               $ff15
40: *RESERVED               $ff16
41: _FATCHK         equ     $ff17
42: *RESERVED               $ff18
43: _CURDRV         equ     $ff19
44: _GETSS          equ     $ff1a
45: _FGETC          equ     $ff1b
46: _FGETS          equ     $ff1c
47: _FPUTC          equ     $ff1d
48: _FPUTS          equ     $ff1e
```

```
 49: _ALLCLOSE      equ     $ff1f
 50: _SUPER         equ     $ff20
 51: _FNCKEY        equ     $ff21
 52: _KNJCTRL       equ     $ff22
 53: _CONCTRL       equ     $ff23
 54: _KEYCTRL       equ     $ff24
 55: _INTVCS        equ     $ff25
 56: _PSPSET        equ     $ff26
 57: _GETTIM2       equ     $ff27
 58: _SETTIM2       equ     $ff28
 59: _NAMESTS       equ     $ff29
 60: _GETDATE       equ     $ff2a
 61: _SETDATE       equ     $ff2b
 62: _GETTIME       equ     $ff2c
 63: _SETTIME       equ     $ff2d
 64: _VERIFY        equ     $ff2e
 65: _DUP0          equ     $ff2f
 66: _VERNUM        equ     $ff30
 67: _KEEPPR        equ     $ff31
 68: _GETDPB        equ     $ff32
 69: _BREAKCK       equ     $ff33
 70: _DRVXCHG       equ     $ff34
 71: _INTVCG        equ     $ff35
 72: _DSKFRE        equ     $ff36
 73: _NAMECK        equ     $ff37
 74: _MKDIR         equ     $ff39
 75: _RMDIR         equ     $ff3a
 76: _CHDIR         equ     $ff3b
 77: _CREATE        equ     $ff3c
 78: _OPEN          equ     $ff3d
 79: _CLOSE         equ     $ff3e
 80: _READ          equ     $ff3f
 81: _WRITE         equ     $ff40
 82: _DELETE        equ     $ff41
 83: _SEEK          equ     $ff42
 84: _CHMOD         equ     $ff43
 85: _IOCTRL        equ     $ff44
 86: _DUP           equ     $ff45
 87: _DUP2          equ     $ff46
 88: _CURDIR        equ     $ff47
 89: _MALLOC        equ     $ff48
 90: _MFREE         equ     $ff49
 91: _SETBLOCK      equ     $ff4a
 92: _EXEC          equ     $ff4b
 93: _EXIT2         equ     $ff4c
 94: _WAIT          equ     $ff4d
 95: _FILES         equ     $ff4e
 96: _NFILES        equ     $ff4f
 97: _SETPDB        equ     $ff50
 98: _GETPDB        equ     $ff51
 99: _SETENV        equ     $ff52
100: _GETENV        equ     $ff53
101: _VERIFYG       equ     $ff54
102: _COMMON        equ     $ff55
103: _RENAME        equ     $ff56
104: _FILEDATE      equ     $ff57
105: _MALLOC2       equ     $ff58
```

```
106: _MAKETMP        equ     $ff5a
107: _NEWFILE        equ     $ff5b
108: _LOCK           equ     $ff5c
109: *RESERVED               $ff5e
110: _ASSIGN         equ     $ff5f
111: _S_MALLOC       equ     $ff7d
112: _S_MFREE        equ     $ff7e
113: _S_PROCESS      equ     $ff7f
114: _EXITVC         equ     $fff0
115: _CTRLVC         equ     $fff1
116: _ERRJVC         equ     $fff2
117: _DISKRED        equ     $fff3
118: _DISKWRT        equ     $fff4
119: _INDOSFLG       equ     $fff5
120: _SUPER_JSR      equ     $fff6
121: _BUS_ERR        equ     $fff7
122: _OPEN_PR        equ     $fff8
123: _KILL_PR        equ     $fff9
124: _GET_PR         equ     $fffa
125: _SUSPEND        equ     $fffb
126: _SLEEP_PR       equ     $fffc
127: _SEND_PR        equ     $fffd
128: _TIME_PR        equ     $fffe
129: _CHANGE_PR      equ     $ffff
130:         .list
```

そして，プログラムのソース側ではラベル定義を行うかわりに，

```
.include    doscall.mac
```

という1行を入れるようにする(includeはインクルードと読む)。こうしておくと，アセンブラが勝手にDOSCALL.MACをこの位置に読み込んでアセンブルしてくれるという寸法だ。このとき，DOSCALL.MACは必ずカレントディレクトリに置いておくこと。

リスト9の5～7行はマクロの定義というやつで，詳しい話はまたの機会にするが，この定義によってDOSコールの呼び出しが，

```
.dc.w   _PUTCHAR
```

のような形ではなく，

```
DOS     _PUTCHAR
```

というように，いかにも「DOSコールを呼び出しています」という体裁で記述できるようになる。

ということで，次章からはこのDOSCALL.MACが読者の手元にもあるという前提で，DOSコールの定義はリスト中には含めずに，また，DOSコールの呼び出しもマクロを使った形で書き表すことにしたい。

なお，C compiler PRO-68Kのシステムディスクには，このDOSCALL.MACがあらかじめ用意されているので，わざわざ新しく作らなくてもこれをそのまま流用することができる。困ったことに，当然こっちにも含まれているべきTHE福袋V2.0にはDOSCALL.MACは入っていない(この件については，僕はちょっと怒っているのだ)。申し訳ないが，これも試練と諦

めて，明日のために打ち込むべし。

COLUMN

ブレイクチェック

マシン語に対する恐怖を教える伝説の１つに「BREAKキーが効かない」というやつがある。

「BASICならプログラムの実行中，いつでもBREAKキーで止めることができるが，マシン語は……」と，おどろおどろしく語られたりもする。

この程度の話を聞いたくらいで弱気にならずに，「それならBREAKキーが押されたら実行を停止するようなプログラムを作ろう」と考えるのが，プログラマたるものの感覚だ。具体的には，処理の合間ごとにキーの入力状況を調べて，特定のキーが押されていたら実行を中止するように細工すればよい。

ただ，実際にこういうチェックをこまめにするのはかなり煩雑な仕事になる。DOSの設計者もそのあたりを配慮してくれたらしく，Human68kやMS-DOSのDOSコールには，DOSコール実行中にBREAKキーが押されたら，プログラムを終了させる機能があらかじめ組み込まれている。

試しにリスト１のYN.Xを走らせ，〝Y〟や〝N〟のキーを押すかわりに ^Cを入力してみよう。

```
^C
```

の表示が出て，プログラムが途中で終了することが確かめられるだろう[1]。

■ 1) ^Cは，CTRLキーを押しながら〝C〟のキーを押すことで入力されるコードで，ASCIIコード03H。BREAKキーでも入力できる。

ついでにブレイクチェック関係の話をもう少ししておこう。Human68kのオペレーションがらみの話も混じるが，Human68kの上で動くプログラムを作る以上，いちおうは覚えておかなければならない事柄だ。

CONFIG.SYS内で，

```
BREAK=ON
```

を指定した状態(COMMAND.X上でA>BREAK ONを実行しても同じ。以下同様)では，すべてのDOSコールでブレイクチェックが行われるが，

```
BREAK=OFF
```

では，DOSコール$FF01～$FF05，および，$FF08～$FF0Eでのみブレイクチェックが行われる。

なお，Human68kのver2.0では，

```
BREAK=KILL
```

の設定を行うことで，どのDOSコールでもブレイクチェックを行わないようにすることができるようになった。

しかし，Human68kのバージョンにかかわらず，一般には，

```
BREAK=OFF
```

に設定しておくのが自然だろう。今後，この本でも，このような設定になっているものとして話を進めていくつもりでいる。

なお，あまり使うことはないと思うが，DOSコール$FF33のbreakckを使えば，プロ

グラム内部でブレイクチェックの度合いを調べたり，設定したりすることもできる。

それから一部のDOSコールには，^C以外にも^Sで表示を一時停止する機能を持つものがある。『プログラマーズマニュアル』のputcharの解説を参照してもらいたい。

CHAPTER 4

デバッガを使ってみよう

CHAPTER 4

デバッガを使ってみよう

ASSEMBLER

この章では少し趣向を変えてデバッガを使ってみる。Human68k上のデバッガとしてはDB.Xがあり，これはC compiler PRO-68KかTHE福袋V2.0を購入するとついてくる。DB.Xを持っている人は，実際に動かしながら読んでみてほしい。持っていない人は図を見てその気になってもらいたい。

デバッガの諸機能

デバッガはその名のとおり，デバッグ時に使用するツールだ。といっても，実行するだけで自動的にバグを退治してくれる魔法のプログラムというわけではない(当たり前だって)。立ち上げても地味なタイトルを表示してプロンプトを出すだけだ。この状態からデバッガに用意されたさまざまなコマンドを駆使してバグを追いつめていく，これがデバッガの使い方である。

マシン語デバッガ(以下，たんにデバッガ)が最低限持っている機能としては，

1) メモリ内容を表示する(ダンプ)
2) メモリ内容を書き換える(メモリエディット)
3) メモリ上のマシン語プログラムをアセンブリ言語に逆変換して表示する(逆アセンブル)
4) レジスタの内容をみる・書き換える(レジスタダンプ・エディット)
5) プログラムを任意の位置から実行する
6) プログラムの実行を任意の位置で止める(ブレイクポイントの設定)

といったものがある。他には，

7) ファイルのロード・セーブ
8) ディスクの直接読み書き

などもサポートされているだろう。もう少しお洒落なデバッガになると，さらに，

9) プログラムを一命令ごとに実行する(シングルステップ実行，トレース)
10) 1行単位でのアセンブラ機能(ワンラインアセンブラ)

などが用意されているし，その他の付加機能があるかもしれない。なお，DB.Xは，ここで挙げた機能すべてを包含している。

これらのコマンドを使ってプログラムの動きを観察し，じわじわとバグの核心に迫っていくわけだ。デバッガを使ったマシン語プログラムのデバッグは，ちょうどBASICプログラムを途中で止めて変数の内容を調べたりするのと同じような作業になる(考えようによっては「その程度」ともいえるが)。

さて，デバッガは本来のマシン語プログラムのデバッグ以外にも使い道がある。いくつか挙げてみよう。

1) Cなどのコンパイラ言語で生成したプログラムのデバッグ

コンパイラが生成するプログラムも，結局はマシン語プログラムである。というわけで，Cなどで作ったプログラムのデバッグにもマシン語デバッガが使える。しかし，あくまでマシン語レベルでのデバッグであり，Cなどのソースリストとは頭の中で関連づけなければならない。コンパイラがどのようなコードを出すかわかっていないと，デバッグどころではなくなってしまう。

もっとも，世の中にはソースコードデバッガとかソースレベルデバッガとか呼ばれるものもあって，これは高級言語のソースを参照しながらデバッグできるという便利なものだ。

2) 解析

システムプログラムなど，他人の作ったプログラムの動作を調べるのにもデバッガは有効だ。いろいろなテクニックや技術的な情報を得ることもできるだろう。僕なんかの場合，新しいプログラムを手に入れると，使う前についついデバッガで中を覗いてしまう。こうなるともう病気である。

3) ファイルの編集

ファイルを読み込み，内容やファイルサイズを変えてセーブし直すことでファイルの編集が行える。テキストファイルであれば通常のエディタで編集できるが，バイナリファイルともなると，しっかり1バイトずつ修正できるデバッガのほうが有利だ。X68000なら，ADPCMのデータファイルから特定の部分だけを抜き出すといった用途も考えられる[1]。

■ 1) 文字コードと若干のコントロールコードだけから構成されているのがテキストファイルで，その他の実行形式ファイルなどがバイナリファイル。もっとかんたんにいうならば，TYPEコマンドで内容が読めるのがテキストファイルで，意味不明の文字をまき散らすのがバイナリファイルである。

COLUMN

実行可能ファイル

拡張子が〝.S〟のアセンブリソースファイルをアセンブルすると，〝.O〟のオブジェクトファイルが生成され，さらにリンクすることで実行可能ファイルである〝.X〟のファイルができあがる。

これまでは，〝.O〟はマシン語プログラムになる一歩手前の形で，〝.X〟がマシン語プログラムそのものであるような書き方をしてきたが，実際にはちょっと違う。正しくは，Xファイルは〝メモリに読み込まれた時点でマシン語プログラムとなるような形式のファイル〟

というべきだ。

ここでXファイルの起動メカニズムについてかんたんにふれておこう。

XファイルはHuman68kによってメモリに読み込まれ，その後，実行される。どこに(どのアドレスに)読み込むかはHuman68kが〝メモリの空いているところを探して〟決める。メモリの管理はOSの基本的な機能の1つだ。

メモリに読み込むときもベタ読みするのではなく，〝そのアドレスで実行できるように〟手を加える必要がある。具体的には，プログラム中で絶対アドレスを参照している部分だ。

たとえば，

```
        move.l    d0,work
        DOS       _EXIT
work:   .ds.l     1
```

というようなプログラムがあったとする。このプログラムを80000$_H$番地から読み込んだ場合は，workというラベルで指定されたアドレスは8XXXX$_H$にならなければならないし，90000$_H$番地から読み込まれたときには9XXXX$_H$にならなければならない。

このように読み込むアドレスに応じてプログラムの一部を変更する必要があり，Xファイルはそのための情報を持っているわけだ。

4) ディスクエディタとしての使用

ディスクの直接読み書きがサポートされていれば，かんたんなディスクエディットが行える。知識があれば，壊れてしまったディスクの修復や，あやまって消去してしまったファイルの復活なども可能だ。

5) マシン語の学習

デバッガはマシン語プログラムの動作を確認するためのツールだ。これはマシン語入門者にとっても有効な道具となる。機能のよくわからない命令もデバッガ上で実際に動かし，レジスタの変化などを調べれば，「こういう動作をする命令だ」ということが一目でわかる。また，OS上でのプログラム開発では気づかないような，プロセッサの細かな動作を知ることもできる。

というぐあいに，なにかと便利なのがこのデバッガだ。ここで，デバッガを引っ張り出してきたのは，もちろん，マシン語のお勉強のためである。

DB.Xの起動と終了

とりあえずDB.Xを立ち上げてみよう。コマンドラインから次のコマンドを投入すると，図1-aのような起動メッセージが表示される。DB.Xのプロンプトは〝−〟だ。

```
A>DB
```

DB.Xはなかなか多機能なので，すべてのコマンドを覚えるのはたいへんだが，Hコマンドにより，かんたんなヘルプが出る(図1-b)。このヘルプ画面を参照すれば，たいていの操作はマ

ニュアル(DB.Xの解説は『アセンブラマニュアル』にある)を見なくても行える。

デバッガを終了するのはQコマンドだ。

```
-Q
```

によりDB.Xからコマンドモードに戻る(図1-c)。

ａ）起動

```
A>db
X68k Debugger v1.01 copyright 1987 SHARP/Hudson
-
```

ｂ）ヘルプ画面

```
-h
A                    :アセンブル              Q                    :終了
B                    :ブレークポイントの表示  R filename           :ファイルの読み
込み
B[bp] [address]      :ブレークポイントの設定  S[size]<range> data   :メモリのサーチ
BC[<bp>]             :ブレークポイントの消去  T[=address] [count]   :トレース
BE[<bp>]             :ブレークポイント有効    U[=address] [count]   :表示無しトレー
ス
BD[<bp>]             :ブレークポイント無効    V                    :コンソール切り
替え
BR                   :ブレークカウント初期化  W filename,<range>    :ファイルの書き
込み
C string             :コマンドラインの設定    X                    :レジスタの表示
D[size] [<range>]          :メモリのダンプ    X[reg]               :レジスタの変更
E[size] [address]          :メモリの編集      Y/N                  :ポーズ
F[size] <range> data       :フィルメモリ      Z[num=exp]           :システム変数の
表示
G[=address] [adress]       :実行              ?[exp]               :１６進表示
H                          :ヘルプ表示        ??[exp]              :１０進表示
HI                         :ヒストリ表示      ¥                    :コマンドライン
の繰り返し
L[<range>]                 :逆アセンブル
M <range> address          :メモリブロックの移動
P                          :ステータスの表示
PS                         :シンボルの表示

operators  + - * / & (and) | (or) ! (not) % (residue)    (exor)
number     ??(hex.) .??(symbol) ¥??(dec.) _??(bin.)
size       s(byte) w(word) l(long)
-
```

ｃ）終了

```
-q
A>
```

図1
DB.Xの起動/ヘルプ
/終了画面

次に，かんたんなプログラムを作ってその動きをデバッガで確かめてみることにしよう。リスト1の〝TEST.X〟を題材にする[2]。見てのとおり，d0.bとd1.bに適当な数を入れておいて，add命令で足し合わせるというだけのプログラムだ[3]。

■ 2) 正確には，「リスト1の〝TEST.S〟をアセンブル・リンクしてできる〝TEST.X〟」と書くべきだが，毎回こう書くのも面倒なので，手を抜いて略した書き方をしてある。混乱しないように願う。
■ 3) d0.bと書いたら〝d0レジスタの下位8ビットの意味。同様にd0.wなら〝d0レジスタの下位16ビット〟で，d0.lなら〝d0レジスタの全32ビット〟の意味。

リスト1
TEST.S

```
 1:          .include        doscall.mac
 2: *
 3:          .text
 4:          .even
 5: *
 6: start:
 7:          move.b  #10,d0          *d0 = 10
 8:          move.b  #20,d1          *d1 = 20
 9:          add.b   d1,d0           *d0 = d0 + d1
10:
11:          DOS     _EXIT          *終了
12:
13:          .end
```

では，次のようにしてデバッガを起動しよう。

```
A>DB TEST.X
```

図2-aのようなメッセージが表示され，TEST.Xがデバッガとともにメモリ上に読み込まれる。

図で〝PC=～〟以後が，68000の全レジスタの現在の値を表している。この値はデバイスドライバの組み込み状況などによって違ってくるから，図の値は参考程度に見てもらいたい。

左上から順に，プログラムカウンタ(PC)，ユーザースタックポインタ(USP)，スーパーバイザスタックポインタ(SSP)，ステータスレジスタ(SR)が並んでいる。SRの右の〝X:～〟以降は，SRの下位バイトであるCCRの状態を1ビットごとに表したものだ。これについては後述する。

その下の2行は，おなじみのデータレジスタとアドレスレジスタの内容だ。左から順にd0～d7，a0～a7の順に並んでいる。データレジスタは〝デバッガによって〟0で初期化されているのがわかる。また，アドレスレジスタは〝ターゲットプログラム[4]が(デバッガを使わずに)直接起動されたときの状態〟になるように，やはりデバッガによって初期化されている。Human68k上のプログラムは，起動時にOSからコマンドライン引数など，いくつかの情報を渡されることになっている。詳しい話は次章以降でしよう。

■ 4) ターゲットプログラム=デバッグの対象となるプログラム(例の場合はTEST.X)のこと。

a）プログラムのロード

```
A>db test.x
X68k Debugger v1.01 copyright 1987 SHARP/Hudson
loading test.x
No symbol file
PC=000A1BA0 USP=000A1104 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
move.b  #$0A,D0
-
```

b）プログラムのロードアドレスの確認

```
-p
debug program from $0009AA50
user  program from $000A1BA0
              end  $000A1BAC
              exec $000A1BA0
-
```

c）逆アセンブルリストの表示

```
-l
  000A1BA0      move.b  #$0A,D0
  000A1BA4      move.b  #$14,D1
  000A1BA8      add.b   D1,D0
  000A1BAA      _EXIT
  000A1BAC      move.b  (A0)+,D0
  000A1BAE      bsr.w   $000A2114
  000A1BB2      addq.l  #1,D2
  000A1BB4      bra.s   $000A1B60
-
```

d）プログラム本体をダンプ

```
-d a1ba0 a1bab
000A1BA0  103C 000A 123C 0014 D001 FF00                    .<...<..ﾐ...
-
```

e）実行

```
-g
program terminated normally
```

図2
リスト1の動きを
確かめる

レジスタの値の下には，現在PCが指しているアドレスに置かれた命令が逆アセンブルされ，表示されている。すぐ実行できるように，PCにはプログラムの実行開始アドレスがセットされているから，リスト1の最初の命令が表示されることになる。

スーパーバイザモードとユーザーモード

前ページの図2-aでチェックポイントとして挙げられるのがa7レジスタの値である。

a7レジスタは，スタックポインタとして暗黙のうちに使われることになっているのは覚えているだろう。前章までのプログラムの中でspと書かれていたものはa7レジスタを意味するのだった。

また，2章のレジスタ一覧図を見てもらいたい。a7レジスタは，USPとSSPという2つのレジスタをまとめたものとして表されている。これは，a7レジスタが場合によって2つの値を切り替えて使うということを意味している。つまり，a7レジスタはたんなる窓口で，実体はUSPかSSPのどちらかになる。

ここからはほんの少し混み入った話になるから，気合いを入れて〝読み飛ばす〟ように。

MPU68000には2つの走行モードがある。1つはユーザーモード，もう1つがスーパーバイザモードだ。モードがユーザーモードのときにはa7にUSPが割り当てられ，スーパーバイザモードになるとSSPに切り替わる。Human68k上のアプリケーションプログラムは通常ユーザーモードで走るので，スタックポインタとしてはUSPが使われている。図2-aでUSPの値とa7の値が等しいことを確認してもらいたい。この2種類のモードは，スタックポインタの切り替えに影響を与えるだけではなく，もっと本質的な意味がある。メモリの保護によるシステムの信頼性の向上という考え方である。

デカコン，ミニコンクラスのコンピュータは，基本的に1台のマシンを複数の人間が使うことになる。このようなマルチユーザーのシステムでは，誰かがプログラムを暴走させても，他のユーザーやシステム(OS)には被害が出ないように構築されていなければならない。

具体的には，個々のユーザーのプログラムが決められた範囲のメモリにしかアクセスできないようにする(もし，所定の範囲外のメモリにアクセス〝しようとしたら〟，システム側が強制的にそれをやめさせる)などの方法が考えられる。つまり，メモリを保護するわけだ。

この結果，自分のメモリ領域にあるプログラムやデータが他のプログラムによって破壊される(不当に書き換えられる)心配がなくなる。さらには，OS自体もユーザープログラムの暴挙から守られることになり，システムの信頼性も高くなる。

インテルの8086系が電卓から徐々に高機能化してきたのと対照的に，ミニコンクラスの機能をマイクロプロセッサで実現しようという発想で設計された(であろう)68000は(完全な形ではないにしろ)メモリの保護機能を備えている。ここで，さきほどの2つの走行モードが絡んでくる。モードごとに別々の〝メモリ空間〟が割り当てられているのだ。

スーパーバイザ状態では，プログラムはスーパーバイザ空間上で走り，ユーザー状態ではユーザー空間上で走る。名前から見当がつくように，スーパーバイザモードのほうが上位の(偉い!)モードでいろいろな特権を持っていたりもする。そこで，一般にOSなどのシステムをスーパーバイザ空間に置き，ユーザープログラムはユーザー空間に置く前提でシステム(ソフトウェ

ア・ハードウェアとも)が設計される。OSはユーザープログラムと〝別の空間〟にあるので,ちょっとやそっとでは破壊されることがないわけだ。

さて,実際にメモリ空間をどう割り当てるかは68000マシンを設計する段階で適当に決められるが,X68000の場合はメインメモリの000000H番地からHuman68k本体やデバイスドライバが置かれた範囲までがスーパーバイザ空間(スーパーバイザモードでなければアクセスできない)で,それ以降,メインメモリ12Mバイトがユーザー空間(両モードともにアクセスできる),そのまた後ろのVRAMやIPLやIOCSコール用のROMがある範囲がスーパーバイザ空間だ。もちろん,Human68k上の一般的なアプリケーションはシステムの管理下,ユーザーモードで走行する。

もう一度要点を確認しておくと,

1) 68000にはスーパーバイザモードとユーザーモードという2つのモードがあり,それぞれ別の〝空間〟を持つ。

2) OSなどはスーパーバイザ空間に置かれ,アプリケーションはユーザー空間に置かれる。OSはアプリケーションとは〝別の〟空間にいるので,プログラムが暴走したりしてもOSは安全である。

3)だから,ばりばりプログラムを作って走らせましょう。

というわけだ[5]。

■ 5)ただ,このメモリ保護は100%安全というわけではない。アプリケーションはすべてユーザー空間に置かれるから,チャイルドプロセスで複数のプログラムを起動するような場合(COMMAND.XやVS.Xから実行する場合も含む)には,〝子プロセスの悪さ(=バグ)〟によって,あやまって親プロセスが破壊される危険性がある。最悪の場合,プログラムを実行し,COMMAND.Xに帰ってきた時点でCOMMAND.Xが壊れていて暴走するケースだって考えられる。まぁ,たまには暴走を経験できたほうが緊張感があるということか。

スタックポインタの指すアドレス

図2-aのa7レジスタの値にはもう1つ見るべき点がある。現在のスタックポインタがどこを指しているかに注目してもらいたい。

ここでPコマンドを実行してみると,図2-bのような情報が得られる。DB.X自体がメモリ上のどこにいるか,また,ターゲットプログラムが何番地から読み込まれたかが表示されている(例によって,図の値は僕のシステム構成の場合だから,読者のシステム構成の場合は違う値が表示されるはずだ)。

このアドレスとa7の値を比べてみると,スタックポインタはDB.Xの内部を指していることがわかるだろう。この状態でTEST.Xを実行すると,このDB.X内に用意されたスタック領域をそのまま使うことになる(実際にはTEST.Xはスタックを使用しないが)。

68000でスタックにデータをプッシュするということは，〝スタックポインタにプリデクリメント・アドレスレジスタ間接アドレッシングを適用し，データ転送すること〟，つまり〝スタックポインタの値をプッシュするデータサイズ分だけ減らしてから，そのスタックポインタによりポイントされるメモリへデータを書き込むこと〟だった。つづけてデータをプッシュすれば，スタックポインタはさらに小さなアドレスを指すように更新されていく。

スタックに山ほどデータを積めば，それだけ小さいアドレスのほうまでスタックが伸びていき，現在のSPがポイントするアドレスから最初SPが指していたアドレスまで，びっしりとプッシュされたデータで埋めつくされることになる。

では，スタックにはどれだけの量のデータが積めるのだろうか。いいかえると，プログラムに最初に与えられるスタックの大きさはどのくらいなのだろうか？

じつは，これはわからないのだ。

デバッガからではなく，COMMAND.Xから直接起動した場合も同じことで，スタックポインタの再設定をしないかぎり，COMMAND.Xのスタックをそのまま使うことになる。そのままたくさんのデータをスタックに積むと，スタックがどんどん伸びていって，COMMAND.X本体を浸食するかもしれない。いつ溢れるかわからないスタックを使うのはロシアンルーレットみたいなもので，非常に危険である。

そこで，安全を期するには〝自分の内部にスタック領域を確保し，起動時にスタックポインタがその領域のいちばん後ろを指すようにセットする〟必要がある。こうすれば，どこまでスタックとして使っていいかがわかるし，それ以前に〝必要なだけ〟スタック領域を確保しておくことができるわけだ。

いままでのサンプルプログラムでは，それほど多量のスタックを必要としなかったので，COMMAND.Xはある程度大きめのスタックを取ってあるだろうという目論見のもとに，手を抜いてスタックの確保をしていなかった。もちろん，これはあまりほめられたことではない。これからは，きちんとスタックを確保することにしよう。

リスト1の場合はスタックをまったく使っていないので必要ではないのだが，参考までにスタックを確保するように修正しておく(リスト2)。

リスト2
TEST.S(修正版)

```
 1:          .include        doscall.mac
 2: *
 3:          .text
 4:          .even
 5: *
 6: start:
 7:          lea.l   mysp,sp         *spの初期化
 8:
 9:          move.b  #10,d0          *d0 = 10
10:          move.b  #20,d1          *d1 = 20
11:          add.b   d1,d0           *d0 = d0 + d1
12:
```

```
13:          DOS     _EXIT           *終了
14: *
15:          .stack
16:          .even
17: *
18: mystack:
19:          .ds.l   256             *スタック領域
20: mysp:
21:
22:          .end
```

スタック領域は19行で用意している。ここで使っている〝.ds.l〟という命令は，ロングワード単位でメモリを確保するアセンブラの疑似命令で，ここでは256ロングワード＝1024バイト分の連続したメモリを確保している[6]。

■ 6)また，.ds.bはバイト単位で，.ds.wはワード単位でメモリを確保する命令。.dc.bがあれば任意の大きさのメモリを確保できるわけだが，その領域に入れるデータのサイズに応じて「ワードデータ10個分」なら，同じ大きさのメモリを確保するのにも，

```
.ds.b    20
```

と書くより，

```
.ds.w    10
```

と書いたほうがわかりやすいだろう。

スタック領域確保の直前にある

```
.stack
.even
```

というのは〝.text〟と同様の「ここからスタック領域が始まりますよ」という意味のおまじないだ。

さて，メモリを確保しても，それがどこなのかわからなければ困るので，スタック領域の前後にラベルを置いて目印にしている。スタックはアドレスの大きいほうから小さいほうへ伸びていくわけだから，スタック領域の直後につけたラベルmysp(に定義されたアドレス)をSP(＝a7)に代入すれば，スタックは空の状態になり，初期化したことになる。この設定を行っているのが7行の

```
lea.l    mysp, sp
```

だ。

leaははじめて出てきた命令だが，これはアドレスレジスタにアドレスを代入する命令で，ここでの意味は，

```
move.l   #mysp, sp
```

と同じだ(〝#〟の有無に注意)。どちらの命令を使っても同じ働きをするわけだが，〝アドレスを代入する〟ことを明確にする意味で，leaのほうを使ってみた。スタックポインタを初期化するときの決まった形といえる。

このようにプログラム内部で設定したスタックポインタは，DOSコールexitを実行した時点で〝親のもともとのスタック〟を指すように戻される。このため，子プロセス側でスタックポインタを変更しても，親が困ることはない。

メモリ上のマシン語プログラム

メモリ上に読み込まれたマシン語プログラムがどのような形になっているかを調べてみよう。試しにLコマンドで逆アセンブルしてみると，79ページの図2-cのようなリストが表示される。68000の命令ではないが，よく使われるHuman68kのDOSコールを_EXITのようなわかりやすい形で表示してくれるのが嬉しい。また，_EXITの後ろに変な命令が並んでいるが，これはたまたまメモリ上に残っていたゴミデータが顔を出しただけのことだ。

図の逆アセンブルリストでは，左端に16進数が並んでいる。これは命令が置かれているアドレスを表している。これを見ると，間隔が一定ではないことに気づくだろう。上下のアドレスの差が2バイトのものと4バイトのものがある。命令によって何バイトのコードになるのかが変わるのである。

実際にどのようなマシンコードになっているかを知るには，Dコマンドでメモリをダンプしてみればよい。プログラム本体だけをダンプしてみると，図2-dのようになる。

たとえば，

```
move.b  #10, d0
```

は，

```
103C 000A
```

という2ワードのコードに対応していることがわかるだろう[7]。

■ 7) 68000の命令は，最短で1ワード(＝2バイト)，最長で5ワード(＝10バイト)になる。また，必ず偶数バイト長，つまりワード単位となっている。

Gコマンドと Tコマンド

DB.X上でプログラムを実行するにはGコマンドを使う。

```
-G＝実行開始アドレス
```

実行開始アドレスを省略した場合は，現在PC(プログラムカウンタ)が指すアドレスから実行が始まる。

プログラムを読み込んでデバッガを起動した直後は，PCはすでにプログラムの実行開始アドレスを指しているので，いまはたんに，

```
-G
```

と入力するだけで，TEST.Xが実行され，すぐに図2-eのようなメッセージを出して終了する。

TEST.Xは何の出力もないので実行してもおもしろくないから，今度はシングルステップ実行をしてみよう。

縁起ものだから，いったんデバッガを抜け，ふたたび

```
A>DB TEST.X
```

によって再起動する。79ページの図2-aの段階に戻ったわけだ。

シングルステップ実行(トレース)はTコマンドで行う。

```
-T＝トレース開始アドレス
```

のようにして使うのだが，例によってアドレスを省略すれば，現在のPCの位置からトレースが始まる。

まず，一度Tコマンドを実行すると，最初の命令

```
move.b  #10,d0
```

が実行され，またレジスタの状態が表示される。ちゃんとd0レジスタには$0A_H$が入っている。

つづけてTコマンドを何度か実行すると，d1への代入，加算が行われ，DOSコールexitをトレースした時点で正常終了した旨のメッセージが出る(図3)。

こんな感じでプログラムの動作を1ステップずつ確認していけば，マシン語がより身近に感じられるようになるだろう。いままでのサンプルプログラムを片っ端からトレースしてみるのもおもしろいかもしれない。

```
PC=000A1BA0 USP=000A1104 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
move.b  #$0A,D0
-t
PC=000A1BA4 USP=000A1104 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  0000000A 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
move.b  #$14,D1
-t
PC=000A1BA8 USP=000A1104 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  0000000A 00000014 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
add.b   D1,D0
-t
PC=000A1BAA USP=000A1104 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  0000001E 00000014 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
_EXIT
-t
program terminated normally
-
```

図3
シングルステップ実行
の様子

バイト単位で加算すると

さて，TEST.Xを少し変更して，ちょっとした実験をしてみる。最初に，d0.bに1バイトで表せる最大の数255(=FF_H)を，d1.bに1を入れておき，バイト加算した結果がどうなるのかを調べてみる。

単純な算数なら$FF_H+01_H=100_H$になるはずだが，〝バイト単位での演算〟の影響がどう出るかが見ものである(100_Hは1バイトでは収まりきらない)。

疑問に思ったらすぐに試せるのがデバッガのいいところだ。さっそくプログラムを書き直してみよう。

デバッガを抜け，ソースの修正からやり直してもいいが，この程度の修正であれば，DB.Xに

```
-a
  000A1BA0       move.b  #$0A,D0
                 move.b  #$ff,d0
  000A1BA4       move.b  #$14,D1
                 move.b  #$01,d1
  000A1BA8       add.b   D1,D0

-l a1ba0 a1bab
  000A1BA0       move.b  #$FF,D0
  000A1BA4       move.b  #$01,D1
  000A1BA8       add.b   D1,D0
  000A1BAA       _EXIT
-x
PC=000A1BA0 USP=000A1104 SSP=000067F2 SR=0000 X:0  N:0  Z:0  V:0  C:0
D  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
move.b  #$FF,D0
-t
PC=000A1BA4 USP=000A1104 SSP=000067F2 SR=8008 X:0  N:1  Z:0  V:0  C:0
D  000000FF 00000000 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
move.b  #$01,D1
-t
PC=000A1BA8 USP=000A1104 SSP=000067F2 SR=8000 X:0  N:0  Z:0  V:0  C:0
D  000000FF 00000001 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
add.b   D1,D0
-t
PC=000A1BAA USP=000A1104 SSP=000067F2 SR=8015 X:1  N:0  Z:1  V:0  C:1
D  00000000 00000001 00000000 00000000  00000000 00000000 00000000 00000000
A  000A1AA0 000A1BAC 000A1A14 00096FA0  000A1BA0 00000000 00000000 000A1104
_EXIT
-t
program terminated normally
-
```

図4
プログラムの修正と再実行

内蔵されたワンラインアセンブラで事足りる。これは1行のアセンブリ言語をその場でアセンブルしメモリに書き込む機能だ。

```
-A
```

でアセンブラのモードに入ったら，前ページの図4のように2行のソースを打ち込み，〝.〟を入力してアセンブラモードを抜ける。念のため逆アセンブルして，ちゃんとプログラムが書き変わっていることを確認したら，Xコマンドでレジスタの値を表示させてから順次トレースしてみよう。

add.bを実行した直後のd0レジスタに注目してもらいたい。0になっていることがわかる。このことから，add命令は加算結果がオペレーションサイズ(add.bならバイト，add.wならワード)を超えたときは，繰り上がりを無視して収まる範囲だけを残すということがわかる。

また，FF_Hに01_Hを足した結果が00_Hなのだから，FF_Hは〝マイナス01_H〟とみなしてもよいのではないかという発想が出てくる(コラム「負の数の表し方」参照のこと)。

COLUMN

負の数の表し方

これまでは，数はつねに0以上だと考え，符号の存在も負の数も無視してきた。実際，マシン語プログラムでは表立って負の数を扱わなければならない場合はそれほど多くない。それでもたまにはどうしても負の数を扱いたいときだってある。

当然こういった要求はプロセッサを設計するときに考慮されるべき問題だが，無符号(符号がつかない，つまりつねに0以上)の数用の加算命令，符号つき(＋，－の符号がつく。つまり，正の数だけではなく負の数も扱える)の数用の加算命令というようにばらばらに命令を用意するのも面倒な話だ。

そこで，1種類の演算命令で無符号数も符号つき数も扱えるように，〝負の数の表現〟の側に工夫がされている。

本文でも述べたように，バイト(＝8ビット)データのFF_Hに01_Hを足すと，本当は100_Hになるはずだ。が，8ビットに収まりきらなくなってしまうので，結果として下位8ビットである00_Hだけが残る。つまり，8ビットデータに限定した世界では，

$$FF_H+01_H=00_H$$

という不思議な式が成り立つ。

また，逆に00_Hから01_Hを引けばFF_Hになる。これはFF_Hが-01_Hとみなせることを意味している。このような形で負の数を表す形式を「2の補数表現」と呼ぶ。

さて，FF_Hが-01_Hならば，01_Hは$-FF_H$と考えてもかまわないことになる。こうなってくると，数の大小比較をどうすればいいのかが問題だ。で，考え出された(のかな？)が，〝データの最上位ビットが1であれば負の数とみなす〟方法だ。こういう決まりがあれば，

$$FF_H=11111111_B$$

だから，これは負の数で-1，

$$01_H=00000001_B$$

だから正の数で$+1$，というように見分けがつくようになり，符号つき数の比較もまた可能になる。

ここまでの話をまとめると，8ビットの数は無符号とみなせば，

00000000_B～11111111_B

10進に直せば,

0～255

の数を表現でき，符号つきとみなせば,

10000000_B～00000000_B～01111111_B

10進数で,

−128～0～127

の数を表せることになる。

ある8ビットデータが符号つきかどうかは，〝プログラマがどちらのつもりでいるか〟によってのみ区別される。どっちとみなすかはプログラマ次第ということだ(命令によっては，無条件に符号つきか無符号かが決まっているものもあるが)。そして，無符号のつもりだろうと符号つきのつもりだろうと，加算のときにはaddを使い，減算のときにはsubを使えばすむことになる。

さて，以上の話はそのまま16ビット以上のデータにも適用できる。16ビットデータの世界では,

$FFFF_H+0001_H=0000_H$

であり,$FFFF_H$は-0001_Hとみなすことができる。ここでも最上位ビットにより正負を判断するから，16ビットでは無符号で,

0000_H～$FFFF_H$

0～65535

符号つきと考えれば,

8000_H～0000_H～$7FFF_H$

−32768～0～32768

の範囲の数を表現できることになる。これは32ビットになっても同じことだ。

なお，2の補数表現で表された数の符号を表す最上位ビットのことを「符号ビット」という(そのままだね)。

演算とコンディションコード

上の実験でもう1点見逃せないのがコンディションコードの変化だ。add.b実行直後のレジスタの値を見てみるとCCRに変化が現れている。ZとCが1になっているはずだ。いよいよコンディションコードに関してきちんと話しておくべき時がきたようだ。

CCRはSRの下位8ビットにつけられた別名だ。このCCRというレジスタは，8ビット全体でいくつになっているかは問題ではなく，1ビットごとが独立した意味を持っている。1ビット長のレジスタが8つまとまったものと考えたほうがいいかもしれない。

実際には68000ではCCRの第0ビットから第4ビットまでが意味を持ち，残りの3ビットは〝CCRを8ビットという(コンピュータにとって)区切りのいい大きさに揃える〟だけの目的でついている。

CCR中の各ビットは，〝特定の状況〟かどうかを0か1で表すフラグとして働く。たとえば，Zは演算結果が0のとき1，そうでないとき0になり，Cは演算の結果，繰り上がりや桁借りがあったときに1になり，そうでないときは0になる。また，Nは演算結果が負のときに1になり，0または正のときは0だ[8]。

■ 8) それぞれZero，Carry，Negateの意味。残りのXとVはいまは覚える必要はない。

例の結果をこうしてみると，加算結果が0なので，Zが1でNが0，繰り上がりがあった(結果がバイトで収まらなかった)のでCが1になっている。

このように演算結果によってくるくる変わるコンディションコードが何のためにあるのかといえば，ずばり〝条件分岐〟のためだ。

すでにcmp命令で2つの数を比較し，直後にbeqかbneを置いて，等しいかどうかで分岐する方法は話した。じつをいうと，cmp命令は内部では〝subと同様に減算を行い，結果を捨てる〟という動作をしている。結果の値はどこにも残らないが，いちおう減算を行ったのでコンディションコードが変化する。もし2つの数が等しければ減算結果は0になるから，CCRのZビットは1になり，等しくなければ減算結果は非0だからZは0だ。

このZビットの状態によりbeqやbneで分岐するかどうかが決まる。結局，beqは〝Zが1であれば分岐する〟命令で，bneは〝Zが0であれば分岐する〟命令だったというわけだ。

beqとbneの正体が割れた以上，これら条件分岐命令の直前は必ずしもcmpを置く必要はなく，addやsubなどのコンディションコードを変化させる命令であれば何でもいいことがわかるだろう。前章で作ったプログラムの中には，〝レジスタから1を引いてみて，結果が0でなければ分岐する〟という処理があり，

```
sub.w    #1, d0
cmp.w    #0, d0
bne      loop
```

というコーディングをした。本当はsub命令を実行した段階で，結果が0かどうかはZビットに反映されているのでわざわざ0との比較をしなくても，

```
sub.w    #1, d0
bne      loop
```

と書けばすんだのだ。

コンディションコードに関してはまだまだ話しておかなければならないことがあるが，本章ではここまでで押さえておくことにしよう。

COLUMN

余談だが……

68000はメモリ保護の概念をかなり早い時期に取り入れたマイクロプロセッサである。当

時としては斬新だったが，その後発表されたプロセッサに比べれば多少見劣りする部分もある。

たとえば，完璧なマルチタスクOSを作るには，メモリ管理が大ざっぱすぎる。「OS-9ではやっているじゃないか」といわれそうだが，僕がいう〝完璧なマルチタスク〟というのは〝安全な〟という意味も含んでいるのだ。

つまり，OSをスーパーバイザ空間に，アプリケーションをユーザー空間で走らせるときに，OSはアプリケーションから保護されてはいるが，アプリケーションどうしはたがいに保護されていないので，〝事故〟が起こる〝可能性〟が残っている≠安全，というわけだ。

ハード側で細工する(いつでも任意のメモリ領域だけをユーザーモードにできるように設計するとか)か，そうでなければソフトウェアで面倒くさいことをする(アプリケーションもみんなスーパーバイザ空間に置いておき，ある瞬間に実行するアプリケーションだけをユーザー空間に転送して，また戻すとか。うーむ，悲惨だなぁ)ようにでもしないかぎり，安全は確保できない。どちらもハードウェア，ソフトウェアにかかる負担はかなり大きいだろう。

しかたがないから，OS-9などの場合は「アプリケーションはみんなお行儀がいい」ことを前提に，ユーザー側(アプリケーション側)に責任を押しつけて綱渡りをしているわけだ。

もっとも，僕はこれを68000の欠点だとは思っていない。68000はシングルタスク用のプロセッサと割りきっているからだ。そもそもマルチタスクというのはパワーの余ったCPUでやるから意味があるのであって，68000にはそんな余力はない(と，断言してしまうのが怖い)。

68030クラス以上を積んだマシンが個人でも手軽に手に入れられるようになるまでは，マルチタスクは実験か冗談にとどめておくつもりでいる。

CHAPTER 5

文字列操作の基本

CHAPTER 5

文字列操作の基本

ASSEMBLER

この章では文字列操作について書いてみる。これは次章以降で行う予定のファイル処理の準備段階にあたる。はっきりいえば，ファイル処理はマシン語らしくない地味な処理だ。そのわりにはかなり入り組んでいて〝正しく〟作るのは思いのほか骨が折れる。その前提である文字列の処理も，マシン語で書くと煩雑になってしまう場合が多い。ファイル処理・文字列処理はマシン語がもっとも苦手な分野といえるかもしれない。

それなのにあえてやるからには，それなりの見返りがあるはずなのだが……具体的にどんな〝いいこと〟があるかはちょっと説明できそうもない。読者が自力で見つけてくれることに期待したい。

文字列の表現法

コンピュータ内部では，〝文字〟はその文字コードという数字の形で表現される。〝文字列〟は言葉どおり文字が列になったものだから，文字コードの列によって表されるだろう。

たとえば，〝ABC〟という文字列は，任意のアドレス(かりに80000_H番地)から，〝A〟，〝B〟，〝C〟の文字コード(半角文字であればASCIIコード)を順に並べることで表現できる。

80000_H ← 41_H(A)
80001_H ← 42_H(B)
80002_H ← 43_H(C)

実際には，この形では文字列の大事な属性である〝長さ(何文字の文字列か)〟という情報が欠けている。文字列がどこで終わるかがわからないのだ。

ストレートな発想をするなら，先頭の1バイトに文字列の長さを格納しておけば，いちおうの格好はつく。

80000_H ← 03_H(文字列の長さ＝3)
80001_H ← 41_H(A)
80002_H ← 42_H(B)
80003_H ← 43_H(C)

というようにだ。これによって，先頭の1バイトを見れば文字列の長さがすぐ得られ，また文字列の終端がどこかを知ることができるようになる。

しかし，この方法には一定以上の長さの文字列を表現できないという欠点がある。つまり，1バイトで表すことができる数(0～255)によって文字列の長さが制限されてしまうからだ[1]。

■ 1) 一般のBASICでは，この形に似た形式で文字列を表現するので，文字列は255文字までという制限がある。

文字列を表現する別の考え方としては，ある決まったコードで文字列の終端を表すという方法がある[2]。かりに〝/〟で文字列の終わりを表現することにすれば，上の例は，

80000_H ← 41_H(A)
80001_H ← 42_H(B)
80002_H ← 43_H(C)
80003_H ← $2F_H$(/)

という形になる。この場合，先頭から終端コード〝/〟までの文字数(バイト数)を数えることで，文字列の長さが得られる。

■ 2) OSや言語によっては，この終端コードは〝$〟だったり復帰コード($0D_H$)だったりすることもある。Cは$00_H$で文字列の終わりを表す方式を採っている。ちなみに，00_Hで終わる形式の文字列のことを〝ASCIIZ文字列〟とか，縮めて〝ASCIZ文字列〟と呼ぶことがある。

この形式ならば，文字列の長さはいくらでも長くすることができる。ただ，終端コード自体を文字列の中に含むことができない(途中に終端コードを入れると，そこで文字列が終わってしまう)ので，終端コードを何にするかはうまく選ばなければならない。プログラムの中でつじつまがあっていれば何でもいいのだが，一般的なのは00_Hを終端コードにする方法だ。これはプログラムを書く際にもなかなか便利であり，Human68kのDOSコールもそうなっているので，今後，文字列はこの形で表現することにする。

ここで，メモリ上に置かれた文字列は，つねにその〝先頭アドレス〟によって指定することができることを覚えておいてもらいたい。たとえば，

80000_H ← 41_H(A)
80001_H ← 42_H(B)
80002_H ← 43_H(C)
80003_H ← 00_H(終端コード)

という文字列は，80000_Hという先頭アドレスひとつで間接的に表すことができるだろう。

さて，次のページのリスト1-aはDOSコールprintを利用した文字列表示の例だ。文字列はラベルmes以下に.dc.bによって用意し，その先頭アドレスをスタックに積んでDOSコールを呼び出している。.dcは前にもかんたんに紹介したが，この際，コラム「.dc疑似命令」で，もう少し詳しい話をしておく。

リスト1
PRT.S

```
a)
    1: *         DOSコールによる文字列表示
    2:
    3:           .include        doscall.mac
    4: *
    5:           .text
    6:           .even
    7: *
    8: ent:
    9:           lea.l   mysp,sp          *spの初期化
   10:
   11:           move.l  #str,-(sp)       *文字列先頭アドレスをプッシュ
   12:           DOS     _PRINT           *文字列表示
   13:           add.l   #4,sp            *スタック補正
   14:
   15:           DOS     _EXIT            *終了
   16: *
   17: str:      .dc.b    '1234ABCD'      *表示する文字列
   18:           .dc.b   $0d,$0a,0        *
   19: *
   20:           .stack
   21:           .even
   22: *
   23: mystack:
   24:           .ds.l   256              *スタック領域
   25: mysp:
   26:           .end


b)
    8: ent:
    9:           lea.l   mysp,sp          *spの初期化
   10:
   11:           pea.l   str              *文字列先頭アドレスをプッシュ
   12:           DOS     _PRINT           *文字列表示
   13:           addq.l  #4,sp            *スタック補正
   14:
   15:           DOS     _EXIT            *終了
```

COLUMN

.dc疑似命令

忘れられていると困るので念のため書いておくと,疑似命令というのは68000の命令ではなく，アセンブラに対する指令だ。ラベルに値を定義するのに使うequや,任意のメモリ領域を確保する.ds，そして，データを用意する.dcなどはみな疑似命令だ。

.dsがメモリを確保するだけなのに対し,.dcはメモリを確保して，そこを指定の定数で初期化する。例によって〝.b〟〝.w〟〝.l〟のバリエーションがあり，それぞれバイト長，ワード長，ロングワード長のデータを用意するのに使う。

たとえば，

```
.dc.b  10
```

は1バイトの領域を確保し，そこに10を入れておくことを意味する。いいかえると，10と

という値をそのままオブジェクトに埋め込むことになる。極端な話，アセンブリ言語の命令がどのようなコードに変換されるかを知っていれば，.dcだけを使ってプログラムを書くことができる。もっとも，そんなことをわざわざする人もいないだろうが。

.dcの後ろには，複数のデータをカンマで区切って並べることができる。

```
    .dc.b   10, 20, 30
```

なら，3バイトの連続したメモリ領域に10，20，30が順に書き込まれる。また，文字はクオーテーションマークでくくって表すことができるから，

```
    .dc.b   'A', 'B', 'C'
```

は

```
    .dc.b   $41, $42, $43
```

と同じ意味になるし，さらには，

```
    .dc.b   'ABC'
```

のようにまとめて表現することもできる。これは文字列データを用意する際の基本形といえる。

文字列は00_Hで終わるわけだから，改行コード$0D_H$や$0A_H$も文字列中に含めることができる点は見落とさないでもらいたい。必要とあらば，1つの文字列の中にいくつも改行コードをはさんで，1回の文字列表示で複数行の表示を行うことだってできるということだ。

ここで，今後のこともあってリスト1-aのメインルーチンを若干格好よく書き直してみた。リスト1-bだ。peaとaddqという新しい命令が顔を出している。これらに関しても，コラム「命令のバリエーション」と「実効アドレス」を参照してもらいたい。

COLUMN

命令のバリエーション

move，add，sub，cmp，and，or，eorは，厳密にはオペランドによっていくつかのバリエーションがある。このあたりは68000のドロドロとした部分で，真面目に区別しようとすると悲惨なことになるかもしれない。

たとえば，

```
    move.l      d0, a0
    add.l       d0, a0
    sub.l       d0, a0
```

という命令は本当はなく，正しくは，

```
    movea.l     d0, a0
    adda.l      d0, a0
    suba.l      d0, a0
```

と，Addressの〝a〟をつけた命令を使わなければならない。また，

```
    add.l       #10, d0
    sub.l       #10, d0
```

は，

```
    addi.l      #10, d0
```

```
    subi.l      #10, d0
```

のようにImmediateの〝i〟をつけるのが正しい。

といって，

```
    movei.l     #10, d0
    addi.l      #10, a0
```

という命令はありそうで，ない。

さて，洞察力に優れた人なら，上に挙げた例からなんらかの規則性を見つけることができるだろう。また，几帳面な人なら，『アセンブラマニュアル』などで各命令とアドレッシングモードの組み合わせの有効・無効を調べようと思ったかもしれない。

しかし，たぶん多くの読者は「めんどくせー」と思ったことだろう。その感覚は全面的に正しい。これはいってみれば，BASICのPRINT文に整数表示用，実数用，文字列用が別々に用意されているようなもので，使い分けるのはうざったいだけだ。

そこで，たいていの68000用アセンブラは，たんにmoveと書けばアセンブラが勝手に適切な命令に置き換えてくれるように作られている。だから，読者は慌ててmoveaだの，addiだのなどの新しい命令を覚える必要はなく，これまでどおりデータ転送ならmove，加算ならadd，減算ならsub，論理演算ならand，or，eorを使えばよい。

とはいうものの，僕は運よくというか運悪くというか，命令を正しく使い分けるのが癖になってしまっている。今後のリストにはmoveaやsubiなどがしばしば登場することになるだろう。読者に僕の癖を押しつけるつもりはないが，いちおう，ここで話したことを頭に入れておいて，リスト中の〝知らない命令〟に驚かないようにしてもらいたい。

ただの潔癖性だと思われるのは癪なのでちょっと弁解しておくと，僕が命令をフルスペルで書くのには保険みたいな意味がないでもない。

```
    add.l       #10, d0
```

の〝#〟をつけ忘れて

```
    add.l       10, d0
```

と書いてしまうのはありがちな単純ミスだし，

```
    move.l      d1, a0
```

は1ヵ所打ちまちがうだけで，

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

に化けてしまうが，いつも

```
    addi.l      #10, d0
    movea.l     d1, a0
```

と書くようにしていれば，万一ミスタイプしても，

```
    addi.l      10, d0
    movea.l     d1, a0
```

をエラーとしてアセンブラがはじいてくれる。

僕はドジなもので，なるべくミスが早目に発見できるように予備工作しておかないと信頼性の高いプログラムが作れないのだ。

というところで，コラム「クイック・イミディエイト・アドレッシング」に続く。

COLUMN

クイック・イミディエイト・アドレッシング

move, add, subのバリエーションにはmoveq, addq, subqというのがあって，クイック・イミディエイト・アドレッシングはこれら3命令だけで使えるアドレッシングモードだ(いうまでもなくqはQuickの頭文字である)。AS.Xでは〝a付き〟，〝i付き〟の命令同様，たんにmove，add，subと書いておけば，使えるところでは適切に〝q付き〟に置き換えてくれるから，無理に覚える必要はないのだが，無視するわけにもいかないのでいっきに解説しておく。

イミディエイトというからには，イミディエイトデータ(即値)を扱うアドレッシングモードであり，それぞれの書式は，

```
moveq.l     #1,d0
addq.l      #1,d0
subq.l      #1,a0
```

のようになる。ふつうのイミディエイト・アドレッシングとの違いは，〝コードが短く，実行速度が速い〟という1点につきる。たとえば，

```
move.l      #1,d0
```

が

```
2F00    0000    0001
```

の3ワードのコードに変換されるのに対して，

```
moveq.l     #1,d0
```

は，

```
7000
```

の1ワードですむ。コードが短くなる分，メモリから命令コードを読み出す時間も短くなるので，実行速度も3倍になる。

それなら何でもかんでもmoveqを使えばいいじゃないかと思った人，残念でした。クイック・イミディエイト形式は〝扱える数の範囲などに制限がある〟のだ。

そもそも，

```
move.l      #1,d0
```

が3ワードものコードになってしまうのは，1というロングワードのデータもコード化しなければならないからだ。68000の命令コードはワード単位なので，1ワードを命令自体に，2ワード(＝1ロングワード)をデータにあてると，合計3ワードになってしまう。

ちなみに，命令自体を表す1ワードのコードはビットフィールドになっていて(数ビット長の〝フィールド〟をいくつかまとめて1ワードにしてある)，2ビットでmoveという命令であることを，2ビットでオペレーションサイズを，また，各6ビットずつでソース/デスティネーションオペランドのアドレッシングモードを表すようになっている(詳しくは『アセンブラマニュアル』などを参考のこと)。

moveqは，扱える数の範囲を1バイトの符号つき数で表せる範囲の数(−128〜127)に制限してデータ部分を1バイトに詰め込み，さらにサイズはロングワード固定，デスティネーションオペランドはデータレジスタのみに限定することで，強引に1ワードのコードに押さえているのだ。

つまり，データレジスタへロングワードの−128〜127を代入する場合にのみ，moveqの

恩恵に与れるというわけ。

addq, subqはサイズとデスティネーションオペランドの自由度は下げないかわりに，扱える数の範囲を1～8に制限して，やはり1ワードのコードにまとめている。1～8というのは範囲が狭すぎるように見えるかもしれないが，ポインタとして使っているアドレスレジスタを少しずらすとか，(似たようなものだが)DOSコール呼び出し後のスタックポインタを補正するとか，カウンタを増減するなどの用途に使う分には十分だ。

他のプロセッサでは〝1を足す命令〟，〝1を引く命令〟が特別に用意されているものだが，68000のaddq, subqはそれらの命令を拡張したものと考えてよい。

なお，moveqに関しては，さらにコラム「符号拡張」も参照してほしい。

COLUMN

符号拡張

前章で話した〝負の数の表現〟を思い出してもらおう。−1は8ビットの2の補数表現で表すと，

$11111111_B=FF_H$

になるのだった。また，−1を16ビットの2の補数表現で表せば，

$1111111111111111_B=FFFF_H$

になる。

いま，8ビットの符号付き数を16ビットに変換することを考える。正の数であれば，

8ビット		16ビット
00_H(0)	→	0000_H(0)
01_H(1)	→	0001_H(1)
$7F_H$(127)	→	$007F_H$(127)

のように上位に8ビット分の0を補えばいいことはすぐにわかるだろう。

ところが，負の数は単純に0を付け足すと，

80_H(-128)	→	0080_H(128)
FE_H(-2)	→	$00FE_H$(254)
FF_H(-1)	→	$00FF_H$(255)

というおかしなことになってしまう。この場合は，8ビット分の0ではなく1を補って，

80_H(-128)	→	$FF80_H$(-128)
FE_H(-2)	→	$FFFE_H$(-2)
FF_H(-1)	→	$FFFF_H$(-1)

と変換しなければならない。

2の補数表現で表された数の正負は，符号ビットが0なら正，1なら負とみなすという約束だったから，上の〝正なら8ビットの0を補い，負なら8ビットの1を補う〟という規則は，〝上位ビットを符号ビットで埋める〟とまとめることができる。もちろん，この規則は8ビット→16ビットだけでなく，16ビット→32ビット，8ビット→32ビットなどの場合にもそのまま適用できる。

別項でmoveq命令は内部的にはデータを8ビットで表し，それを32ビットに展開するという話をした。これも符号拡張のなせる技だ。

さて，ここで質問しよう。

```
moveq.l    #128,d0
```

を実行すると，d0レジスタはいくつになるでしょう。また，それはなぜでしょう(答えは載せないよ)。

COLUMN

実効アドレス

混乱しないように気をつけて読んでもらいたい。

```
move.l  LABEL, d0
```

は〝LABELというラベルで指定されるアドレスからのロングワードデータをd0に転送する〟という意味だが，ここで〝LABELで指定されるアドレス〟のことを「実効アドレス(effective address)」と呼ぶ。

```
move.l  (a0), d0
```

の場合は，〝a0でポイントされるアドレス(＝a0の値)〟が実効アドレスとなる。

つまり，〝データを指定するのに使われるアドレス〟のことを「実効アドレス」という。

68000にはこの実効アドレス自体を対象とする命令として，leaとpeaが用意されている。

lea(Load Effective Address)は，実効アドレスをアドレスレジスタに入れるための専用の命令で，アドレスを扱う都合上，サイズはつねにロングワードになる。

```
lea.l    LABEL, a0
```

は〝LABELで指定されるアドレスそのもの〟をa0レジスタに代入する。動作としては，

```
movea.l #LABEL, a0
```

とまったく変わらない。また，

```
lea.l    (a1), a0
```

は〝a1で指定されるアドレスにあるデータのアドレス(結局a1の値)〟をa0レジスタに入れる。つまり，

```
movea.l a1,a0
```

と同じ動作だ。

メモリの読み書きをするとき，プロセッサはアドレスバスと呼ばれる信号線でアドレスを指定し，その後データバスを介してデータのやりとりをすることになるのだが，leaは〝本当はアドレスバスに乗るはずだった値を無理やりデータバスに乗せてデータにしてしまい〟，アドレスレジスタに入れる命令といえる。

pea(Push Effective Address)は，実効アドレスをスタックにプッシュする命令で，デスティネーションがつねに〝－(sp)〟だということがわかっているから省略して，

```
pea.l    LABEL
pea.l    (a1)
```

のように記述する。それぞれの動作は，

```
move.l  #LABEL, －(sp)
move.l  a1, －(sp)
```

と同じ意味だ。

さて，ここまでの説明ではlea，peaの有用性が見えないと思う(moveで代用できるのだ

から)。まだ説明していないアドレッシングモードと組み合わせると，カッコよかったり，速かったり，コードが短かったりする場合があるのだが，いまのところはそういうメリットもない。

現時点では〝対象がアドレスであることを明確にする意味で使う〟ということにしておこう。

他言語にみる文字列操作

マシン語での文字列操作のしかたに入る前に，他の言語では文字列をどのように扱っているかを確認しておく。おなじみのX-BASICとCを取り上げよう。

X-BASICでは，

```
str a, b, c
```

というぐあいに文字列型の変数を用意しておけば，

```
a="ABCDEFG"
b=a
c=a + b
if (a=b) then ～
```

のように代入・連結・比較などの処理がかんたんに行える。見た目では数値変数を扱う場合と何も変わらないといっていい。さすがは高級言語というべきか。

C言語になると，文字列型の変数というものはない。が，char型の配列を，

```
char a[256], b[256], c[256];
```

のように定義してから，あらかじめ用意された文字列操作関数を使い，

```
strcpy(a, "ABCDEFG");
strcpy(b, a);
strcat(strcpy(c, a), b);
if (!strcmp(a, b))～
```

という感じで同等の処理ができる。

では，マシン語はというと，文字列変数などという便利なものはないし，文字列操作用の命令が用意されているわけでもない。

ここで，Cにおける文字列の扱い方には参考になる部分も多い。

```
char a[256];
```

を〝256バイトの領域を確保し，その先頭アドレスをaで表す〟というように読みかえると，これはアセンブリ言語の

```
a:  .ds.b  256
```

に直接対応することがわかる。aやbなどの配列名を〝アドレスを表すラベル〟だと思えば，

```
strcpy( b,a );
```

は，〝a，bという2つのアドレスを指定して，strcpyというサブルーチンを呼び出す〟ようなものだ。

こと文字列操作に関するかぎり，Cとマシン語の差はstrcpyなどの関数があらかじめ用意されているかどうかの違いしかないことになる。となると，Cの文字列操作関数に相当するサブルーチンを一度作っておけば，後はそのサブルーチンを使い回しにすることでC程度には楽に文字列を扱えるようになるだろう。

次の節以降では，C風の文字列操作サブルーチンをいくつか作ってみよう。

文字列の複写

アドレスstr2から格納された文字列を，アドレスstr1以下にコピーすることを考える。文字列は00_Hで終わることにしてあるから，この処理は〝アドレスstr1から00_Hに出会うまで，1バイトずつstr2以下に転送する〟といいかえることができる。

連続したメモリ領域を操作するには〝ポインタ〟の考え方が有用だ。68000のマシン語でいうと〝アドレスレジスタ間接アドレッシング〟を利用することになる。

まず，2つのメモリ領域の先頭アドレスをアドレスレジスタに入れておく。a0に転送先，a1に転送元を入れることにすると，

```
lea     str1,a0
lea     str2,a1
```

により，a0，a1はそれぞれの領域へのポインタとして使えるようになる(a0，a1は間接的にstr1，str2以下のメモリ領域を表す代名詞のようなものと考える)。こう，お膳立てを整えてから文字列を複写するサブルーチンを呼び出す。これは関数へ引数を渡すようなものだ。

実際のサブルーチン側の中身は次のようになる。

1) a0の指すアドレスから1バイト取り出し，a1の指すアドレスに転送する。
2) 次の文字に備えてa0，a1ともに1を足す。
3) 〝いま転送した1バイトデータ〟が00_Hでなければ，文字列がまだ続いていることになるので1)に戻る。
4) 00_Hであれば転送は終了した。

ここで，文字列終了コードの00_Hも転送に含めなければならないことに注意してもらいたい。00_Hを転送しないとどうなるかを考えれば，その理由は明らかだ(よね？)。

上の話をそのままプログラムにすると，次のページのリスト2-aのようになる。ラベルstrcpy以下が文字列をコピーするサブルーチンだ。

リスト2
STRCPY.S

a)

```
1: *       文字列の複写
2:
3:         .include        doscall.mac
4: *
5:         .text
6:         .even
7: *
8: ent:
9:         lea.l   mysp,sp         *spの初期化
10:
11:        lea.l   str1,a0         *複写先へのポインタ
12:        lea.l   str2,a1         *複写元へのポインタ
13:        bsr     strcpy          *文字列複写
14:
15:        pea.l   str1            *コピーした文字列を
16:        DOS     _PRINT          *表示してみる
17:        addq.l  #4,sp           *
18:
19:        DOS     _EXIT           *終了
20: *
21: *文字列複写サブルーチン
22: strcpy:
23:        move.b  (a1),d0         *1文字取り出し
24:        move.b  d0,(a0)         *転送
25:        addq.l  #1,a0           *転送先ポインタを進める
26:        addq.l  #1,a1           *転送元ポインタを進める
27:        cmpi.b  #0,d0           *転送したのは終了コードか？
28:        bne     strcpy          *そうでなければ繰り返す
29:        rts
30: *
31: str2:  .dc.b    '1234ABCD',0    *複写元
32: str1:  .ds.b    256             *複写先
33: *
34:        .stack
35:        .even
36: *
37: mystack:
38:        .ds.l   256             *スタック領域
39: mysp:
40:        .end
```

b)

```
21: *文字列複写サブルーチン
22: strcpy:
23:        move.b  (a1)+,d0        *1文字取り出し
24:        move.b  d0,(a0)+        *転送
25:        tst.b   d0              *転送したのは終了コードか？
26:        bne     strcpy          *そうでなければ繰り返す
27:        rts
28: *
29: str2:  ～
```

```
c)
   21: *文字列複写サブルーチン
   22: strcpy:
   23:          move.b   (a1)+, d0          *1文字取り出し
   24:          move.b   d0, (a0)+          *転送
   25:          bne      strcpy             *終了コードまで繰り返す
   26:          rts
   27: *
   28: str2:    ～


d)
   21: *文字列複写サブルーチン
   22: strcpy:
   23:          move.b   (a1)+, (a0)+       *1文字転送
   24:          bne      strcpy             *終了コードまで繰り返す
   25:          rts
   26: *
   27: str2:    ～
```

転送元文字列はラベルstr2以下に.dc.bで，転送先領域はstr1以下に.ds.bでそれぞれ用意している。転送先は，転送される文字列が十分格納できるだけの大きさを確保しておかなければならない。ここではゆとりを持って256バイトとってある。

さて，〝ポストインクリメント・アドレスレジスタ間接形式〟を利用すれば，22～29行のあたりはリスト2-bにまで簡略化できる。どさくさに新しい命令tst(tstはTeSTの略)を使っているが，これは，

```
cmpi    #0, ～
```

とまったく同じ働きをする(つまり，0との比較専用の)命令だ。データが0かどうか，また正か負かを調べるのに使う。

で，せっかく出てきたtst命令だが，move命令は，〝データを転送し，同時に0と比較してコンディションコードに反映する〟ので(さらっと書いたが，これは重要)，いまの場合はtstが省略できて，リスト2-cになる。

さらに，moveでコンディションコードが変化するのなら，いちいち〝データをレジスタに取り出し，転送して0と比較する〟必要もなくなり，最終的に文字列転送処理はリスト2-dの形になる。ここまでかんたんになってしまうと，もうカッコイイとしかいいようがない。

文字列の連結

文字列の複写ができれば連結もかんたんだ。上の例同様，a0に転送先，a1に転送元文字列の先頭アドレスが入っているとすると，

1) a0が転送先文字列の最後を指すようにポインタを進める。これはa0がポイントするアドレスから順に00_Hを探すことで行う。

2) 後は文字列複写と同等の処理を繰り返す。

つまり，転送先文字列の末尾の00_Hを上書きする位置以降に文字列を転送すれば，文字列の連結が行える。

プログラムにすると，リスト3のようになる。22～25行が終了コードを探す処理だ。ポストインクリメントしている関係で，00_Hが見つかった時点でのa0レジスタは〝00_Hの次のアドレス〟を指しているから，つじつまあわせに25行でa0から1を引いている。

リスト3
STRCAT.S

```
 1: *        文字列の連結
 2:
 3:          .include        doscall.mac
 4: *
 5:          .text
 6:          .even
 7: *
 8: ent:
 9:          lea.l   mysp,sp         *spの初期化
10:
11:          lea.l   str1,a0         *連結先へのポインタ
12:          lea.l   str2,a1         *連結元へのポインタ
13:          bsr     strcat          *文字列連結
14:
15:          pea.l   str1            *連結した文字列を
16:          DOS     _PRINT          *表示してみる
17:          addq.l  #4,sp           *
18:
19:          DOS     _EXIT           *終了
20: *
21: *文字列連結サブルーチン
22: strcat:
23:          tst.b   (a0)+           *(a0)は0か？
24:          bne     strcat          *そうでなければ繰り返す
25:          subq.l  #1,a0           *行きすぎたから1つ戻る
26: strcpy:
27:          move.b  (a1)+,(a0)+     *1文字転送
28:          bne     strcpy          *終了コードまで繰り返す
29:          rts
30: *
31: str2:    .dc.b   '1234ABCD',0    *連結元
32: str1:    .dc.b   '5678EFGH',0    *連結先
33:          .ds.b   256             *ゆとり
34: *
35:          .stack
36:          .even
37: *
38: mystack:
39:          .ds.l   256             *スタック領域
40: mysp:
41:          .end
```

文字列の比較

つづいて，指定された2つの文字列が等しいか等しくないかを判断するサブルーチンを作ってみよう。理屈は単純で，2つの文字列の先頭から1バイトずつ比較して，最後まで一致したら2つの文字列は等しいし，途中で不一致の箇所がみつかれば等しくない，というだけのことだ。

リスト4は，a0，a1で指定される2つの文字列を比較するサブルーチンの例だ。2つの文字列の長さが等しいとはかぎらないので，28行で比較文字列(どちらでも同じことだが，便宜上a1のほう)の最後まで比較が終わったかどうかを調べている。まだ文字列が残っていれば30行のcmpm命令[3]で比較を行い，1文字でも不一致ならすぐにループを抜ける。

■ 3) cmpm命令(末尾の"m"はMemoryの略)は2つのアドレスレジスタでポイントされるメモリ領域を比較する命令で，アドレッシングモードは必ずポストインクリメント・アドレスレジスタ間接形式になる。
つまり，
cmpm.X (an)+, (an)+
の形でだけ使われる。なお，例によってAS.Xでは，
cmp.b (a0)+, (a1)+
と書けば
cmpm.b (a0)+, (a1)+
と解釈してくれる。

リスト4
STRCMP.S

```
 1: *        文字列の比較
 2:
 3:         .include        doscall.mac
 4: *
 5:         .text
 6:         .even
 7: *
 8: ent:
 9:         lea.l   mysp,sp         *spの初期化
10:
11:         lea.l   str1,a0         *被比較文字列へのポインタ
12:         lea.l   str2,a1         *比較文字列へのポインタ
13:         bsr     strcmp          *文字列比較
14:
15:         beq     match           *一致したか？
16:
17: not:    pea.l   notmes          *一致しなかった
18:         bra     match0
19:
20: match:  pea.l   matmes          *一致した
21: match0: DOS     _PRINT          *それなりのメッセージを
22:         addq.l  #4,sp           *       表示
23:
24:         DOS     _EXIT           *終了
25: *
```

```
26: *文字列比較サブルーチン
27: strcmp:
28:         tst.b   (a1)            *比較文字列は終わりか？
29:         beq     strcmp0         *そうであればループを抜ける
30:         cmpm.b  (a1)+,(a0)+     *1文字比較
31:         beq     strcmp          *一致している間繰り返す
32:         rts                     *一致しなかった
33: strcmp0:
34:         cmpm.b  (a1)+,(a0)+     *ラストチャンス
35:         rts
36: *
37: str1:   .dc.b   '1234ABCD',0    *非比較文字列
38: str2:   .dc.b   '1234ABCD',0    *比較文字列
39: *
40: matmes: .dc.b   '一致しました',$0d,$0a,0
41: notmes: .dc.b   '一致しません',$0d,$0a,0
42: *
43:         .stack
44:         .even
45: *
46: mystack:
47:         .ds.l   256             *スタック領域
48: mysp:
49:         .end
```

被比較文字列のほう(a0でポイントされるほう)が先に終わってしまったときのことが心配かもしれないが，その場合は被比較文字列の終端コード00_Hと，比較文字列中の１文字(00_Hではない)が比較されることになり，ちゃんと不一致が検出される。

また，28行で被比較文字列が終わっていることがわかった場合は単純に不一致にしてしまわないで，〝もしかすると比較文字列も終わっているかもしれない〟ので34行でラストチャンスを与える。

最終的に，このサブルーチンは文字列が一致したらZ＝１，不一致ならZ＝0でリターンする。後はサブルーチンから戻ったら，beq，bneで処理を振り分ければよい。リスト4では文字列が一致したかどうかに応じてそれなりのメッセージを表示するようにしてある。文字列を適当に変えて試してみてほしい。デバッガで動作を確認するのもおもしろいだろう。

また，BASICの文字列比較やCのstrcmpでは〝文字列の大小比較〟が行えるが，いま作ったサブルーチンは，期せずしてこれにも対応している。リターン時のccrのZビットとCビットの組み合わせで，文字列の大小がわかるのだ。

文字列の長さを得る

文字列複写のときに，文字列の先頭から順に00_Hを探す処理が出てきた。文字列の長さを知りたければ，その〝00_Hを探す処理〟の過程で何バイト飛ばしたかを数えればよい。

リスト5は文字列の長さを数え，結果が何文字かを表示するプログラムの例だ。長さを数えるサブルーチンstrlenは，d0.wをカウンタにして文字列の先頭から00_Hまでのバイト数を数えている(00_H自体は数えない)。結果はそのままd0.wに返す。カウンタがワードなので文字列は65535文字以下でなければならないが，実用上の問題はないだろう。

リスト5
STRLEN.S

```
 1: *       文字列の長さを数え表示する
 2:
 3:         .include        doscall.mac
 4: *
 5:         .text
 6:         .even
 7: *
 8: ent:
 9:         lea.l   mysp,sp         *spの初期化
10:
11:         lea.l   str1,a0         *文字列へのポインタ
12:         bsr     strlen          *文字列の長さを数える
13:
14:         bsr     prtdec          *結果を10進表示
15:                                 *改行はしていない
16:         DOS     _EXIT           *終了
17: *
18: *文字列の長さを数えるサブルーチン
19: strlen:
20:         moveq.l #-1,d0          *カウンタの初期化
21: strlen0:
22:         addq.w  #1,d0           *カウント
23:         tst.b   (a0)+           *終了コードか？
24:         bne     strlen0         *そうでなければ繰り返す
25:         rts
26: *
27: *D0.Wを10進左詰めで表示するサブルーチン
28: prtdec:
29:         movem.l d0/a0,-(sp)     *d0,a0をスタックに待避
30:
31:         andi.l  #$0000ffff,d0   *上位ワードをクリア
32:         lea.l   bufend,a0       *ポインタ初期化
33: prtdec0:
34:         divu.w  #10,d0          *d0.lを10で割る
35:                                 *       上位ワード = 余り
36:                                 *       下位ワード = 商
37:         swap.w  d0              *上位ワードと下位ワードを交換
38:                                 *       上位ワード = 商
39:                                 *       下位ワード = 余り
40:         addi.w  #'0',d0         *0～9 → '0'～'9'
41:         move.b  d0,-(a0)        *1桁格納
42:         clr.w   d0              *次の除算に備える
43:                                 *       上位ワード = さっきの商
44:                                 *       下位ワード = 0
45:         swap.w  d0              *上位ワードと下位ワードを交換
46:                                 *       上位ワード = 0
47:                                 *       下位ワード = さっきの商
48:         bne     prtdec0
```

```
49:
50:          move.l   a0,-(sp)
51:          DOS      _PRINT
52:          addq.l   #4,sp
53:
54:          movem.l  (sp)+,d0/a0
55:          rts
56: *
57: str1:    .dc.b    '1234567890ABCDEFGHIJK',0        *テスト文字列
58: *
59: buff:    .ds.b    5                *10進文字列格納領域
60: bufend:  .dc.b    0                *文字列の終了コード
61: *
62:          .stack
63:          .even
64: *
65: mystack:
66:          .ds.l    256              *スタック領域
67: mysp:
68:          .end
```

20行でカウンタに使うd0.wを−1で初期化する。0ではなく−1で初期化するのは，22〜25行で終端コードをチェックするのに先立ってカウントアップするためだ。この順序を逆にし，

```
tst.b       (a0)+   *テスト
addq.w      #1,d0   *カウントアップ
bne         〜
```

とやってしまうと，せっかくtstで変化させたコンディションコードがaddqでさらに変化してしまい，正しい結果が得られない。

このサブルーチンよりも28行以下の〝d0.wを10進左詰めで表示するサブルーチン〟のほうが複雑なので，ちょっと脱線して次節で解説を加えてみよう。

10進表示

リスト5のprtdecは，d0.wを無符号数と見なして10進数左詰めで表示するサブルーチンだ。いちおう汎用性を狙ったので，かなりがっちり作ってある。

話をかんたんにするために，d0.wには0〜9の10進1桁で表せる範囲の数しか入っていないものとすると，d0.wに〝0〟のASCIIコードを足して表示すれば10進表示が行える。

また，d0.wが0〜99の範囲に収まるのなら，

1) 10で割った商に〝0〟のASCIIコードを足して表示

2) 1)の余りに〝0〟のASCIIコードを足して表示

という2段階の処理で10進2桁表示が行われる。

いま，d0.wには0〜65535の数が入っている可能性があるわけだから，10進2桁の場合を拡張

して，次のような手順で10進5桁での表示が行えるだろう。

1) 10000で割った商に〝0〟のASCIIコードを足して表示
2) 1)の余りを1000で割った商に〝0〟のASCIIコードを足して表示
3) 2)の余りを100で割って表示
4) 3)の余りを10で割って表示
5) 4)の余りを1で割って(結局そのまま)表示

このアルゴリズムでは，表示する数の上位桁から順に1桁ずつ取り出しているが，逆に下位桁から取り出す手順もある。

1) 10で割った余りに〝0〟のASCIIコードを足し，結果をどこかにしまっておく。
2) 1)の商が0でなければ1)に戻る。
3) 1)～2)により表示すべき文字が1の位から順に求まるので，最後にこれを逆順に表示する。

以上，2つの方法を比べると，前者は上位桁から処理するから，その場その場でどんどん表示できるが，10000で割って，1000で割って……というあたりが冗長な感じがする。対して後者は，逆順になるため，一度どこかのメモリに文字を溜めておく必要がある半面，単純なループで処理することができる。

どちらも一長一短があるわけだが，リスト5ではいちおう後者の方法を採った。

さて，アルゴリズムはこれでいいとして，それをプログラムで実現するにあたってはマシン語で割り算する方法を知らなければならない。幸いなことに68000には除算専用の命令divuがあるので，これを使わせてもらおう(「除算命令」のコラムを参照してほしい)。

COLUMN

除算命令

68000には割り算を行う命令としてdivuとdivsの2つが用意されている。それぞれDIVide as Unsigned，DIVide as Signedの略で，無符号数と符号付き数の割算を行う命令だ。符号付き数の除算はめったに使われないので，まずはdivuのほうだけ覚えておこう(使い方は変わらないけど)。

divuは，

```
divu.w  #10,d0
divu.w  d1,d0
```

のようにして使う。サイズはワード固定だ。

デスティネーションはつねにデータレジスタであり，32ビットすべてが演算に使用される。また，ソースオペランドは16ビットだけが演算に使われる。演算結果はデスティネーションに指定されたデータレジスタに格納されるが，この格納のされ方がちょっと変わっていて，下位ワードに商が，上位ワードに余りが返されることになっている。

d0.lが256(=00000100_H)のとき

```
divu.w  #10,d0
```

を実行すると，商が25(=0019_H)，余りが6(=0006_H)だから，結果のd0.lは

d0.l = 00060019_H

になる。この後,

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

とすれば，商がd1.wに取り出され，別項で説明するswap命令を使って

```
swap.w  d0
```

により，d0.lの上位ワードと下位ワードを交換してから,

```
move.w  d0, d2
```

で今度は余りがd2.wに取り出される。

商か余りが16ビットで収まらなかったときには,演算が失敗した印にccrのVビットを立て(1にして),デスティネーションは変化しない(=除算は実行されない)。このため,divuは32ビット÷16ビットとはいうものの，実質的には16ビット÷16ビット程度の演算にしか利用できないと考えてよいだろう。

また，0で割ろうとしたときは68000がそれを検出し,X68000の場合は画面中央に〝0で除算しました〞というエラーメッセージを出してプログラムの実行を中止する。中止されて困るのであれば，divu実行前にソースオペランドが0でないことを調べておかなければならないということだ。

では,107ページのリスト5のprtdecサブルーチンを頭から順に見てもらいたい。最初にいきなり,

```
movem.l     d0/a0, -(sp)
```

という変な命令がある[4]。書式からだいたい見当はつくと思うが，これはレジスタをまとめてスタックにプッシュするのに使う命令だ。ここでは，d0, a0の2つをスタックに待避している。

■ 4) movemの末尾のmはMultiple registersの略で，〝たくさんのレジスタ〞程度の意味。movemでは複数のレジスタを〝/〞で区切って指定することになっているが，d0〜d2, a2〜a4をまとめて待避するような場合には,

```
movem.l d0-d2/a2-a4, -(sp)
```

と書くことが許されている。
ちなみに，〝d0/a0〞や〝d0-d2/a2-a4〞の部分のことを〝レジスタリスト〞と呼ぶ。

また，待避したレジスタの値はサブルーチンの最後で,

```
movem.l  (sp)+, d0/a0
```

により，やはりまとめて復帰している。

このレジスタの待避・復帰は，汎用性のあるサブルーチンを作るときの常套手段だ。つまり，サブルーチンの中で使うレジスタをメインルーチンで使っている場合に備えて，値を保存しておくわけだ。こうしておけば，メインルーチンで使っているレジスタをあやまって壊してしまう心配がなくなり，サブルーチンの独立性が高くなる。他のプログラムに組み込むときにも，ただそのまま持っていけばいい。

つづいてサブルーチンの初期化部分。まず,

```
andi.l      #$0000ffff, d0
```

により，d0.lの下位ワードはそのままで上位ワードを0にしておく。これは後でdivu命令で割算をするのに必要な処理だ。

さらに,

```
lea     bufend, a0
```

により，数値を〝数字を表す文字〟に変換した結果を格納するバッファ[5]領域へのポインタを初期化する。いま表示するのは0～65535までの5桁の数なので，バッファも5バイト用意しておけばよい。

■ 5)バッファ(buffer)は〝データを溜めておく場所〟程度の意味。

ポインタがバッファの先頭ではなく最後を指すようにしているのは，さきほどいった〝逆順〟の関係だ。表示のときにひっくり返すのではなく，最初から逆に格納しておこうというわけだ。これにより，変換結果を最後にまとめて〝文字列〟として表示することができる。バッファの直後に.dc.bで文字列の終了コード00_Hをあらかじめ書き込んであるのもこのためだ。うーん，深慮遠謀。

33行以下が数字を1桁ずつ取り出すメインループだ。

```
divu.w  #10, d0
```

により，d0を10で割り，d0の下位ワードに商，上位ワードに余りを求める。

とりあえず必要なのは，上位ワードに入っている余りのほうなので,

```
swap.w  d0
```

により，d0の上位ワードと下位ワードを交換する。この段階でd0.wには10で割った余りが入っているから，〝0〟のASCIIコードを足し，a0の指す領域へ格納する。a0に対してプリデクリメント・アドレスレジスタ間接アドレッシングを適用しているのがポイントだ。

ここで次のループに備え，ふたたび,

```
swap.w  d0
```

で上位ワードと下位ワードを交換するのだが，その前に,

```
clr.w   d0
```

でd0の下位ワード(すぐひっくり返すから結果として上位ワード)を0でクリアしておく[6]。

■ 6)clr(当然CLeaRの略)はデスティネーションを0クリアする命令で,

```
clr.X  ～
```

は,

```
move.X    #0, ～
```

と同じ動作をする。0を代入する専用命令だから，一般にmoveを使うよりも速くて短いコードになる。唯一の例外は〝データレジスタの32ビットを0にする〟場合で，このときは,

```
clr.l     d0
```

よりも

```
moveq.l   #0, d0
```

のほうが速い。また，clrはアドレスレジスタに対しては使えないが，AS.Xでは,

```
clr.l     a0
```

を

```
suba.l    a0, a0
```

に置き換えるというアクロバットを見せる。これは少々やりすぎのような気がするが。

これは次の除算に備えるためで，冒頭の

```
andi.l    #$0000ffff,d0
```

と同じような役割を果たしている。

swapは，交換の結果が0であればZビットを1にするのでbneで処理を振り分ける。交換結果が0でなければ(まだ数字が残っていることになるから)ループし，Z=1なら変換が完了したことになるからループを抜ける(なお，swapに関してはコラムを参照のこと)。

COLUMN

swap命令

swapはデータレジスタの上位ワードと下位ワードを交換する命令だ。交換はワード単位なので，オペレーションサイズもワード固定になっている。結局，書式は次のような形になる。

```
swap.w  d0
```

念のため例を挙げておくと，d0.lが12345678_Hのとき，

```
swap.w  d0
```

を実行すると$d0.l=56781234_H$になり，再度

```
swap.w  d0
```

を実行すれば，$d0.l=12345678_H$となり元に戻る。

変換がすんだ時点で，a0は変換結果の文字列の先頭をぴったり指しているから，そのままスタックに積んでDOSコールprintを呼び出せば，表示が完了する。

フィルタへの第一歩

以上の基本的な文字列操作ができるようになれば，その応用で文字列の英大文字→小文字変換やその逆変換(strlwrやstruprに対応)，文字列を逆順に並べ換える(strrev)，文字列の部分を取り出す(left$，mid$，right$)などの処理も行える。当然，これは読者への課題である。

最後に，次章へのつなぎとして〝キーボードから入力した文字列の英小文字をすべて英大文字に置き換えて表示するという処理をえんえんと続けるプログラム〟をリスト6に示す。これは，標準入力からの入力になんらかの処理を加えて標準出力に出すフィルタコマンドのシンプルな例になっている。文字列入力待ちのときに行の先頭でCTRL+Z(CTRLキーを押しながらZのキーを押す)を入力し，リターンキーを押すことで終了する。

リスト6
UPPER.S

```
 1: *       英小文字→英大文字変換フィルタの出来そこない
 2:
 3:         .include        doscall.mac
 4: *
 5:         .text
 6:         .even
 7: *
 8: ent:
 9:         lea.l   mysp,sp         *spの初期化
10:
11: loop:   pea     buff            *1行入力
12:         DOS     _GETS           *
13:         addq.l  #4,sp           *
14:
15:         lea.l   str,a0          *a0=入力文字列先頭
16:         cmpi.b  #$1a,(a0)       *先頭は^Zか？
17:         beq     skip            *そうであれば終了
18:
19:         bsr     strupr          *小文字→大文字変換
20:
21:         pea.l   str             *変換結果を表示
22:         DOS     _PRINT          *
23:         addq.l  #4,sp           *
24:
25:         pea.l   crlf            *改行
26:         DOS     _PRINT          *
27:         addq.l  #4,sp           *
28:
29:         bra     loop            *えんえんと繰り返す
30: skip:
31:         DOS     _EXIT           *終了
32: *
33: *英小文字→英大文字変換サブルーチン
34: strupr:
35:         tst.b   (a0)            *文字列の終わりか？
36:         beq     strup1          *そうであれば変換終了
37:
38:         cmpi.b  #'a',(a0)       *英小文字か？
39:         bcs     strup0          *
40:         cmpi.b  #'z'+1,(a0)     *
41:         bcc     strup0          *
42:
43:         subi.b  #$20,(a0)       *小文字なら大文字に変換
44:
45: strup0: addq.l  #1,a0           *ポインタを進める
46:         bra     strupr          *繰り返す
47: strup1: rts                     *サブルーチンからリターン
48: *
49: buff:   .dc.b   255             *入力可能最大文字数
50: cnt:    .dc.b   0               *入力された文字数
51: str:    .ds.b   256             *文字列入力バッファ
52: *
53: crlf:   .dc.b   $0d,$0a,0       *改行コードだけの文字列
54: *
55:         .stack
56:         .even
```

```
57: *
58: mystack:
59:         .ds.l   256             *スタック領域
60: mysp:
61:         .end
```

標準入力からの文字列読み込みにはDOSコールgetsを使っている。このDOSコールは,

```
.dc.b   n           * 入力可能最大文字数
.dc.b   x           * 入力された文字数
.ds.b   n+1         * 文字列格納領域
```

の形で用意したメモリ領域の先頭アドレスをスタックに積んでから呼び出す。文字列の格納領域は，文字列の終端コードの分も計算に入れ，入力可能最大文字数よりも1バイト多く用意しておかなければならない。

この領域の先頭アドレスをかりにbuffというラベルで表すと，DOSコールから戻った時点で,

```
d0.l        ← 実際に入力された文字数
buff+1      ← d0.lと同じ値
buff+2～    ← 入力された文字列
```

が返される。なお，入力された文字列末尾の改行コードは自動的に00_Hに置き換えられて格納されることになっている。

文字列の小文字→大文字変換はサブルーチンstruprで行っている。a0でポイントされるメモリ領域を順に1バイトずつ取り出して英小文字かどうかを調べ，そうであれば大文字に変換してから格納しなおす。

英小文字かどうかの判断は，現在注目している1文字のASCIIコードが〝a〟～〝z〟に収まっているかどうかで調べている(38～41行)。

さて，このプログラムには手抜きがいっぱいある。まず，全角文字の入力が考慮されていない。全角文字列を適当に打ち込むと，たまに変な文字に化けることがあると思う。

また，標準入力をリダイレクトしたときの動作にも問題がある(あるなんてもんじゃない)。試しに,

```
A>upper <upper.s
```

とやってみると，DOSコールprintには本来あるはずのブレイクチェックが効かないことがわかる。当然，^Sによる一時停止も働かない。

しかも，TABコードが〝^I〟という2文字になっているし，最後まで表示した時点で黙りこくってしまう(改行コードが入力されるのを待ち続けているのだ)。止めるには本体上面にあるINTERRUPTスイッチ(おっと，PROでは本体前面か)のお世話になるしかない。

このあたりのことを教訓にして，次章では正しいフィルタの作り方を紹介する。

COLUMN

続・余談だが……

ビープ音，ベル，某マシンではブザー。BASICのエラーのときなどにピーっと鳴るあの音だ。X68000では大胆にもビープ音をAD PCMで鳴らし，PCMデータさえ用意すれば好きな音でビープ音を鳴らすことができるわけで，X68000を手に入れた人が1度はビープ音を変えて遊んでみるとまでいわれている。「ジュワッ」とか「OUCH!」とか「やめて，お兄さん……」とか，ね。あ，いや，そういう話じゃなくって。

思い出したときに話しておかないと忘れそうなので，ここで，ビープ音を鳴らす方法について話しておく。

調べてみた人にはわかると思うが，『プログラマーズマニュアル』のどこを見てもビープ音を鳴らすDOSコールはみつからない。もっと探すとIOCSコールにAD PCMを鳴らす機能があるのだが，これは一般的なPCMの鳴らし方であって，CONFIG.SYSで設定したビープ音を鳴らすものではない。

では，どうやるかというと，コントロールコードを使って鳴らすことになっていたりする。$0D_H$，$0A_H$というコントロールコードを使って改行を行ったように，07_Hというコードを〝画面〟に出力すればビープ音が鳴る。

標準出力がリダイレクトされていない前提であれば，

```
move.w    #$07,-(sp)
DOS       _PUTCHAR
addq.l    #2,sp
```

によってベルを鳴らすことができる。

しかし，標準出力がファイルにリダイレクトされていたりすると，ファイルに07_Hというコードが埋め込まれるだけで音は出ない(試してみること)。あくまで〝画面〟にコントロールコードを送らなければならないのだ。

画面に直接文字を出力するには，標準出力のかわりに〝標準エラー出力〟を使う(コマンドモードからリダイレクトする方法がないので，通常は画面に直接対応している)，IOCSコールの文字表示ファンクションを利用する(OSを通さないからOSの機能であるリダイレクションは働かない)などの方法もあるが，DOSコールにも〝画面直接出力〟を行うものがちゃんとある。

以前に1度使ったDOSコールconctrlはいくつかの機能を兼ねていて，その中の1つに画面への1文字直接出力が用意されている。これを利用すると，ビープ音を鳴らす処理は，

```
move.w    #$07,-(sp)    *ベルのコード
clr.w     -(sp)         *モード0
DOS       _CONCTRL      *1文字出力
addq.l    #4,sp         *sp補正
```

というように書ける。

話はぐっと軽くなる。僕は最近ある仮説を立てた。それは「プログラマのセンスはビープ音の使い方に現れる」というものだ。

センスのよいプログラマはビープ音にまで気を使って適切なところで適切に鳴らす。ときにはあんまりタイミングよく鳴るものだから「鳴らしてくれてありがとう」という気にさえなる(実話である。が，どのプログラムだったかは忘れてしまった)。

対して〝そうでないプログラマ〟は，どういうわけかやみくもにベルを鳴らしたがり，エラーが発生したときや無効なキー入力があったときはもちろん，ひどいのになるとプログラム起動時にピー，処理の合間にピー，実行終了間際にピー，とビープ音を鳴らしまくる。気持ちはわからないでもない。メッセージを表示するだけではなんとなく寂しいような気がしてアクセントをつけたくなるのだろう。しかし，あまりピーピー(もしくは「ジュワッジュワッ」とか「やめてお兄さん，やめてお兄さん」)鳴らされると，だんだんいらいらしてきてしまう。

では，回数が少なければそれでいいかといえばそうでもない。〝最悪のタイミングで鳴らすわずか１回のビープ音〟がすべてをぶち壊しにすることだってあるのだ。

「たかがビープ音ごときにそんなに目くじら立てることはないんじゃない？」という意見もあるだろうが，無神経にビープ音が鳴るようなプログラムはきっと他の部分も無神経に作ってあるだろうし，ひょっとするとバグがいくつも潜んでいるかもしれないじゃないか(説得力ない？)。

さて，時を同じくして，僕はまた別の仮説を立てた。いわく「プログラマのセンスは画面の色使いに表れるの法則」である。

起動メッセージが意味もなく青かったりするのはまだ可愛いが，エラーメッセージが〝黄色のリバース＋強調〟だったりすると，作ったヤツの頭を思いきりド突きたくなる。

ま，そういう話。

CHAPTER 6

正しいフィルタの作り方

CHAPTER 6

正しいフィルタの作り方

ASSEMBLER

この章の題材は〝フィルタ〟である。半角の英小文字を大文字に変換する(他の文字は素通りさせる)フィルタを例にとって「フィルタはこう作る!」というところを見てもらおうと思う。プログラムはかりにUPPER.Xと名づけよう。これは僕たちが作る初の実用プログラム(使い道があるかどうかは別にして)になる。

とはいえ,最初からすぐに使えるプログラムを作ろうなんて欲張ると後で後悔することになるから,まずはできる範囲で作る。そして,いちおう動くものができたら動作試験をしてみて,不備を探し出しては修正していく。この試す・直すというサイクルは,プログラムを作り,デバッグする過程そのものであり,本章はそのあたりもちょっと強調してみるつもりでいる。なお,フィルタに関してはコラムを参照のこと。

COLUMN

フィルタ

狭義では,標準入力からデータを読み込み,適当な処理を加えて結果を標準出力に書き出すプログラムのことを〝フィルタコマンド〟ないしはたんに〝フィルタ〟という。この場合,扱うデータはテキスト(文書,文字データ)であることが仮定されている。

標準入出力は通常コンソール(キーボードとCRT)に割りつけられているから,その状態でのフィルタは〝キーボードからデータを読み込み(処理を加えて)結果を画面に表示する〟だけのプログラムとして機能する。しかし,DOSのサービス(正確にはCOMMAND.Xの機能)であるリダイレクションを利用して,

```
A>FILTER <FILE1
```

というぐあいに標準入力をファイルにリダイレクトしてやれば,〝FILE1からデータを読み込んで結果を画面に出す〟ようになるし,

```
A>FILTER >FILE2
```

と標準出力をリダイレクトすれば,〝キーボードから読み込んだデータを処理して結果をFILE2に書き出す〟プログラムに早変わりする。

さらには,

```
A>FILTER <FILE1 >FILE2
```

と入出力ともにリダイレクトすれば,〝FILE1からデータを読み込んで結果をFILE2に書き出す〟こともできるし[1],

```
A>FILTER <FILE1 >PRN
```

なら，〝FILE1からデータを読み込み，結果をプリンタに出力する〟ことになる。

フィルタ関係でリダイレクションと並んでもう1つ重要なDOSのサービス(これも本当はCOMMAND.Xの機能)が〝パイプ〟で，これは複数のフィルタを連結するものだ。たとえば，

```
A>FILTER1 <FILE1 | FILTER2
```

とやれば，FILTER1の出力がそのままFILTER2の入力となる[2]。もちろん，フィルタはさらに3つ4つといくらでもつなげて使うこともできる[3]。

ひとつひとつのフィルタは単純な機能しか持っていなくても，このようにリダイレクションを活用したりパイプでつないだりすることで，より複雑な処理を行うことができるようになる。逆にいえば，個々のフィルタは単純な処理だけを行うように作ればすむし，むしろ単純であればあるだけ応用が効くともいえる。

■ 1) フィルタをリダイレクションと組み合わせて使う場合，

```
A>FILTER <FILE1 >FILE1
```

のように入力ファイルと出力ファイルに同一のものを指定すると，ファイル内容が失なわれるから気をつけよう。

■ 2) Human68kでのパイプは一時ファイルを介して行われる。つまり，一段目のフィルタの出力はこっそりファイルに書き出され，そのファイルが次のフィルタの入力となる。

この一時的に作成されたファイルはコマンドの実行後自動的に削除される。要するに，

```
A>FILTER1 | FILTER2
```

は，内部では

```
A>FILTER1 >TMP
A>FILTER2 <TMP
A>DEL TMP
```

と同等な処理に置き換えて実行されている。なお，この一時ファイルが生成されるパスは，

```
A>TEMP A:¥
```

のようにして指定しておくことができる。この指定がない場合にはカレントディレクトリに一時ファイルが生成される。RAMディスクが使えるのであれば，一時ファイルをRAMディスク上に生成するようにしておくと速くて静かである。

■ 3) 実際には，COMMAND.Xのコマンド行の最大入力文字数＝255文字によって制限を受ける。

小文字→大文字変換の手順

標準入出力を使った文字の取り扱いはもう飽きるほどやった。後は英小文字を大文字に変換する方法を押さえれば，とりあえず〝小文字→大文字変換フィルタ〟の原型を作ることができる。小文字→大文字変換の手法は前章最後のプログラムで示してあるが，ここでもう一度詳しく話しておこう。

この処理は，次の2つの段階からなる。

1) 対象となる文字が英小文字かどうか調べる。

2) そうであれば大文字に変換する。

しごく当然。

第1のステップである英小文字かどうかの判断はASCIIコードの比較で行う。〝a〟～〝z〟の文字には61_H～$7A_H$の連続したASCIIコードが割り当てられているから，下限の61_Hと上限の$7A_H$との計2回比較し，その大小関係から小文字とそうでないものに選り分けることができる。つまり，次のような手順になる。

1-1) 下限である61_Hと比較する。61_H未満であれば小文字ではない。

1-2) 上限である$7A_H$と比較する。$7A_H$より大きければ小文字ではない。

1-3) 1-1)と1-2)のチェックに引っかからなければ小文字である。

第2のステップもまたASCIIコードレベルでの単純な操作だ。〝A〟～〝Z〟の文字には41_H～$5A_H$のASCIIコードが割り当てられており，これは小文字のコードとちょうど20_Hずれている。ということは，小文字のASCIIコードから20_Hを引けば大文字に変換できるわけだ[1]。

■ 1) 小文字→大文字変換の処理は20_Hを引くかわりに，DF_HとANDをとっても同じ結果が得られる。理由は各自考えよう。

以上の考えをそのままサブルーチンの形にすると，リスト1のようになる。このサブルーチンtoupperはd0.wにASCIIコードを入れて呼び出すと，小文字→大文字の変換を行って，結果をd0.wに入れて戻る。

リスト1
英小文字→大文字変換サブルーチン

```
1: *半角英小文字→英大文字変換サブルーチン
2:
3: toupper:
4:          cmpi.b  #'a',d0          *英小文字か？
5:          bcs     toupr0           *
6:          cmpi.b  #'z'+1,d0        *
7:          bcc     toupr0           *
8:          subi.b  #$20,d0          *小文字なら大文字に変換
9: toupr0: rts                       *サブルーチンからリターン
```

4～7行が小文字かどうかを調べている部分だ。この判定の結果，英小文字と判断されたら8行で20_Hを引いて大文字に変換する。なお，ここで使っているbcc，bcsに関してはコラム〝無符号数の大小比較〟を参照してもらいたい。

COLUMN

無符号数の大小比較

条件分岐の基本中の基本である，〝比較してみた結果が等しければ分岐する〟とか〝等しくなければ分岐する〟手法は何度も顔を出し，cmp後にbeqやbneを使えばよいのだった。ここでbeqとbneの実体は，〝Zフラグがセットされていれば(1ならば)分岐する〟命令と，〝Zフラグがリセットされていれば(0ならば)分岐する〟命令だということ，68000にはZ以外にCやNなどのフラグもあったことを思い出してもらうと，Z以外のフラグを条件とする分岐命令もあるだろうということが想像できる。

bcs, bccはCフラグを条件とする分岐命令で, csとccはそれぞれCarry Set, Carry Clearの略だ。言葉どおり, C(キャリ)フラグがセットされているときに分岐する命令と, リセットされているときに分岐する命令である。

どういう用途に使うかは, Cフラグがどんな場合に変化するかを考えてみるとわかるだろう。Cフラグは加算の結果が演算サイズを超えて繰り上がりが発生した場合や, 減算の結果, 桁借りが発生した場合に立つ(セットされる＝1になる)フラグだから, まず, 加減算のオーバーフローの判定に使えるだろう。

たとえば, d1.wが$FFFF_H$でd0.wが0002_Hのときに,

```
add.w     d1,d0
```

を実行すると, 加算結果は10001_Hになってしまい, オペレーションサイズ(この場合はワード)で表現できる範囲の数を超えるので, Cフラグがセットされる(もちろんd0.wには下位ワードの0001_Hだけが残る)。ワードからの繰り上がりが生じたときに分岐したければ, この直後に

```
bcs       ～
```

を置けばよいし, 繰り上がりが生じなかったときに分岐したければ,

```
bcc       ～
```

を使えばよい。

また同様の設定で,

```
sub.w     d1,d0
```

を実行すると, d0.wには0003_Hという結果が得られる。これを無符号演算と考えると, 2から65535(＝$FFFF_H$)が引けないので, 上位から桁借りし, 10002_H-FFFF_Hと見なして演算したものと考えられる。そして, この桁借りが発生したことを表すために, やはりCフラグがセットされ(意味的にはCarryではなくBorrow), 直後に

```
bcs       ～
```

を置けば, 〝桁借りが生じたときに分岐する〟ことができる。

次に, 比較時にCフラグが持つ意味について考えてみる。比較命令cmpは〝減算を行い, フラグだけを変化させて結果を捨てる〟命令だから, subとフラグの変化はまったく同じだ。subでは, 減算の結果, 上位桁からの桁借りがあったときにCフラグがセットされたわけだが, 桁借りが生じたということは被減数が減数より小さかったということだ。すると,

```
cmp.w     d1,d0
```

実行後〝(無符号数で考えて)d0.wがd1.wより小さい〟ときにはCフラグが立つ。つまり,

```
cmp.w     d1,d0
bcs       ～
```

は, d0.wのほうがd1.wより小さいときに分岐する,

```
cmp.w     d1,d0
bcc       ～
```

なら, d0.wがd1.w以上のとき分岐する, という意味になる。bcs, bccの導入により, 無符号数の大小比較が行えるようになったわけだ。

68000には, さらにCフラグとZフラグを組み合わせて条件分岐する賢い命令が用意されている。これまでに出てきたものとあわせて, 下にまとめておく。

```
bhi       C＝0かつZ＝0のとき分岐
```

bcc　C＝0のとき分岐
beq　Z＝1のとき分岐
bne　Z＝0のとき分岐
bls　C＝1またはZ＝1のとき分岐
bcs　C＝1のとき分岐

ここでなにかを感じとってもらえれば話は早いのだが，ひらめくものがなければ次を見てほしい。これは，

cmp　X,Y

実行後のフラグ変化とXとYの大小関係との関連を示したものだ。

Y＞X　C＝0,Z＝0
Y≧X　C＝0,Z＝0またはC＝0,Z＝1
Y＝X　C＝0,Z＝1
Y≠X　C＝0,Z＝0またはC＝1,Z＝0
Y≦X　C＝1,Z＝0またはC＝0,Z＝1
Y＜X　C＝1,Z＝0

すべての可能性を網羅したのでちょっとごちゃごちゃしているが，下線部が必要十分条件である。

つまり，上に挙げた6つの条件分岐命令により，任意の(無符号数の)大小比較が行える。ちなみに，bhiのhiはHIgh，blsのlsはLess or Sameの意味だ。

さて，しつこく〝無符号数の場合〟ということを強調してきた。ということは，符号付き数の比較にはここで紹介した方法は使えないということだ。では，符号付き数の比較を行う方法はないのかというと，じつはちゃんとある。が，符号付き数を扱う場面はあまりないので，いまここで覚える必要はない(よって紹介もしない)。まとめて覚えようとすると，かえって混乱してしまうことにもなりかねないが，ゆとりのある人は各自調べてみてほしい。

試作する:UPPER.X第1版

材料が揃ったところで，UPPER.Xの第1版をリスト2に示す。プログラム自体はかんたんなものだから，とくに解説すべき点もない。DOSコールgetcで1文字読み込み，英小文字だったらサブルーチンtoupperで大文字に変換してからputcharで標準出力に書き出す。これを無限ループの中でえんえんと繰り返している。

リスト2
UPPER.S(その1)

```
1: *       英小文字→英大文字変換フィルタ　第1版
2:
3:         .include        doscall.mac
4: *
5:         .text
6:         .even
7: *
```

```
 8: ent:
 9:          lea.l   mysp,sp          *spの初期化
10:
11: loop:    DOS     _GETC            *1文字入力
12:          bsr     toupper          *小文字→大文字変換
13:          move.w  d0,-(sp)         *1文字出力
14:          DOS     _PUTCHAR         *
15:          addq.l  #2,sp            *
16:          bra     loop             *えんえんと繰り返す
17: *
18: *英小文字→英大文字変換サブルーチン
19: toupper:
20:          cmpi.b  #'a',d0          *英小文字か？
21:          bcs     toupr0           *
22:          cmpi.b  #'z'+1,d0        *
23:          bcc     toupr0           *
24:          subi.b  #$20,d0          *小文字なら大文字に変換
25: toupr0:  rts                      *サブルーチンからリターン
26: *
27:          .stack
28:          .even
29: *
30: mystack:
31:          .ds.l   256              *スタック領域
32: mysp:
33:          .end
```

では，アセンブル後，

A>UPPER

で起動し，適当にキーボードから文字を打ち込んで，正しく動作するかどうかを確認してもらいたい。英小文字が大文字に変換されるかどうか確認するのはもちろん，〝そうでない文字〟が化けたりしないかどうかなどの点についても忘れずにチェックしてほしい。いいかげんに作ってあるだけあって，早々に問題点が見つかるだろう。

まず，リターンキーを押したときの改行動作がおかしい。カーソルは行頭に戻るだけで次の行には移動してくれない。そのため，続けて入力した文字が重ねて表示されてしまい，たいへん見苦しい。この原因を探るために，プログラムの動作を追ってみよう。

リターンキーを押すと，$0D_H$というコードが入力される。これは英小文字ではないので，大文字への変換処理は行われずに素通りし，そのまま標準出力に吐き出される。$0D_H$は〝カーソルを行頭に移動する〟コントロールコードだから，これによってカーソルは行の先頭に戻る。この後，プログラムはループしてふたたび文字の入力待ちになり，次に入力された文字は重ねて表示されることになる。

Human68kでは，改行は$0D_H$と$0A_H$という2つのコードで行うことになっているから，$0A_H$の〝カーソルを次の行に移動する〟動作が欠けている分，改行が中途半端になってしまっているわけだ。

結局，この症状はOSの改行コード($0D_H+0A_H$)とキーボードから入力される改行コード(0D

$_H$)との差に原因があった。本来，このような場合はOSがつじつまをあわせてくれる(キーボードから$0D_H$を入力したら，自動的に$0D_H+0A_H$の入力に変換してくれる)べきだと思うのだが，少なくともDOSコールgetcにはそういうサービスはないようだ。

そこで，$0D_H$が入力されたら$0D_H$に続けて$0A_H$も〝入力されたことにしてやる〟ことを考える。〝本来ならOSにやってほしかったこと〟をプログラム側で肩代わりしてやろうという発想だ。けれど，無条件に$0A_H$をつけるような単純な処理では，後でまた別の問題が生じることになる。この件に関してはもう少し動作試験をしてから結論を出すことにしよう。

さて，UPPER.Xの不備その2。こんなことは動作試験をするまでもないのだが，無限ループになっている関係でいつまでたってもプログラムが終了しない。これも立派なバグといえるだろう。DOSコールgetc内ではブレイクチェックが効くから，BREAKキーを押せば止まることは止まるのだが，スマートとはいえないし，標準入力をファイルにリダイレクトしたときにどうなるかが心配だ。後でテストしてみる必要がある。

動作試験の心構え

上で見つけた2つの動作不良は，ちょっと試してみれば誰にでも見つけられる類のものだった。しかし，現実のプログラムのバグはこんな素直なものばかりとはかぎらない。もう大丈夫だと思ったプログラムに，ある日突然ひょんなことから虫が見つかるというのはよくある話である。後で困りたくなければ，考えうるかぎりの状況を想定して事前に厳しく動作試験するほかないだろう。見つからないバグはつぶしようがないのだ。そのためには動作試験のときにどれだけの〝状況〟を考えつけるかが鍵になる。

ここで一時脱線して，一般的な動作試験時のコツというかポイントをかんたんにまとめておこう。これは，なにもマシン語にかぎった話ではない。

たとえば，X-BASICで平方根を求めるプログラムなり関数なりを作ったとする。平方根ぐらいなら関数として用意されているが，あえてなにやら得体のしれないアルゴリズムを駆使して作ったものとしよう。プログラムはいちおう動くようになり，動作試験をすることになった。読者なら，どこをどうやってテストするだろうか。

2とか3とか，平方根がいくつになるか暗記している数を引数として与え，「ひとよひとよにひとみごろ」とか「ひとなみにおごれや」とかブツブツいいながら，結果が正しいかどうか見比べるぐらいのことは誰でもやりそうだ。もう少し多くの数について結果を求め，数表なり電卓なり，すでに動作が確認されている他のプログラム(BASICの関数など)と比較すれば，なお安心だろう。このとき可能であれば，目で見比べるのではなく，プログラムでチェックできればより確実になる。

次に〝誤った入力があった場合〟に何が起こるかも確かめておかなければならない。これはプログラムにとっては〝入力が適切なときに正しい結果を返す〟ことと同じくらい重要なこと

だ。プログラムにエラーチェックを組み込んであるのなら，故意にエラーを引き起こすようなデータを与えてみて，エラーチェック自体が正しく機能するかどうかを確認しておく。平方根プログラムの場合であれば，引数が負の数だったら(複素数を扱うことを考えないかぎり)エラーとしてはじくように作られているはずだから，−1かなんかを食わせてみて〝予定どおりエラーになる〟ことを確かめる。エラーチェックを組み込むことを忘れていた場合には，この段階でプログラムの異常が発見されるだろう。

しかし，やっかいなことに，エラーチェックを忘れたときには，そもそも〝プログラムに不当な入力があるかもしれないこと〟も忘れてしまっている場合が多い。プログラムを作ったときの先入観を動作試験に持ち込んでしまうと，肝心なことを見落としてしまいかねない。動作試験に臨むときには，スパッと頭を切り替える必要がある(口でいうほどかんたんなことでもないが)。この場合の必殺技は，〝制作者とは完全に別な頭〟つまり他人に動作試験してもらうことだろう。人数は多ければ多いほどよい。「完成したら君にも使わせてあげるからさー」とかなんとかいって友達を丸め込もう。

１人黙々とデバッグしているのであれば，無理をしてでもさらに〝状況〟をひねり出してみる。

〝極端な場合〟というのも要チェック事項である。この平方根プログラムも，もしかすると非常に大きな数を与えると演算過程でオーバーフローして誤った答えを返すかもしれないし，絶対値の非常に小さな数を与えると，予想外に精度が低下するという症状が見つかるかもしれない。これらは実際に大きな数や小さな数を与えて動作試験しなければ，潜在的なバグとして後々まで残ってしまうことになる。

架空の平方根プログラムでは関係ないが，〝条件判断の境界付近〟も重点チェック項目の１つだ。〝10以下〟と判断すべきところを〝10より大きい〟と逆にしてしまうようなミスは単純なだけに見つけにくいが，実際に10前後の数をデータとして与えて実験してみれば一目瞭然だ。

それから，ループが正しい回数だけ回るかどうかも確かめておかなければならない。BASICのforループのような一定数回すループで，ループカウンタの初期値が１少なかったり多かったりするのはかなり(マシン語では非常に)危険なバグである。また，不定回数のループ(X-BASICのwhile～endwhileやrepeat～untilで表されるような構文)では，〝最初から条件が成立している場合は１度もループを通らないのか，それとも１度は回るのか〟という点がチェック項目になる。これは設計時に考慮すべきことだが，実際に何通りかの条件下でプログラムを動かしてみればはっきりするだろう。

プログラムを作り慣れないころは，どこからテストすればよいのか，なかなかわからないかもしれない。が，いくつもプログラムを作っていくうちに，〝この間はこういうことがあったから，今度はこうかな〟という勘(？)が働くようになる。これを推し進めていけば，動作試験時ではなく，設計の段階でバグをつぶせるようになる寸法だ。

では，ここで話したことを教訓にUPPER.Xをいま一度厳しく動作試験してみてもらいたい。致命的なバグ，〝一部の文字が不当に化ける〟という症状が見つかるはずだ(見つからない

ようではまだまだ頭が堅い)。

このUPPER.X3つ目のバグは,〝全角文字を入力すると,変な文字に化けて出力される場合がある〟というものだ。たとえば〝敏幸〟と入力してみると〝媛幸〟のように化ける。これは漢字コードの関係で起こる症状であり,詳しくはコラム〝日本語の呪い〟で説明しておく。

COLUMN

日本語の呪い

英語圏では,コンピュータが扱う文字といえば英数字と若干の記号だけだ。これはたかだか100個程度の数であり,7ビット(128通り)もあればカバーできる範囲だし,もう少し特殊な記号やセミグラフィック文字などを加えたとしても,8ビット=1バイト(256通り)のコードで十分表現できる範囲である。

対して,日本では漢字というものがあるおかげで,1バイトのコードでは足りずに,2バイトで1文字を表す漢字コードがASCIIコードと併用されている。〝併用〟というのが悲惨なところで,これは文字を扱うプログラムを作るうえで,ときにややこしい問題を引き起こす。本文にあるUPPER.Xの第1版は,見事にこの問題に引っかかってしまっている。つねに1文字=1バイトである英語圏のプログラマとは違って,僕たちはこの〝日本語の呪い〟を引きずっているのだ。

漢字コードには何種類かあるが,ふつう使われるのは〝JIS漢字コード〟と〝シフトJIS漢字コード〟だけと考えて差し支えないだろう。X68000では表向きシフトJISコードが採用されているが,内部的にはときおりJISコードを使う場面もあるようだ。

JISコードでは,第1,第2バイトともに21_H~$7E_H$の範囲の数だけが使われる。つまり2121_Hから始まり,$217E_H$の次は2221_Hに飛ぶわけだ。JISコードを考えた人は,00_H~$1F_H$と$7F_H$(スタンダードなASCIIコードでは〝カーソル位置の1文字削除〟の意味を持つ)のコントロールコードおよび20_Hのスペースとダブらないように気をつかったらしい。80_H以降を使わないのは,文字を7ビットで表すようなプログラムを考慮したためだろうか。

JISコードの最大の欠点は,ASCIIコードと単純に区別できないということだ。文字列中に41_H,42_Hというバイト列があった場合,これを〝AB〟というASCIIコード2文字と見なしたらよいのか,JISコードの1文字〝疎〟と見なしたらよいのかがわからない。苦肉の策として,文字列中の漢字の前後に〝漢字イン(「ここから漢字ですよ」という意味のコード)〟と〝漢字アウト(「ここまでが漢字ですよ」という意味のコード)〟を置く方法が使われることになった。この方法では〝疎疎〟という文字列は

漢字イン,41_H,42_H,41_H,42_H,漢字アウト

という形で表される。

が,この方法には〝データが長くなる〟,〝部分文字列を抜き出すのが面倒〟などの欠点が残っている。

シフトJISコードはもう少しうまい方法で1バイトコード(ASCIIコード)と混在できるように作られている。シフトJISコードの1バイト目に漢字インコードの意味をあわせ持たせるとでもいったらいいのだろうか。

シフトJISコードでは,第1バイトは81_H~$9F_H$,$E0_H$~EF_Hが,第2バイトは40_H~$7E_H$,80_H~FC_Hが使われる。第1バイトには妙な隙間があるが,ここはカタカナのASCIIコード[1]が割り当てられている部分だ。つまり,シフトJISコードの1バイト目はASCIIコー

ド00$_H$～7F$_H$の英数記号・コントロールコード，A0$_H$～DF$_H$のカナと重複しないように作られている。これにより，半角英数字・カナとシフトJISコードが混在しても，しっかり区別できるというわけだ[2]。

■1)日本のコンピュータで使われている1バイト系文字コードは，正確には〝ASCIIコードを拡張してJISで制定したコード〟というべきなのだが，以下ASCIIコードで通すことにする
■2)機種によっては，ASCIIコード80$_H$～9F$_H$やE0$_H$以降にセミグラフィック文字が割り当てられている関係で，漢字を使うと使用できない半角文字があったりするのだが，X68000ではもともと漢字の使用が前提であるため，こういった心配はしなくてすむ。

シフトJISコードとASCIIコードが混在した文字列は，先頭から1バイトずつ順に見ていって81$_H$～9F$_H$，E0$_H$～EF$_H$のコードが現れたら，その次の1バイトとあわせた2バイトが漢字を表すものと判断する。JISコードを使ったときのような漢字イン，漢字アウトが必要でない点に注目してほしい。

ここで，せっかくASCIIコードと区別できるようになってはいても，プログラム側で対応していなければ意味がない。一例を挙げると，DOSコールgetcではシフトJISコードは1バイトずつ2度に分けて入力されるから，これを元の2バイトコードに再構成するのはユーザープログラムの仕事だ。

UPPER.Xの第1版では，これを怠っていたために，たとえば〝敏〟(シフトJISコード9571$_H$)が入力されると，

1) 1バイト目の95$_H$を持ってくる。ASCIIコードの小文字の範囲外だから素通りさせる
2) 2バイト目の71$_H$を持ってくる。〝q〟のASCIIコードだから，大文字〝Q〟(ASCIIコード51$_H$)に変換する。

というように誤動作し，結果として，〝敏〟が〝媛〟(シフトJISコード9551$_H$)に化けてしまう。正しくは，1バイト目を取り出した時点で，シフトJISコードだということを調べ，2バイト目も素通りさせるように細工をしなければならない。

さて，実際にはX68000ではシフトJISコードを若干拡張して使っている。これらは他機種では通用しないX68000独自の文字コードであり，あれば便利なことは確かなのだが，プログラムを作るときには問題になることもある。

まず，EB9F$_H$～EC9E$_H$は(全角)外字のコードになっている。これはシフトJISコードの空き部分だから，プログラムでは他の全角文字と区別して扱う必要はない[3]。

■3)外字はUSKCGM.Xで作る。ちなみに，WP.Xでも同様の範囲が外字になっているが，WP.Xで作った外字はあくまでWP.X内だけで有効であり，システム側の外字とはまったく別の扱いになっている。僕は，これを「外字は使うな」というシステム設計者からのメッセージと受け取った!

80XX$_H$で表されるコードは，半角1文字を表す(XXの部分にはASCIIコードが入る)。たとえば，8041$_H$は半角の〝A〟を意味する。画面に表示した場合はASCIIコード41$_H$の〝A〟とまったく同じである。これだけならおもしろくもなんともないが，8001$_H$～801F$_H$は〝コントロールコードを実行せずに，文字として表示する〟のに利用できる(うまく説明しにくいのでASKの記号入力などを使って調べてみてほしい)。

また，80A0$_H$～80DF$_H$にはカタカナでなく，ひらがなが割り当てられているので，知っ

ていると役に立つこともあるだろう。

F0XX$_H$，F1XX$_H$には1/4角上付き文字が，F2XX$_H$，F3XX$_H$には同下付き文字が割り当てられている(〝^{2}〟とか〝$_A$〟のようなやつ)。F0XX$_H$とF1XX$_H$，F2XX$_H$とF3XX$_H$は，XXA0$_H$～XXDF$_H$がひらがなになるか，カタカナになるかの違いがある(F0XX$_H$，F2XX$_H$はカタカナ，F1XX$_H$，F3XX$_H$がひらがな)。

さらに，F4XX$_H$，F5XX$_H$には半角外字が割り当てられている。このうち，F400$_H$～F41F$_H$，F500$_H$～F51F$_H$は〝特殊半角外字〟とか名づけられているようで，マニュアルによれば，一般のアプリケーションからは使用できないことになっている。

なお，EC9F$_H$～EDE2$_H$のコードを画面に表示してみると，半角外字が2つ合成されたもの(単純にくっつけた形ではない)が表示される。これはうまくやれば使用できる全角外字の数を増やすことができることを意味する。

F6XX$_H$～FFXX$_H$は未使用だが，コード入力などを使って無理やり表示すると，半角スペースが表示されるようだ。これらはプログラム中では2バイト半角文字として扱うのが自然だろう。

以上が，X68000で使用できる文字のすべてだ。X68000上で動くプログラムを作るからには，これらの文字すべてが正しく扱えるように設計する必要がある。これは僕たちがプログラムを作るうえでいつも頭に入れておかなければならない事柄である。

しつこく動作試験してみる

さて，キーボードからの入力ですらこれだけ問題があったわけだが，フィルタであるからにはリダイレクションをまじえて実験してみる必要がある。次のようにして，標準入力を自分自身のソースファイルにリダイレクトして実行してみよう。キーボードからデータを入力した場合とはまた違った症状がいくつか見られると思う。

```
A>UPPER <UPPER.S
```

まず，改行は正しく行われている。正しく動作しているのはよいことだが，キーボードからデータ入力した場合と結果が違うのはやはりおかしい。また懸念であった〝無限ループになっていることの影響〟は，しっかり〝ファイルの最後まで処理した時点でそのまま黙り込んでしまう〟という症状になって現れる[2]。しかも黙り込む直前に画面がクリアされてしまうという不可解なオマケももれなくついてくる。

■ 2) 読者がHuman68kのver.2.0を使用しているのであれば，BREAKキーによって，この沈黙状態から抜けることができる。しかし，Human68kのver.1.0を使用している場合はなぜかBREAKキーが効かないので，INTERRUPTスイッチを押して強制終了してもらいたい。これはver.1.0のマイナーなバグだと思う。

ここまでの実験で，UPPER.Xの現バージョンにおける問題点がかなりはっきりしてきた。

1) BREAKキーを押さないと止まらない。キーボードからデータを入力する場合はともか

く，標準入力をリダイレクトした場合はファイルの最後まで処理したら停止してくれないと困る。

2) キーボードから入力する場合に改行動作が正しくない(標準入力をファイルにリダイレクトすると正しく改行する)。

3) 入力をファイルに切り替えた場合，ファイルの最後でなぜか画面がクリアされてしまう。

4) 全角文字の入力が考慮されていない。

次の節では，これらの問題点をいっきにクリアする。

改良する:UPPER.X第2版

まず，ファイルの最後まで処理したら，自動的に終了するようにしよう。そのためには，ファイルの最後を検出する方法を知らなくてはならない。

Human68kのテキストファイルは，一般に次のような構造をしている。

1) 各行は$0D_H$，$0A_H$で終わる。

2) ファイルの最後は$1A_H$という1バイトのコードで表す。

試しにUPPER.SをDUMP.Xで覗いてみると(図1)，このとおりの構造だということが確認できるだろう。

この新しい情報によって，プログラムをいつ終了させたらよいかがわかる。つまり，$1A_H$が入力されたら，実行を終えるようにすればよい。これはgetcにより入力された1バイトデータを$1A_H$と比較して処理を振り分けるような数行の追加で実現できるだろう。この変更により，データをキーボードから入力するときも，CTRL+Zを押せば終了するようになる。

ここで，ついでに$1A_H$というコードについてもう少し調べてみよう。『Human68kユーザーズマニュアル』のコントロールコード一覧を見てみると，$1A_H$は画面クリアのコントロールコードになっている。さきほどの実験でファイルの最後まで処理した時点で画面が消去されたのは，このファイルエンドコード兼画面消去コードの$1A_H$がフィルタを素通りし画面に出力されてしまったためだった。入力が$1A_H$だったら，即座にプログラムを終了する($1A_H$は出力しない)

```
00000000  2A 09 89 70 8F AC 95 B6 8E 9A 81 A8 89 70 91 E5 *.英小文字→英大
00000010  95 B6 8E 9A 95 CF 8A B7 83 74 83 42 83 8B 83 5E 文字変換フィルタ
00000020  81 40 91 E6 82 50 94 C5 0D 0A 0D 0A 09 2E 69 6E   第1版.....in
00000030  63 6C 75 64 65 09 64 6F 73 63 61 6C 6C 2E 6D 61 clude.doscall.ma
                                  :
                                  :
00000210  0A 6D 79 73 74 61 63 6B 3A 0D 0A 09 2E 64 73 2E .mystack:....ds.
00000220  6C 09 32 35 36 09 09 2A 83 58 83 5E 83 62 83 4E l.256..*スタック
00000230  97 CC 88 E6 0D 0A 6D 79 73 70 3A 0D 0A 09 2E 65 領域..mysp:....e
00000240  6E 64 0D 0A 1A                                 nd...
```

図1

ようにすれば，謎の画面クリアの問題も片づくことになる。

次に改行動作の不備を直す。入力の$0D_H$を特別扱いするわけだが，単純に$0A_H$を付け加えるのではうまくない。というのは，入力をファイルに切り替えているときには，各行の終わりの$0D_H$，$0A_H$が，$0D_H$，$0A_H$，$0A_H$に変換されてしまうことになるからだ。

入力がキーボードかファイルかを調べることができれば処理を振り分けることができるし，実際そうすることも可能なのだが，ここではもっとかんたんな方法を採ることにした。

1) $0D_H$が入力されたら，$0D_H$，$0A_H$の2バイトに変換して出力する。
2) $0A_H$が入力されたら，捨てる(出力しない)。

これにより，$0D_H$が単独で現れた場合にも，$0D_H$，$0A_H$が連続して現れた場合にも正しく対応できるようになる。ファイルの途中にぽつんと$0A_H$が入っていたりする可能性がないとはいえないから，これは100%の方法ではないが，実用上の問題はないという判断だ。

最後に，全角文字を考慮して英小文字→大文字変換の部分を練り直す。いままでは文字を単純に1バイトずつ処理していたために，2バイトを組みにして扱うべきシフトJISコードが不当に化けてしまっていたわけだから，入力された文字がシフトJISコードかどうかを調べ，そうであれば素通りさせるような処理を付け加えればよいだろう。手順は次のようになる。

1) 1バイト持ってくる。
2) それがシフトJISコードの1バイト目でなければ，英小文字→大文字変換してから結果を出力し，1)へ戻る。
3) シフトJISコードの1バイト目であれば，無条件に素通りさせる(変換はいっさい行わない)。続いてもう1バイト持ってきて，これも素通りさせる。それから1)へ戻る。

こうして，以上の発見されたかぎりの不備を直したUPPER.Xの修正版がリスト3だ。13～14行がファイルの終端かどうかを調べる部分，16行あたりが改行部分の処理になっている。また，22～27行が全角文字の1バイト目かどうかのチェックで，ここでやっていることは，

1) 80_Hより小さいコードは半角文字
2) 80_H～$9F_H$ならば，シフトJISコードの1バイト目
3) $A0_H$～DF_Hならば半角(カタカナ)
4) $E0_H$以上ならシフトJISコードの1バイト目

という一連の比較と処理の振り分けだ。本来，シフトJISコードの1バイト目は81_H～$9F_H$または$E0_H$～EF_Hの範囲だから，2)と4)のチェックは正しくないように見えるが，X68000では8000_H台と$F000_H$以降の2バイトコードで各種の半角文字を表すことになっているのでこれでよいのだ。

リスト3
UPPER.S(その2)

```
1: *       英小文字→英大文字変換フィルタ　第２版
2:
3:         .include        doscall.mac
4: *
5:         .text
```

```
 6:          .even
 7: *
 8: ent:
 9:          lea.l   mysp,sp         *spの初期化
10:
11: loop:    DOS     _GETC           *１文字入力
12:
13:          cmpi.b  #$1a,d0         *ファイルエンドコードか？
14:          beq     done            *そうなら終了
15:
16:          cmpi.b  #$0a,d0         *LFコードか？
17:          beq     loop            *そうなら無視
18:
19:          cmpi.b  #$0d,d0         *CRコードか？
20:          beq     cr_lf           *そうならLF,CRにして出力
21:
22:          cmpi.b  #$80,d0         *80Hより小さければ
23:          bcs     hankaku         *       ASCIIコード
24:          cmpi.b  #$a0,d0         *80H以上A0H未満なら
25:          bcs     zenkaku         *       シフトJISの１バイト目
26:          cmpi.b  #$e0,d0         *A0H以上E0H未満なら
27:          bcs     hankaku         *       ASCIIカタカナ
28:                                  *E0H以上なら
29:                                  *       シフトJISの１バイト目
30: *
31: zenkaku:
32:          move.w  d0,-(sp)        *シフトJISの１バイト目を
33:          DOS     _PUTCHAR        *       そのまま出力
34:          addq.l  #2,sp           *
35:
36:          DOS     _GETC           *もう１バイト持ってくる
37:          move.w  d0,-(sp)        *シフトJISの２バイト目も
38:          DOS     _PUTCHAR        *       そのまま出力
39:          addq.l  #2,sp           *
40:
41:          bra     loop            *繰り返す
42: *
43: hankaku:
44:          bsr     toupper         *小文字→大文字変換
45:
46:          move.w  d0,-(sp)        *１文字出力
47:          DOS     _PUTCHAR        *
48:          addq.l  #2,sp           *
49:
50:          bra     loop            *繰り返す
51: *
52: cr_lf:
53:          move.w  d0,-(sp)        *d0にはCRコードが入っている
54:          DOS     _PUTCHAR        *CRコードを出力
55:          move.w  #$0a,(sp)       *LFコードを出力
56:          DOS     _PUTCHAR        *
57:          addq.l  #2,sp           *
58:
59:          bra     loop            *繰り返す
60: *
61: done:
62:          DOS     _EXIT           *終了
```

```
63: *
64: *英小文字→英大文字変換サブルーチン
65: toupper:
66:         cmpi.b  #'a',d0         *英小文字か？
67:         bcs     toupr0          *
68:         cmpi.b  #'z'+1,d0       *
69:         bcc     toupr0          *
70:         subi.b  #$20,d0         *小文字なら大文字に変換
71: toupr0: rts                     *サブルーチンからリターン
72: *
73:         .stack
74:         .even
75: *
76: mystack:
77:         .ds.l   256             *スタック領域
78: mysp:
79:         .end
```

プログラムの各部の働きがわかったら，またさきほどのようにあれこれと実験してみよう。キーボードから入力中にリターンキーを押したときの動作は正しいか，標準入力をファイルにリダイレクトしたときにはファイルの最後でちゃんと止まるか，全角文字が化けたりしないか，ひとつひとつ確認しておくこと。

さらに改良する:UPPER.X第3版

さきほどHuman68kのテキストファイルは$1A_H$で終わるといった。しかし，じつはこのファイルエンドコードはなくてもよいことになっている。試しに，

```
A>MORE <UPPER.S >TEST.S
```

を実行してもらいたい。TEST.SとUPPER.Sの内容はまったく変わらないが(TYPEしてみよう)，DIRコマンドでディレクトリをとってみると，TEST.Sのほうが1バイト短いことがわかるはずだ。DUMP.Xでダンプしてみれば，UPPER.Sにはあったファイルエンドコード$1A_H$がTEST.Sにはないことも確認できるだろう。ファイル末の$1A_H$が削られてしまうのは，MORE.X[3]にかぎらずフィルタの一般的な副作用だ。

■ 3) MORE.Xは1画面ごとに表示を一時停止する拡張TYPEコマンドのようなプログラムだが，このように標準入出力ともにリダイレクトしたときには〝何もしないフィルタ〟として動作する。いまの場合は，UPPER.SはTEST.Sにそのままコピーされる。

TEST.Sは，当然エディタで読み込んで編集することができるし，AS.Xでアセンブルすることもできる。このことはファイルエンドコードがついていなくてもテキストファイルとして成り立つということの証拠である。

さて，いまのところ，UPPER.Xはファイルエンドコードのみに頼ってファイルの終端を調べている。このため，上の話にあった〝エンドコードのないテキストファイル〟はきちんと処

理できない[4]。

■ 4) じつはHuman68kではテキストファイルとそうでないファイル(バイナリファイル)を内部ではまったく区別していない。あくまでユーザーが勝手に区別しているにすぎない。ところで，エンドコードがないとすると，どうやってファイルの終わりを表しているのかという疑問が生じるが，これは単純なことで，ディスクにはファイルの本体以外にファイルの長さなどの情報が別に格納してあるのだ。ほら，DIRでファイルサイズが表示されるじゃないか。

```
A>UPPER <TEST.S
```

を実行すると，予想どおりファイルの最後まで処理した時点で黙り込んでしまう。

そこで，UPPER.Xをさらにリスト4のように修正する。初登場のDOSコールfgetcを使って標準入力からの1文字入力を行っている。詳しい話は後でするが，15行からの

```
clr.w     -(sp)
DOS       _FGETC
addq.l    #2, sp
```

は，ほぼ

```
DOS       _GETC
```

と同等の処理を行うものだ。大きな違いは，getcがまったくエラーを返さないのに対して，fgetcのほうはエラーであればd0.lに負の数を入れて戻るという点だ。ファイルが終わっているのにさらにデータを読み込もうしたときに発生するエラーを検出し，その時点でプログラムを終了しようという姑息な手である。エラーが生じたかどうかを調べているのが19行の

```
tst.l  d0
```

であり，tstの実行によりd0.lが負であればNフラグが立つ(Nフラグが1になる)から，

```
bmi   ～
```

より処理を振り分けている[5]。

■ 5) このbmiはbeqやbneの親戚で，その意味は〝演算結果が負(＝Nフラグが1)であれば分岐〟である。ついでにいっておくと，〝演算結果が正(Nフラグが0)のとき分岐〟させたければ，

```
bpl   ～
```

を使う。それぞれはMInusとPLusの略になっている。

リスト4
UPPER.S(その3)

```
 1: *       英小文字→英大文字変換フィルタ　第3版
 2:
 3:         .include        doscall.mac
 4: *
 5: CR      equ     $0d
 6: LF      equ     $0a
 7: EOF     equ     $1a
 8: *
 9:         .text
10:         .even
```

```
11: *
12: ent:
13:             lea.l   mysp,sp         *spの初期化
14:
15: loop:       clr.w   -(sp)           *１文字入力
16:             DOS     _FGETC          *
17:             addq.l  #2,sp           *
18:
19:             tst.l   d0              *エラーか？
20:             bmi     done            *そうなら終了
21:
22:             cmpi.b  #EOF,d0         *ファイルエンドコードか？
23:             beq     done            *そうなら終了
24:
25:             cmpi.b  #LF,d0          *LFコードか？
26:             beq     loop            *そうなら無視
27:
28:             cmpi.b  #CR,d0          *CRコードか？
29:             beq     cr_lf           *そうならLF,CRにして出力
30:
31:             cmpi.b  #$80,d0         *80Hより小さければ
32:             bcs     hankaku         *       ASCIIコード
33:             cmpi.b  #$a0,d0         *80H以上A0H未満なら
34:             bcs     zenkaku         *       シフトJISの１バイト目
35:             cmpi.b  #$e0,d0         *A0H以上E0H未満なら
36:             bcs     hankaku         *       ASCIIカタカナ
37:                                     *E0H以上なら
38:                                     *       シフトJISの１バイト目
39: *
40: zenkaku:
41:             move.w  d0,-(sp)        *シフトJISの１バイト目を
42:             DOS     _PUTCHAR        *       そのまま出力
43:             addq.l  #2,sp           *
44:
45:             DOS     _GETC           *もう１バイト持ってくる
46:             move.w  d0,-(sp)        *シフトJISの２バイト目も
47:             DOS     _PUTCHAR        *       そのまま出力
48:             addq.l  #2,sp           *
49:
50:             bra     loop            *繰り返す
51: *
52: hankaku:
53:             bsr     toupper         *小文字→大文字変換
54:
55:             move.w  d0,-(sp)        *１文字出力
56:             DOS     _PUTCHAR        *
57:             addq.l  #2,sp           *
58:
59:             bra     loop            *繰り返す
60: *
61: cr_lf:
62:             move.w  d0,-(sp)        *d0にはCRコードが入っている
63:             DOS     _PUTCHAR        *CRコードを出力
64:             move.w  #LF,(sp)        *LFコードを出力
65:             DOS     _PUTCHAR        *
66:             addq.l  #2,sp           *
```

```
67:
68:         bra     loop            *繰り返す
69: *
70: done:
71:         DOS     _EXIT           *終了
72: *
73: *英小文字→英大文字変換サブルーチン
74: toupper:
75:         cmpi.b  #'a',d0         *英小文字か？
76:         bcs     toupr0          *
77:         cmpi.b  #'z'+1,d0       *
78:         bcc     toupr0          *
79:         subi.b  #$20,d0         *小文字なら大文字に変換
80: toupr0: rts                     *サブルーチンからリターン
81: *
82:         .stack
83:         .even
84: *
85: mystack:
86:         .ds.l   256             *スタック領域
87: mysp:
88:         .end
```

厳密にはこのチェックは不十分であり，ファイルエンドに達したとき以外のエラーも同列に扱ってしまっているのはよいことではない。いわば，応急手当てといったところだ。

それと，いままで数字で記述していた改行コードの類が5～7行で定義したラベルには置き換えられている点に目を止めてもらいたい。このようにプログラム中で使う定数を，意味を持った記号で書き表すようにすれば，プログラムの読みやすさが向上し，また，ミスタイプの危険も少なくすることができる。

第4版に向けて

UPPER.Xは，2段階の修正を経て，だいぶまともなフィルタになってきた。しかし，まだ完全とはいえず，バグと呼んで差し支えないような不備が残っている。

十分長いテキストファイルを用意し，

```
A>UPPER <FILE
```

を実行してみると，たぶん，気持ちの悪い症状が見られるだろう(出力結果が読めるか？)。また，リスト5に示すX-BASICプログラムを実行すると，TESTというファイル名の短いファイルが生成されるから，TYPE, DUMPで内容を確認したうえでUPPER.Xの第1～3版それぞれに対して，

```
A>UPPER <TEST
```

を実行してもらいたい。そして，この2つの実験からバグの原因を想像してみよう。

リスト5
テストプログラム
(X-BASIC用)

```
10 /*UPPER.Xテスト用プログラム
20 int fp
30 dim char dat(9) = {
40     'a','b','c',3,'d','e','f',13,10,26
50 }
60 /*
70     fp = fopen( "test","c" )
80     fwrite( dat, 10, fp )
90     fclose( fp )
```

さらに，実行速度の問題がある。現バージョンのUPPER.Xはマシン語で書かれているにもかかわらず，遅いのだ。XCを持っている人は，リスト6のUPPER.Xと同等な処理を行う(ただし，全角文字の入力は考慮していない)Cプログラム〝C_UPPER.C〟をコンパイルし，

A>UPPER <UPPER.S >TEST1

A>C_UPPER <UPPER.S >TEST2

をそれぞれ実行して両者の速度を比較してもらいたい。この結果は少々ショックである。

リスト6
C_UPPER.C
(XC用)

```
 1: #include    <stdio.h>
 2: #include    <ctype.h>
 3:
 4: int main()
 5: {
 6:     int c;
 7:
 8:     while ( EOF != ( c = getchar() ) )
 9:         putchar( toupper( c ) );
10:
11:     return 0;
12: }
```

それに，人間工学的(?)に見て，このプログラムをもっと使いやすくできないかどうかを一度検討してみる必要がある。たとえば，フィルタはほとんどの場合，リダイレクトを利用して使うものだから，わざわざ

A>UPPER <UPPER.S

と打ち込むのではなく，

A>UPPER UPPER.S

のように直接ファイル名を指定できるようにするべきではないだろうか。

さっそく改良に取りかかりたいところだが，その前に標準入出力を使わない一般的なファイルの取り扱いについて話しておきたい。じつは，これはフィルタを作るうえでも重要なことなのである。

ちょっと長くなるが一息に説明する。

DOSコールを使ったファイル処理

Human68kのDOSコールを使ってファイルを操作する場合の手順は次のようになる。

1) ファイルを開く(オープンする)
2) 読み書きする
3) ファイルを閉じる(クローズする)

それぞれの処理は対応するDOSコールを呼び出すことで行う。基本的にはCやX-BASICに用意されているファイル処理関数が,DOSコールの呼び出しに置き換えられたような感じといえる。X-BASICでファイルの入出力ルーチンが書ける人であればすぐに慣れるだろう。

ファイルをオープンする

これはファイルを読んだり書いたりすることができるようにするための準備操作で,DOSコール$FF3Dのopenまたは$FF3Cのcreateを使って行う。openは,次のようにして使用する。

```
move.w    アクセスモード,-(sp)
move.l    ファイル名,-(sp)
DOS       _OPEN
addq.l    #6,sp
```

スタックに積むパラメータがワードデータ1つとロングワードデータ1つの計6バイトなので,呼び出し後のスタック補正も6バイト分になる。

アクセスモードは,ファイルを読み込み用にオープンするか,書き込み用にオープンするかといった指定で,以下のような数値で指定する。

0……読み込み用
1……書き込み用
2……読み書き両用

この他に特殊なものとして日本語FEPの辞書専用のモードがあるが,ユーザープログラムから使用する意味はあまりないのでここでは無視する。

ファイル名は,00_Hで終わる形式の文字列としてメモリ上に用意し,その先頭アドレスをスタックに積んで指定する。なお,読み込み用のときはもちろん,他のモードのときでも指定したファイルはすでにディスク上に存在していなければならず,見つからない場合はエラーになる。openの書き込み用モード,読み書き両用モードは〝すでにあるファイルを上書きして更新する〟ために用意されているモードだ。

たとえば〝ABCDEFG〟という1行からなるファイルをDOSコールopenを使って書き込み用にオープンしてから〝123〟を出力すると,ファイルの内容は〝123DEFG〟になる。

どちらにしろ，DOSコールopenは主として読み込み用に使われる。ファイル名を表す文字列がラベルFILENAME以下に用意されているとすると，読み込みモードでのopenの典型的な呼び出し方は次のようになる。

```
clr.w       -(sp)
pea.l       FILENAME
DOS         _OPEN
addq.l      #6, sp
```

1行目は，

```
move.w      #0, -(sp)
```

でもよいし，2行目も

```
move.l      #FILENAME, -(sp)
```

でかまわないのだが，clr，peaを使ったほうがスマートだ。

ファイルを新規に作成する(同時に書き込み用にオープンする)場合はcreateを使う。呼び出し方は次のとおり。

```
move.w      ファイル属性, -(sp)
move.l      ファイル名, -(sp)
DOS         _CREATE
addq.l      #6_sp
```

ファイル属性は，そのファイルがどのような種類・性格のものであるかを表し，ファイル名やファイルサイズなどの情報といっしょにディスク上に(ファイル本体とは別に)記録される。ファイル属性は図2のようなビットフィールドになっている(ビット単位に意味を持つ)。ごくふつうのファイルを作成する場合は第5ビットだけを1にした値である0020_Hを指定すればよい。よってcreateのスタンダードな使い方は，

```
move.w      #$0020, -(sp)
pea.l       FILENAME
DOS         _CREATE
```

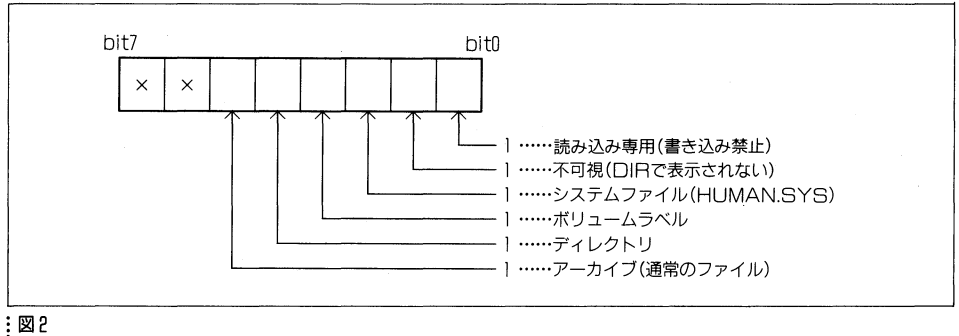


図2
ファイル属性

```
addq.l      #6, sp
```

のようになる。

createはファイルを新規に作成するわけだから，指定したファイルはディスク上になくてもよい。逆に，すでに存在するファイルを指定した場合は，そのファイルが新規作成されたときの状態になるように(ファイルサイズを0に切り詰める＝そのファイル内容は失われる)書き込み用にオープンする。これにより，プログラム側で新規にファイルを作成する場合と，古いファイルの使い回しをする場合とを区別する必要がなくなっている。ファイルを書き込み用にオープンする場合には，openよりもcreateのほうが好んで使用されるのはこのためだ。

open，createともに，エラーが発生した場合はd0.1にDOSのエラー番号を返す。エラー番号はつねに(2の補数表現で)負の値をとる。一般にHuman68kのDOSコールは，エラーが発生したらd0.1に負の数を返す約束になっており，DOSコールから戻った時点でのd0.1の符号を調べるだけで，エラーがあったかどうかを判別できる。具体的には，tst命令[6]を使ってd0.1の符号をNフラグに反映させてから，適当な条件分岐命令を用いればよい。エラーのときにエラー処理ルーチンに分岐させたければ，

```
DOSコールの呼び出し
tst.l    d0
bmi      エラー処理ルーチンへ
正常時の処理～
```

のようにプログラムを組む。

■ 6) tstは0と比較する命令であるが，cmpとsubとの関係に照らしあわせれば，〝0と比較する〟のは〝0を引いた結果に応じてフラグを変化させる〟ことに等しい。0を引いてみて結果が負であれば，元の数が負であったことがわかるわけである。

なお，実際のプログラムでは〝エラーかどうか〟がわかれば十分である場合のほうが多いが，その気になればエラー番号からどんなエラーが生じたかを判別することもできる。気になる人は『プログラマーズマニュアル』のエラーコード一覧を参考にされたい。

さて，open，createではエラーがなくファイルがオープンできたら，d0.1に〝ファイルハンドル〟と呼ばれる正の値を返す。より正確にはd0.1の上位ワードは0であり，ファイルハンドルとしての意味を持つのは下位ワードだけである。ファイルハンドルは，以後ファイルを読み書きする際に識別番号として使われる。これは，X-BASICの関数fopenが返すファイル番号と同等のものと考えてよい。

ここで，d0.wに返されたファイルハンドルはすかさず余っているデータレジスタないしメモリ上に確保したワークエリア[7]に待避しておく必要がある。d0レジスタは他のDOSコールでも値を返すのに用いられるので，放っておけば次にDOSコールを実行した時点で破壊されて(レジスタの値が変わって)しまうからだ。識別番号がわからなくなれば，以後の読み書きが行えなくなるのは説明の必要もないくらい当然のことである。

■ 7)ワークエリア(work area):作業用のメモリ領域のこと。たんにワークともいう。

読み書きする

オープンに成功すれば，そのファイルはファイルハンドルを介して読み書きできるようになる。ファイルの読み書きは，それぞれ該当するDOSコールを呼び出すことで行う。うすうす気づいているように，X-BASICやCに見られる1バイト単位の入出力，行単位の入出力，任意サイズ単位の入出力などが別々のDOSコールとして用意されている。使い方もX-BASICやCと大差はない。ここでは，基本である1バイト単位の入出力を行うDOSコール$FF1Bのfgetcと$FF1Dのfputcだけを紹介しておく。

fgetcは，次のようにして使用する。

```
move.w      ファイルハンドル,－(sp)
DOS         _FGETC
addq.l      #2,sp
```

ファイルから読み込んだデータはd0.lに返されるが，1バイト単位の読み込みだから意味を持つのは最下位8ビットであるd0.bだけになる。エラーが生じたときはd0.lに負のエラー番号を返す。

fgetcと対になるfputcは，次のようにして使う。

```
move.w      ファイルハンドル,－(sp)
move.w      出力データ,－(sp)
DOS         _FPUTC
addq.l      #4,sp
```

出力データはワードで積むが，意味があるのは下位バイトだけだ。エラーが発生した場合はd0.lにエラー番号を返す。正常に書き込みが行われたときは，DOSコールから戻った時点でのd0.lは正の値であることが保証されている。

ファイルをクローズする

読み書きが終了したらファイルをクローズし，現在使用しているファイルハンドルを解放して他のファイルで使用できるようにする。ファイルハンドルの数にはかぎりがあるから，これを怠るとファイルハンドルが足りなくなることもありうる[8]。

■8) 同時にオープンできるファイルの最大数は，CONFIG.SYSの"FILES＝～"で指定した数で決まる。

ファイルをクローズするDOSコール$FF3Eのcloseは，以下のようにして使用する。

```
move.w    ファイルハンドル,－(sp)
DOS       _CLOSE
addq.l    #2,sp
```

めったにないことだが，正常にクローズできなかった場合はd0.lにエラー番号を返す。

closeはファイルを個別にクローズするが，オープンした全ファイルをまとめてクローズしたければ＄FF1Fのallcloseを使う。このDOSコールはパラメータを持たないので，たんに

```
DOS    _ALLCLOSE
```

として呼び出す。ただし，allcloseはエラーを返さないので，正常にクローズできたことを確認する必要がある場合は，面倒でもcloseを使って個別に閉じてやる必要がある。

なお，Human68kではexitやexit2でプログラムを終了するときに，自動的にallcloseが呼び出されて全ファイルが閉じられるようになっている。これによって，あやまってファイルをクローズし忘れてもディスクが破壊されるといった心配はないわけだ。しかし，だからといってファイルをクローズせずに終了するようなプログラムはかっこよくない(この言葉につきる)ので，ファイルを開いたら必ず閉じる習慣を身につけたい。

標準入出力との関係

前節で，

```
clr.w     －(sp)
DOS       _FGETC
addq.l    #2,sp
```

により標準入力からの1バイト入力を行ってみせた。1行目は

```
move.w    #0,－(sp)
```

と同じ意味だから，fgetcの呼び出し方と見比べれば，0というファイルハンドルが標準入力を指定するのに使えることがわかる。同様に，ファイルハンドル1は標準出力を意味する。

```
move.w    #1,－(sp)
move.w    #'A',－(sp)
DOS       _FPUTC
addq.l    #4,sp
```

は，

```
move.w    #'A',－(sp)
DOS       _PUTCHAR
addq.l    #2,sp
```

とほぼ同等の動作をするわけだ[9]。

■9) putcharがエラーをまったく返さないのに対して，fputcはエラーがあったらエラー番号を返すといった細かな違いはある。

つまり，いままで扱ってきた標準入出力による入出力は，Human68kにおけるファイルの扱いのうちの限定された一部のケースであったと考えられる。

ついでながら，ファイルハンドルの2番は〝標準エラー出力〟，3は〝標準補助装置〟，4は〝標準プリンタ〟を意味する。標準エラー出力は出力用で，通常はデバイスCON（ディスプレイ）に割り当てられている。標準エラー出力はコマンドモードからリダイレクトする方法が用意されていないから，リダイレクトされては困るような各種メッセージを出力するのに利用する。

標準補助装置はデバイスAUX(RS-232C)に割り当てられており，入出力ともに行える。RS-232Cを介して他のマシンや周辺機器とデータをやりとりするのに使用する。また，標準プリンタはデバイスPRNにあたり，このファイルハンドルを使うことでプリンタへの出力が行える。

これら0～4のファイルハンドルは，プログラムが起動した時点ですでにオープンされているので，あらためてオープンする必要はなく，いきなり読み書きを行うことができる。

サンプル

リスト7のCOPYTEST.Xは，いま紹介したDOSコールを組み合わせて作ったファイル複写プログラムだ。コピーするファイル名はプログラム中に埋め込まれているので，サンプル以外には何の使い道もない間の抜けたプログラムである。FILE1というファイルをFILE2というファイル名でコピーする。

リスト7
COPYTEST.S
(その1)

```
 1: *       ファイルコピーサンプル
 2:
 3:         .include        doscall.mac
 4: *
 5:         .text
 6:         .even
 7: *
 8: ent:
 9:         lea.l   mysp,sp         *spの初期化
10:
11:         clr.w   -(sp)           *入力先ファイルを
12:         pea.l   sour            *  読み込み用にオープン
13:         DOS     _OPEN           *
14:         addq.l  #6,sp           *
15:
16:         tst.l   d0              *エラー？
17:         bmi     error           *  そうならエラー終了
18:
19:         move.w  d0,d1           *d1.w=入力先ファイルハンドル
20:
21:         move.w  #$0020,-(sp)    *出力先ファイルを新規作成
22:         pea.l   dest            *
23:         DOS     _CREATE         *
24:         addq.l  #6,sp           *
25:
26:         tst.l   d0              *エラー？
```

```
27:          bmi     error           *  そうならエラー終了
28:
29:          move.w  d0,d2           *d2.w=出力先ファイルハンドル
30:
31: loop:
32:          move.w  d1,-(sp)        *入力先ファイルハンドルから
33:          DOS     _FGETC          *  1バイト読み込む
34:          addq.l  #2,sp           *
35:
36:          tst.l   d0              *エラー?
37:          bmi     done            *  そうならファイルエンドと見なす
38:
39:          move.w  d2,-(sp)        *出力先ファイルハンドルへ
40:          move.w  d0,-(sp)        *  1バイト書き出す
41:          DOS     _FPUTC          *
42:          addq.l  #4,sp           *
43:
44:          tst.l   d0              *エラー?
45:          bmi     error           *  そうならエラー終了
46:
47:          bra     loop            *繰り返す
48:
49: done:    DOS     _ALLCLOSE       *ファイルを閉じる
50:          DOS     _EXIT           *終了
51:
52: *
53: error:                   *エラー処理
54:          pea.l   errmes          *エラーメッセージを表示
55:          DOS     _PRINT          *
56:          addq.l  #4,sp           *
57:
58:          move.w  #1,-(sp)        *終了コード1を持って
59:          DOS     _EXIT2          *  終了
60:
61: *
62: sour:    .dc.b   'FILE1',0       *転送元ファイル名
63: dest:    .dc.b   'FILE2',0       *転送先ファイル名
64: errmes:  .dc.b   'エラーです'    *エラーメッセージ
65:          .dc.b   $0d,$0a,0       *
66: *
67:          .stack
68:          .even
69: *
70: mystack:
71:          .ds.l   256             *スタック領域
72: mysp:
73:          .end
```

11～19行で転送元ファイルを読み込み用に，21～29行で転送先ファイルを新しく書き込み用に作成している。どちらの場合も正しくオープンできなかった場合は53行に飛び，エラーメッセージを出して終了コード1を持ってエラー終了する。

31～47行が実際にファイルコピーを行うループになっている。fgetcで1バイト読み込んではfputcで書き出すだけである。書き込み時にエラーが生じた場合は，さきほどと同じエラー処

理ルーチンに分岐する。また，読み込み時にエラーが発生したら，それ以上ファイルがないものと判断してループを脱出している。読み込み時にはこの他のエラーが発生する可能性があるから，本来ならエラー番号を解析してファイルの終わりなのか，それとも何か他のエラーなのかを判別する必要があるのだが，ここでは手を抜いている。

転送がすんだら，49行で安直にallcloseを使ってオープンした2つのファイルをまとめてクローズして実行を終える。

では，適当な長さのファイルをFILE1という名前で用意して，このプログラムを走らせてみて正しくコピーできているかどうかを確認してもらいたい。目で見比べてもよいのだが，ファイル比較プログラムFC.Xを使えばよりかんたん・確実である。

```
A>FC  FILE1  FILE2  /B
```

ところで，COPYTEST.Xの実行速度はCOMMAND.XのCOPYコマンドに比べると数段遅い。フロッピーディスクよりもRAMディスクなどの高速なメディア上で実行すると，その差ははっきりとしてくる。理屈のうえでは実際にディスクが回っている(?)時間はCOPYでもCOPYTEST.Xでも変わらないはずだから，COPYTEST.Xはその他の余計な処理に手間取っていると考えられる。具体的に挙げれば，DOSコールの呼び出し部分である。

COPYTEST.Xはループの中でちまちま1バイトずつ入出力しており，そのたびにDOSコールを2度ずつ実行する。このDOSと行ったり来たりするのにかかる時間はミクロに見ればたいしたことはないのだが，このプログラムのようにループの中で何度もDOSコールを実行すると，積もり積もって馬鹿にならない大きさになる。この場合，1バイト単位ではなく，もう少しまとまった単位で入出力を行うDOSコールを利用し，DOSコールの呼び出し回数を減らすべきである。

サンプルの改良

リスト7を高速化したのがリスト8である。ファイルをオープンするところまではリスト1とまったく同じだ。fgetc，fputcで1バイトずつ入出力を行うかわりに，read，writeというDOSコールを使って1024バイト単位で転送している。この1024という値はHuman68k内部でディスクの読み書きを行うときの単位であり，プログラム側で同じサイズに揃えたのは，DOSによけいな負担をかけないようにするためである。

リスト8
COPYTEST.S
(その2)

```
 1: *       ファイルコピーサンプル  その2

31: loop:
32:         move.l  #1024,-(sp)     *1024バイト読み込む
33:         pea.l   buffer          *
34:         move.w  d1,-(sp)        *
35:         DOS     _READ           *
36:         lea.l   10(sp),sp       *
37:
```

```
 38:          tst.l   d0              *d0.lが
 39:          bmi     error           *  負ならエラー
 40:          beq     done            *  0ならファイルの終わり
 41:                                  *  それ以外なら読み込んだバイト数
 42:          move.l  d0,d3           *d3.l = 読み込んだバイト数
 43:
 44:
 45:          move.l  d3,-(sp)        *読み込んだ分だけ書き出す
 46:          pea.l   buffer          *
 47:          move.w  d2,-(sp)        *
 48:          DOS     _WRITE          *
 49:          lea.l   10(sp),sp       *
 50:
 51:          tst.l   d0              *エラー?
 52:          bmi     error           *  そうならエラー終了
 53:                                  *d0.l = 実際に書き出したバイト数
 54:          cmp.l   d3,d0           *実際に指定しただけ書き出せたか?
 55:          bcs     error           *  足りなければエラー
 56:
 57:          bra     loop            *繰り返す
 58:
 59: done:    DOS     _ALLCLOSE       *ファイルを閉じる
 60:          DOS     _EXIT           *終了
 61:
 62: *
 63: error:                   *エラー処理
 64:          pea.l   errmes          *エラーメッセージを表示
 65:          DOS     _PRINT          *
 66:          addq.l  #4,sp           *
 67:
 68:          move.w  #1,-(sp)        *終了コード1を持って
 69:          DOS     _EXIT2          *  終了
 70:
 71: *
 72: sour:    .dc.b   'FILE1',0       *転送元ファイル名
 73: dest:    .dc.b   'FILE2',0       *転送先ファイル名
 74: errmes:  .dc.b   'エラーです'    *エラーメッセージ
 75:          .dc.b   $0d,$0a,0       *
 76: *
 77: buffer:  .ds.b   1024            *読み込み用バッファ1024バイト分
 78: *
 79:          .stack
 80:          .even
 81: *
 82: mystack:
 83:          .ds.l   256             *スタック領域
 84: mysp:
 85:          .end
```

read, writeの呼び出し方はリストを参照してもらえばわかると思うが, まとめておくと次のようになる[10]。

move.l　読み込みバイト数, −(sp)

move.l　読み込みバッファアドレス, −(sp)

```
move.w    ファイルハンドル,－(sp)
DOS       _READ
lea.l     10(sp),sp

move.l    書き出しバイト数,－(sp)
move.l    書き出しデータアドレス,－(sp)
move.w    ファイルハンドル,－(sp)
DOS       _WRITE
lea.l     10(sp),sp
```

■10) DOSコール呼び出し後の

```
lea.l  10(sp),sp
```

というのは見慣れない形だが，これは10バイト分のスタック補正を行う場合の常套手段である。〝d(aX)〟(spはa7と同じものである)の形式は，〝ディスプレースメント付きアドレスレジスタ間接アドレッシング〟と呼ばれるもので，〟アドレスレジスタにdを足したアドレスのデータ〟を意味する。たとえば，〝2(a0)〟は〝a0+2のアドレスで指定されるメモリ〟の意味になる。また，〝0(a0)〟は〝(a0)〟と同じものを指す。

ここで，leaはデータのアドレスを扱う命令だから，

```
lea.l  10(sp),sp
```

は〝spに10を足したアドレスにあるデータのアドレスをspに入れる〟＝〝spに10を足したアドレスをspに入れる〟ことになり，結果としてspに10を足したのと同じことになる。この表現はaddq.lよりは遅く，コードサイズも大きいが，adda.lよりは速くて短い。addqでは8バイトまでの加算しか行えないから，それ以上のスタック補正にはセカンドベストである

```
lea.l  10(sp),sp
```

の形式が使われるわけだ。定石として覚えておいてもらいたい。

readは，エラーがなければ実際に読み込んだバイト数をd0.lに返す。この値が呼び出し時に指定したバイト数より小さければ，ファイルが途中で終わったことがわかる。0だったら読むべきものがなにもなかった(＝すでにファイルの終わりに達している)ことを表している。これを利用して，リスト8では読み込んだバイト数が0の場合は実行を終える。

また，writeはエラーがなければ実際に書き出したバイト数を返す。この値が呼び出し時に指定した数より小さければ，ディスクがいっぱいでもうこれ以上データを書き出せないことを意味する。リスト8ではこれもエラーとみなし，54行でこのチェックを行って引っかかればエラー処理に分岐している。

フィルタの高速化を目指す

さて，話はようやくフィルタに戻る。現バージョンのUPPER.X(第3版)は，これまでに行ったいくつかの実験の結果，次のような不備が発見されている。

1) 入力がファイル，出力が画面のときに^Cによる実行中止，^Sによる一時停止が効かない。これは不便である。

2) 同等な処理をするCプログラムよりも遅い。これは不愉快である。

順序からいけば，まず1)のバグ(といえるだろう)をとってから，2)の高速化にとりかかるべきなのだが，Cで書いたプログラムよりも遅いのはどう考えても癪なので，さきに高速化をはかることにしたい。

高速化に関しては，さきほどのCOPYTEST.Xがヒントになる。read，writeを使ってデータをまとめて読み書きすることを検討してみよう。

いま作っているUPPER.Xのことだけを考えるなら，単純にreadでバッファにごっそりデータを読み込んで，バッファの中で小文字→大文字変換してからwriteでまとめて書き出すのがベストである。

しかし，別のフィルタを作るときにまた頭を悩ますことのないよう，ここではもう少し汎用性のある(使い回しの効く)入出力処理ルーチンをサブルーチンの形で作っておくことにする。

まず，入力側である。この入力サブルーチンは，メインプログラムから見ると見かけ上は1バイトずつ入出力を行うが，内部的にはごっそりとバッファにデータをまとめて入力するものとする。つまり，必要に応じてバッファにまとめてデータを読み込み，そのバッファから1バイトずつ取り出してくるサブルーチンだ。こうやってバッファへのデータ読み込みを巧妙にサブルーチンの奥底に隠してしまえば，いままでDOSコールを使って1文字入力していた部分を，このサブルーチンの呼び出しに変更するだけでよく，メインルーチンの骨格自体は変更しなくてすむだろう。

このサブルーチン(かりにgetcharと呼ぶ)は，次のような動作をする。

1) バッファにまだデータが残っていれば，そこから1バイト取り出してくる。

2) バッファが空であれば，DOSコールreadを呼び出してバッファをいっぱいにしてから1バイト持ってくる。

データを書き出すときはこの逆を考えればよい。1バイト書き出すサブルーチンputcharは，次のような動作をすることに決める。

1) バッファに1バイトずつデータを溜める。

2) バッファがいっぱいになったらDOSコールwriteで書き出す。

大筋はこれでよいのだが，キーボードから入力する場合はバッファがいっぱいになるまで入力を待つのではなく，1行分入力されたらすぐに変換処理をしてくれたほうがぐあいがよい。また，画面・プリンタなどのキャラクタデバイス[11]に出力するときにもバッファに1行分が揃ったらすぐ出力してくれたほうがありがたい。そこで，もうひと工夫する。

■11) キーボードやプリンタなど，入出力を1文字単位で行うデバイス(device：周辺装置ぐらいの意味)を〝キャラクタデバイス〟という。キャラクタデバイスはHuman68k上ではCONやPRNなどのデバイス名で表される。対して，フロッピーディスク，ハードディスクなどのように，入出力をあるまとまった単位で行うデバイスを〝ブロック

デバイス〟という。

入力に関しては,DOSコールreadがなかなかうまくできているので僕らが考えなければならないことはほとんどない。readはCONからの入力に使うと，次のように動作する。

1) バッファがいっぱいにならなくても,リターンキーが押された時点ですぐ戻る。何バイト読み込んだかはd0.lに返される。なお，1行は改行コードを入れて最大256バイトまでで，それ以上の入力は許されない。
2) 入力はエコーバックされる。つまり,打ち込んだ文字はその場で画面に表示される。また，リターンキーを押す前であれば，BSキーによる修正が効く。
3) リターンキーは$0D_H$，$0A_H$に置き換えて入力される。
4) ^Cを入力すると，プログラムの実行を中断する。

つまり，readを使えば，勝手に行単位で入力してくれるし，BSによる修正も効くし，さきほど泣かされた改行コードまわりもDOS側でつじつまあわせをしてくれるし，細工しなくても入力のキャンセルがいつでもできる，というわけだ。入力先がファイルの場合と区別する必要すらない。

次に，キャラクタデバイスへの出力を行単位にする方法だが，これは思ったより面倒くさい。いろいろ試してみた結果，最後の手段をとらざるをえなかった。すなわち，〝出力先＝標準出力がキャラクタデバイスかファイルかを調べ〟，処理を振り分けるのである。これによりサブルーチンputcharは，次のように変更される。

1) バッファに1バイトずつデータを溜める。
2) バッファがいっぱいになったら書き出す。
3) バッファがいっぱいにならなくても，出力先がキャラクタデバイスの場合は1行分が揃った時点で書き出してしまう。1行分が揃ったかどうかは，出力データがLFコードかどうかで判断する。

肝心の〝出力先がキャラクタデバイスかファイルかを識別する〟処理にはDOSコール＄FF44のioctrlを利用する。このDOSコールのモード0により，指定したファイルハンドルの装置情報を得ることができる。標準出力の装置情報が得たければ,

```
move.w    #1, -(sp)    *標準出力
clr.w     -(sp)        *モード0
DOS       _IOCTRL
addq.l    #4, sp
```

のようにする。

この結果，d0.lに図3に示すような情報が返される。図を見てもらえばわかるように，返された情報の第7ビットが0か1かで，ファイルかキャラクタデバイスかを判別することができる。1ならキャラクタデバイスである。

d0の第7ビットのON/OFFを調べる手順はいくつか考えられる。たとえば，ビットマスクを

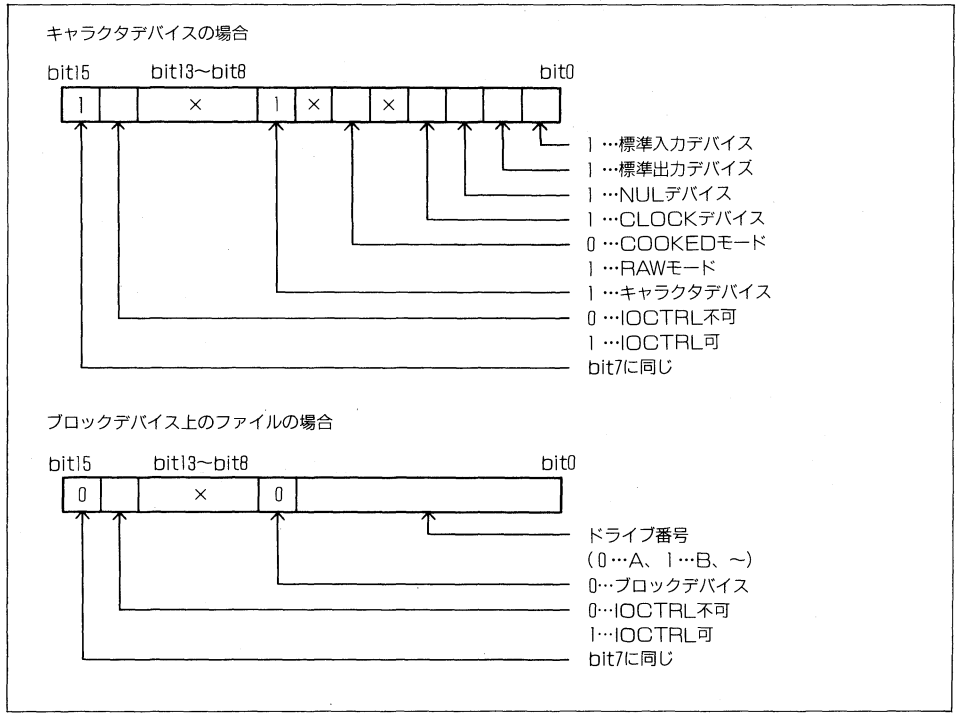


図3
装置情報

利用して,

```
andi.b  #%1000_0000, d0
bne     第7ビットが1の処理
```

というのも1つの方法だし,

$1XXXXXXX_B$

で表される2進8ビットの数は, 2倍すると8ビットで表される範囲を超えることを利用して,

```
add.b   d0, d0
bcs     第7ビットが1の処理
```

とやってもよい。

また, 第7ビットはバイトデータの符号ビットであることを利用すれば,

```
tst.b   d0
bmi     第7ビットが1の処理
```

となる。いちばんスマートなのは最後の書き方だろうか。

UPPER.X第4版

リスト9
CONST.H

```
1: *       定数定義ファイル
2:
3: *
4: *       コントロールコード
5: *
6: NULL            equ     0       *^@
7: BELL            equ     $07     *^G
8: BS              equ     $08     *^H
9: TAB             equ     $09     *^I
10: LF             equ     $0a     *^J
11: CR             equ     $0d     *^M
12: EOF            equ     $1a     *^Z
13: ESC            equ     $1b     *^[
14: SPACE          equ     $20     *スペース
15:
16: *
17: *       標準ファイルハンドル
18: *
19: STDIN          equ     0       *標準入力
20: STDOUT         equ     1       *標準出力
21: STDERR         equ     2       *標準エラー出力
22: STDAUX         equ     3       *標準補助装置
23: STDPRN         equ     4       *標準プリンタ
24:
25: *
26: *       ファイルアクセスモード
27: *
28: ROPEN          equ     0       *読み込み用
29: WOPEN          equ     1       *書き出し用
30: RWOPEN         equ     2       *読み書き両用
31:
32: *
33: *       ファイル属性
34: *
35: ARCHIVE        equ     $20     *ふつうのファイル
36: SUBDIR         equ     $10     *ディレクトリ
37: VOLUME         equ     $08     *ボリューム名
38: SYSTEM         equ     $04     *システムファイル
39: HIDDEN         equ     $02     *不可視ファイル
40: READONLY       equ     $01     *読み込み専用ファイル
```

では，UPPER.Xの第4版をリスト10に示す。4行でCONST.Hというファイルを取り込んでいる。その中身はリスト9のような定数定義である。よく使う定数は，このように別ファイルにしておくことで，毎回定義する手間が省ける。このファイルはこれからもしばしば使う。

リスト10
UPPER.S(その4)

```
1: *       英小文字→英大文字変換フィルタ　第4版
2:
```

```
 3:          .include        doscall.mac
 4:          .include        const.h
 5: *
 6:          .text
 7:          .even
 8: *
 9: BUFFSIZE         equ     1024    *バッファの大きさ
10: *
11: ent:
12:          lea.l   mysp,sp         *spの初期化
13:
14:          bsr     init            *入出力関係の初期化
15:
16:          bsr     do              *フィルタ本体
17:
18:          bsr     puteof          *ファイルエンドコードを出力
19:
20:          bsr     flushbuff       *書き出しバッファを吐き出す
21:          tst.l   d0              *
22:          bmi     werror          *
23:
24:          bsr     nl              *改行する（標準エラー出力）
25:
26:          DOS     _EXIT           *終了
27:
28: *
29: *       小文字→大文字変換メインループ
30: *
31: do:
32: loop:    bsr     getchar         *１バイト読み込む
33:          tst.l   d0              *エラーか？
34:          bmi     rerror          *そうならエラー終了
35:
36:          cmpi.b  #EOF,d0         *ファイルエンドコードか？
37:          beq     done            *そうなら終了
38:
39:          cmpi.b  #$80,d0         *80Hより小さければ
40:          bcs     hankaku         *       ASCIIコード
41:          cmpi.b  #$a0,d0         *80H以上A0H未満なら
42:          bcs     zenkaku         *       シフトJISの１バイト目
43:          cmpi.b  #$e0,d0         *A0H以上E0H未満なら
44:          bcs     hankaku         *       ASCIIカタカナ
45:                                  *E0H以上なら
46:                                  *       シフトJISの１バイト目
47:
48: zenkaku:                    *全角文字の処理
49:          bsr     putchar         *１バイト書き出す
50:          tst.l   d0              *
51:          bmi     werror          *
52:
53:          bsr     getchar         *もう１バイト読み込む
54:          tst.l   d0              *
55:          bmi     rerror          *
56:
57:          bsr     putchar         *そのまま出力する
58:          tst.l   d0              *
59:          bmi     werror          *
```

```
                bra     loop            *繰り返す

hankaku:                        *半角文字の処理
                bsr     toupper         *小文字→大文字変換

                bsr     putchar         *１バイト書き出す
                tst.l   d0              *
                bmi     werror          *

                bra     loop            *繰り返す

done:   rts             *変換終了

*
*       英小文字→英大文字変換
*
toupper:
                cmpi.b  #'a',d0         *英小文字か？
                bcs     toupr0          *
                cmpi.b  #'z'+1,d0       *
                bcc     toupr0          *
                subi.b  #$20,d0         *小文字なら大文字に変換
toupr0: rts                             *サブルーチンからリターン

*
*       ファイルエンドコードの出力（ファイルに対してのみ）
*
puteof:
                tst.b   devflg          *出力先が
                bmi     pteof0          *  ファイルのときのみ
                moveq.l #EOF,d0         *  EOFコードを
                bsr     putchar         *  出力する
                tst.l   d0              *
                bmi     werror          *
pteof0: rts

*
*       １バイト入力する
*
getchar:
                move.l  a0,-(sp)        * {レジスタを待避

                tst.l   rctr            *バッファにデータはあるか？
                bne     getc0           *あればそこから取り出す

                bsr     fillbuff        *バッファを満たす

                tst.l   d0              *d0<0...エラー, d0 = 0...EOF
                bmi     getc1           *エラーが発生した
                beq     eof             *ファイルが終わった

getc0:  movea.l rptr,a0         *ポインタを取り出す
                moveq.l #0,d0           *上位バイトを０にしておく
                move.b  (a0)+,d0        *バッファから１バイト取り出す
                move.l  a0,rptr         *ポインタを更新する
                subq.l  #1,rctr         *カウンタを更新する
```

```
117:            bra     getc1
118: *
119: eof:       moveq.l #EOF,d0         *EOFコードを持って帰る
120:
121: getc1:     movea.l (sp)+,a0        *} レジスタを復帰
122:            rts
123:
124: *
125: *          入力バッファを満たす
126: *
127: fillbuff:
128:            move.l  #BUFFSIZE,-(sp) *バッファにデータを読み込む
129:            pea.l   rbuff           *
130:            move.w  #STDIN,-(sp)    *
131:            DOS     _READ           *
132:            lea.l   10(sp),sp       *
133:
134:            move.l  #rbuff,rptr     *ポインタを再初期化
135:            move.l  d0,rctr         *カウンタを再初期化
136:            rts
137:
138: *
139: *          １バイト出力する
140: *
141: putchar:
142:            move.l  a0,-(sp)        * {レジスタを待避
143:
144:            andi.l  #$0000_00ff,d0  *上位ビットをマスクする
145:
146:            movea.l wptr,a0         *ポインタを取り出す
147:            move.b  d0,(a0)+        *バッファに１バイト追加する
148:            move.l  a0,wptr         *ポインタを更新する
149:            addq.l  #1,wctr         *カウンタを更新する
150:
151:            cmpi.l  #BUFFSIZE,wctr  *バッファが一杯になったか？
152:            bcc     putc0           *そうならバッファ内容を吐き出す
153:
154:            tst.b   devflg          *出力先はキャラクタデバイスか？
155:            bpl     putc1           *そうでなければリターン
156:            cmpi.b  #LF,d0          *出力データはLFコードか
157:            bne     putc1           *そうでなければリターン
158:
159: putc0:     bsr     flushbuff       *バッファ内容を吐き出す
160:
161: putc1:     movea.l (sp)+,a0        *} レジスタを復帰
162:            rts
163:
164: *
165: *          書き出しバッファ内容を吐き出す
166: *
167: flushbuff:
168:            tst.l   wctr            *バッファが空であれば
169:            beq     flush0          *  なにもしない
170:
171:            move.l  wctr,-(sp)      *バッファ内容を書き出す
172:            pea.l   wbuff           *
173:            move.w  #STDOUT,-(sp)   *
```

```
174:            DOS     _WRITE          *
175:            lea.l   10(sp),sp       *
176:            tst.l   d0              *エラー？
177:            bmi     flush0          *  エラーコードを持って帰る
178:                                    *d0.l = 実際に書き出したバイト数
179:            sub.l   wctr,d0         *d0.l=wctr ... d0.l = 0  非エラー
180:                                    *d0.l<wctr ... d0.l < 0  エラー
181:
182:            move.l  #wbuff,wptr     *ポインタを再初期化
183:            clr.l   wctr            *カウンタを再初期化
184:
185: flush0:   rts
186:
187: *
188: *       入出力初期化
189: *
190: init:
191:            move.l  #rbuff,rptr     *読み込みバッファへのポインタ
192:            move.l  #wbuff,wptr     *書き出しバッファへのポインタ
193:            clr.l   rctr            *読み込み用カウンタ
194:            clr.l   wctr            *書き出し用カウンタ
195:
196:            move.w  #STDOUT,-(sp)   *標準出力の装置情報を取り出す
197:            clr.w   -(sp)           *
198:            DOS     _IOCTRL         *
199:            addq.l  #4,sp           *
200:
201:            move.b  d0,devflg       *devflgの第７ビットが１であれば
202:                                    *  キャラクタデバイス
203:            rts
204:
205: *
206: *       カーソルが画面の左端になければ改行する
207: *
208: nl:
209:            move.w  #-1,-(sp)       *カーソル座標を取り出す
210:            move.w  #-1,-(sp)       *
211:            move.w  #3,-(sp)        *
212:            DOS     _CONCTRL        *
213:            addq.l  #6,sp           *
214:                                    *d0.l = xxxx_yyyy
215:            swap.w  d0              *d0.l = yyyy_xxxx
216:            tst.w   d0              *x座標は０か？
217:            beq     nl0             *  ０ならなにもしない
218:
219: ltnl:     move.l  a0,-(sp)        * {
220:            lea.l   crlfms,a0       *改行する
221:            bsr     puterr          *
222:            movea.l (sp)+,a0        *}
223:
224: nl0:      rts
225:
226: *
227: *       メッセージ表示（標準エラー出力へ）
228: *
229: puterr:   move.w  #STDERR,-(sp)   *標準エラー出力へ
230:            move.l  a0,-(sp)        *  文字列を
```

```
231:            DOS     _FPUTS          *  出力する
232:            addq.l  #6,sp           *スタック補正
233:            rts
234:
235: *
236: *       エラー終了
237: *
238: rerror:  lea.l   rerrms,a0        *読み込み時エラー
239:            bra     error
240: werror:  lea.l   werrms,a0        *書き出し時エラー
241:            *
242: error:   bsr     puterr          *メッセージを表示
243:
244:            move.w  #1,-(sp)        *終了コード１を持って
245:            DOS     _EXIT2          *  エラー終了
246:
247: *
248: *       メッセージデータ
249: *
250:            .data
251:            .even
252: *
253: rerrms:  .dc.b   CR,LF,'UPPER：うまく読み込めませんでした',CR,LF,0
254: werrms:  .dc.b   CR,LF,'UPPER：うまく書き出せませんでした',CR,LF,0
255: crlfms:  .dc.b   CR,LF,0
256:
257: *
258: *       ワークエリア
259: *
260:            .bss
261:            .even
262: *
263: rptr:    .ds.l   1               *読み込みポインタ（getcharで使用）
264: rctr:    .ds.l   1               *読み込みカウンタ（getcharで使用）
265: wptr:    .ds.l   1               *書き出しポインタ（putcharで使用）
266: wctr:    .ds.l   1               *書き出しカウンタ（putcharで使用）
267: *
268: devflg:  .ds.b   1               *出力先フラグ（putchar,puteofで使用）
269:                                   *        bit7=0...ファイル
270:                                   *             =1...キャラクタデバイス
271: *
272: rbuff:   .ds.b   BUFFSIZE        *読み込みバッファ（getcharで使用）
273: wbuff:   .ds.b   BUFFSIZE        *書き出しバッファ（putcharで使用）
274: *
275:            .stack
276:            .even
277: *
278: mystack:
279:            .ds.l   256             *スタック領域
280: mysp:
281:            .end
```

リストがいきなり長くなってしまったが，83行までのメイン部分はほとんどいままでのままであり，その後ろに入出力サブルーチンが追加されただけだから圧倒されないようにしよう。

では，個々のサブルーチンの解説である。リストと対比しながら読んでもらいたい。

1バイト入力サブルーチン

100行以下が，ファイルから1バイト入力を行うサブルーチンだ。入力データはd0.bに入れて戻るが，エラーが発生した場合はd0.lに負の数を返すものとする。この関係でエラーでなかった場合は，d0.l全体が正の数の状態で戻るようにつじつまをあわせる必要がある。サブルーチン内では，265～266行で用意してあるワークエリアrptrとrctrを，それぞれバッファ内の次に読み込むべきデータを指すポインタ，バッファ内の残りデータ数を表すカウンタとして使用している。読み込みバッファは272行で確保している。

サブルーチンgetcharでは，まずカウンタrctrが0かどうかでバッファにデータが残っているかどうかを調べる(103行)。バッファが空であれば，サブルーチンfillbuffを呼び出して(106行)バッファをデータでいっぱいにする。fillbuffの中身はDOSコールreadによる読み込みと，ポインタ，カウンタの再初期化である。ポインタであるrptrはバッファの先頭を指すようにし(134行)，読み込んだバイト数をカウンタrcntに入れておく(135行)。このくらいのことであれば，サブルーチンにせずにそのままgetcharに埋め込んでもよいように見えるが，こうやって機能を分散することでプログラムの動作をよりはっきりさせている[12]。

■12) 68000にはレジスタがたくさんあるから，rptrやrctrのようなワークエリアを使うかわりに余ったレジスタをポインタ，カウンタとして使うことも考えられる。実際，そのほうが処理速度がいくらか速くなることが期待できる。しかし，その半面，サブルーチンの独立性・汎用性が落ちてしまう。つまり，サブルーチンでどのレジスタを使うか覚えておいて，メインルーチンや他のサブルーチンで壊さないように気をつけなければならない。他のプログラムに流用するのも難しくなるだろう。

128行や272行のバッファ確保で使っているBUFFSIZEというラベルは9行で1024に定義してある。1024と直接書かずに記号定数を使っているのは，後でバッファのサイズを変更する(かもしれない)場合に備えてのことだ。もし，1024という数字で記述したとすると，変更時にリスト中のすべての1024を探し出して書き換える必要があり，もしかすると1ヵ所くらい見落とすかもしれないし，関係のない1024をあやまって書き換えてしまうかもしれない。記号定数を使えば，定義行を修正するだけですむわけである。また，数字ではなく意味のある単語を記号定数として使うことでプログラムの読みやすさも上がることが期待できる。

なお，記号定数に大文字を使っているのは，小文字で書かれた他のラベルと一目で区別できるようにする意味がある。

130行では，STDIN[13]という記号定数でファイルハンドルを指定している。STDINは，CONST.H内で0に定義してある。

■13) STDINは，STanDard INputの略で，標準入力を意味する一般的な略語。

ここは

```
    move.w      #0, -(sp)
```

または

```
    clr.w       -(sp)
```

と書いてもよいのだが，あえて

```
    move.w      #STDIN, -(sp)
```

と書くことによって，標準入力から読み込むことを(プログラムを読む人間に対して)強調しているのだ。さきほどclrを使ったほうがスマートだといったが，ここではほんのわずかコードを短く速くすることよりも，プログラムを読みやすくするほうを優先している。

サブルーチンfillbuffから戻った時点で，d0.lにはDOSコールreadの戻り値がそのまま入っている。tst命令により符号を調べ，負であればエラーなので，エラー番号を持ったままメインルーチンに戻る(109行)。エラーかどうかはメインルーチンでもう1度調べなおすことになる。これは2度手間に見えるかもしれないが，サブルーチンにはよけいなことはさせないのが汎用性を保つための鉄則であり，それにしたがったまでのことだ。それぞれのサブルーチンは与えられた仕事だけを黙々とこなしてくれさえすればよいという考え方である。

また，fillbuffの戻り値(DOSコールreadの戻り値d0.l)が0であれば入力がもうないことになるので，119行に飛んでファイルエンドコードEOFをd0.lに入れてメインルーチンに戻る。このように作っておくことでファイルの物理的な終わり(ファイルの本当の最後に達した場合)と論理的な終わり(ファイルエンドコードが現れた場合)をメインルーチン側で区別する必要がなくなる。メインルーチンでは，getcharからの戻り値がEOFであるかどうかを調べればすむわけである。

なお，入力先がファイルの場合は，readの戻り値がBUFFSIZE未満かどうかでもファイル終端を調べることができるわけだが，キーボードからの入力と処理を兼用した関係で，この方法は使えないことに注意してほしい(キーボードから入力する場合のreadの戻り値はつねに256以下だから，BUFFSIZEが1024の場合はつねにBUFFSIZE未満と見なされてしまう)。

バッファが空でなかった場合には直接，またバッファが空だった場合にはデータで満たした後で112行に制御が移る。ここが実際にバッファから1バイトデータを取り出す部分で，ポインタrptrをa0に取り出しておき，a0の指すアドレスから1バイトデータをd0に転送している(112～114行)。a0はポストインクリメントしているから，読み込み後自動的に次の読み出し位置を指す。これをそのままrptrにしまえば，ポインタを更新したことになる(115行)。

データをメモリから読み出す直前に

```
    moveq.l     #0, d0
```

を実行している(113行)のは，エラーが起きた場合と区別するためである。あらかじめd0.lを0にしてからd0.bにデータを取り出すことで，d0.l全体がつねに正であることを保証している。こうしておかないと，fillbuffでエラーが発生した場合との区別がつけられない。

データを取り出したら，カウンタrctrから1を引く(116行)。こうしてカウンタは徐々に減っていき，0になったら〝次にgetcharが呼び出された時点で〟fliibuffによりふたたびデータで

満たされる。

なお，いまバッファサイズは1024バイトなのでカウンタはワードですむのだが，将来の変更に備え，ゆとりをもってロングワードのカウンタを使うようにしてある。

1バイト出力サブルーチン

141行からが1バイト書き出すサブルーチンputcharだ。d0.bに出力データを入れて呼び出す。ワークエリアwptrが書き出しバッファ内の次の位置を表すポインタとして，rctrがバッファに溜まったデータのバイト数を数えるカウンタとして使われている。

まず，

```
andi.l  #$0000_00ff,d0
```

によって，d0の最下位バイトを保存したまま上位ビットを0にして，d0.l全体が正の数になるようにしている。これはgetcharのときと同様，サブルーチンから戻った後でエラーの場合と区別できるようにするためである。しかし，このプログラムをよく見ると，putcharが呼び出された時点でのd0.lはgetcharで上位ビットをクリアしたままの状態を保っているので，わざわざマスクする必要はない。あくまで安全のためだ。将来getcharの仕様を変更した場合や，メインルーチンに処理を追加した場合でも，このサブルーチンに手を加える必要がないように作ってあるわけだ。

この後，出力データであるd0.bをwptrの指すアドレスへ書き込んでポインタを更新し(146～148行)，カウンタwctrをインクリメントする(149行)。その結果，wctrがBUFFSIZEに達したら(151行)，バッファがいっぱいになったことになるので，サブルーチンflushbuffを呼び出し(159行)，バッファ内容を標準出力へ書き出してバッファを空にする。

ここで，バッファがいっぱいになったかどうかを調べる比較後の条件分岐に，beqではなくbccを使っている(152行)のが奇異に映るかもしれない。カウンタがBUFFSIZEと等しければ書き出すのではなく，〝以上であれば〟書き出すようになっているわけだ。現実にはカウンタはBUFFSIZEと等しくなることはあっても，より大きくなることはないから，beqで間に合うはずである。が，どちらでもよいのであればチェックは厳しいにこしたことはない。たとえ〝そんなことは絶対に起こらない〟としてもである。

また，キャラクタデバイスに対して行単位で出力が行われるようにするために，出力先がキャラクタデバイスであり(154行)，かつ今回の出力データがLFコードであれば(156行)，やはりflushbuffを呼び出す。ここで参照しているワークエリアdevflgには出力先の装置情報の下位バイトを前もって格納してある。〝先にできることはやっておく〟ことで，毎回ioctrlを呼び出す手間を省いているのである。

putcharの下位ルーチンであるflushbuffは，DOSコールwriteによる書き出しとポインタの再初期化を受け持つ。指定したバイト数だけ書き出せたかどうかのチェックを行う179行が多少作為的かもしれない。

またflushbuffは，すでにバッファが空であった場合にはなにもしないで戻るように作ってある(168～169行)。実際にはputcharからflushbuffが呼び出されるのはバッファにデータがあるときだけだし，DOSコールwriteは書き出しバイト数を0にして呼び出しても期待どおりの動作をしてくれる(なにも書き出さない)ので，このチェックは不用である。くどいようだが，これまたプログラムの安全性と読みやすさを重視したコーディングになっている。

初期化

190行以下が，getcharやputcharで使うワークエリアの初期化を行うサブルーチンだ。ここではrptrが読み込みバッファの先頭を，wptrが書き出しバッファの先頭を指すようにしたうえで，カウンタとして使うrctr，wctrを0で初期化している。rptrはfillbuffで勝手に初期化されるから，ここで初期化する必要はないのだが，例によって安全のためである。

念のため，この初期化が正しいかどうか確認しておこう。サブルーチンgetcharがはじめて呼び出されたときは，カウンタrctrが0に初期化されているのでfillbuffが呼び出され，バッファがデータで満たされる。rptrは読み込みバッファの先頭を指すようになり，以後の読み出しは正しく行われる。

また，putcharがはじめて呼び出されるときには，ポインタwptrは書き出しバッファの先頭を指しているし，カウンタwctrは0になっているから，やはり以後の書き出しは問題なく行われる。

これらの初期化の後に，196行以下で標準出力がファイルに割り当てられているか，キャラクタデバイスかという情報をワークエリアdevflgに格納している。196～199行はさきほど示した装置情報の取り出し手順であり，201行で必要な部分である下位バイトだけをワークにしまっている。

エラー処理ルーチン

238行以下は，エラー処理ルーチンになっている。読み込み時のエラー処理はrerrorで，書き出し時のエラー処理はwerrorで行う。どちらもかんたんなメッセージを出すだけだ。メッセージの表示自体は，サブルーチンputerrで行っている。puterrはa0にエラーメッセージの先頭アドレスを入れて呼び出すと，DOSコールfputs(ファイルハンドルへ1行出力する。printのファイル版)を使って標準エラー出力にメッセージを出力する。DOSコールprintを使わずにfputsで標準エラー出力に書き出すようになっているのは，標準出力がリダイレクトされていてもエラーメッセージをつねに画面に表示するためだ。printを使うと，リダイレクトされたときにエラーメッセージが画面ではなく出力ファイルのほうに混ざってしまうことになる。

エラーメッセージは253行以下に用意してある。日本語が当たり前のように使えるX68000だから，エラーメッセージもまた日本語である。エラーメッセージが必ず行の先頭から表示され

るようにするために，メッセージの先頭には余分に見える改行コードがついている。

エラーメッセージの直前に，

```
.data
```

という行がある。これは〝ここからデータ領域が始まる〟という宣言だ。すでに出てきた.textや.stackの同類である。この行はなくてもべつに困らないのだが，せっかくアセンブラに用意されているのだから，今後データ領域は.dataの後に置くことにしよう。

ついでに，263行以下のワークエリア用メモリ領域確保の直前にも，

```
.bss
```

という宣言がついている。これは，〝非初期化データ領域〟の始まりを意味する。.dsで大きな領域を確保するときには，このセクションに置くと実行ファイルが必要以上に大きくなるのを避けられるのだ。

メインルーチン

メインルーチンの基本的な流れは昔のままだが，いままでメインルーチンに埋め込まれていた小文字→大文字変換処理本体はごっそり抜き出してサブルーチンにするなど，必要な処理はすべてサブルーチン側で行うようにしてある。12行から順にスタックの初期化をしてから，初期化サブルーチン，変換処理本体を呼び出している。

大文字→小文字変換のメイン処理を行うサブルーチンdoの構造は以前と変わっていないが，DOSコールを使って行っていた入出力をサブルーチンの呼び出しに置き換えたなどの細かな変更が加えられている。

第3版と比べると，2ヵ所で調べていた入力の終了テストがただ1ヵ所(36～37行)にまとめられた。これはサブルーチンgetchar側で細工した結果である。また，改行コードまわりの処理がなくなった。キーボードから入力した場合とファイルから入力した場合の改行コードの差は，DOSコールreadが吸収してくれているからだ。

メイン処理がすんだら，もろもろの後始末をする。

1) 出力先がファイルの場合にかぎり，ファイルエンドコードを出力する(サブルーチンputeof)。

   Human68kの他のフィルタはつねにファイルエンドコードを出力しないことはすでに話したとおりだが，UPPER.Xではすでに出力先がファイルかどうかを調べてあるのを利用して，出力先がファイルの場合にかぎって最後にファイルエンドコード$1A_H$を書き出すように細工して体裁を繕ってみた。

2) 書き出しバッファに溜まっている〝かもしれない〟データを掃き出す(サブルーチンflushbuff)。

   これを忘れると，出力の最後が欠けてしまう可能性がある。理由はサブルーチンputcharの作りから考えよう。

3) カーソル位置が行頭にない場合にのみ，標準エラー出力に改行コードを送る(サブルーチンnl)。

これは，画面に出力する場合に改行しないまま終了する可能性があるからである。ちなみに，209～213行がカーソル座標を得る一般的な手順である。conctrlのモード3はカーソル位置の移動を行うものだが，座標に－1を指定することでカーソル位置を変えずに現在の座標を得ることができる(『プログラマーズマニュアル』には，このあたりの記述がない)。この結果，d0.lの上位ワードにカーソルX座標が，下位ワードにY座標が返される。以下，swapで上位下位ワードを交換して(215行)，tst.wでX座標が0かどうかを調べ(216行)，0でなければ改行処理をしている(219～222行)。

最後のバグ取り:UPPER.X第5版

では，動作試験をしてもらいたい。プログラムは大幅に改良され，ひょっとすると新しいバグが入りこんでいるかもしれないので，念入りにテストしてほしい。また，せっかく高速化したのだから，第3版やC_UPPER.Xと速度比較もしてみよう。

これでよしということになれば，いよいよ最後に残ったバグをつぶす。ブレイクチェック関係だ。念のため，このバグを再現してみる。

```
A>UPPER <UPPER.S
```

を実行する。UPPER.Sが小文字→大文字変換されて画面に表示されるだろう。放っておくとどんどんスクロールしてしまうから，CTRL＋Sを押して表示を一時停止する……が，止まってくれない。BREAKキーによる中断も効かない。DOSコールwriteは，出力先が画面の場合はブレイクチェックしてくれることになっているから，この結果はおかしい。

原因はHuman68kのブレイクチェック機能がキーボードではなく，標準入力を調べて行っていることにある。だから，標準入力がファイルにリダイレクトされているときには，BREAKキーやCTRL＋Sを押してもDOSは〝見てくれない〟のだ。逆に，リダイレクト先のファイルに ^Cが含まれていると，そこで勝手にブレイクしてしまうこともある。

そこで〝標準入力をキーボードに割り当て直す〟という技を使う。具体的に，どうやるのかというと，標準入力を意味するファイルハンドル0をクローズしてしまうのだ。ふつうのファイルならクローズすると読み書きできなくなってしまうわけだが，標準入力はクローズすることによってリダイレクションを解除して(リダイレクトされていたファイルがクローズされる)標準状態であるキーボードに戻るようにできている。

さて，リダイレクションを解除してしまうと，今度は〝それまで標準入力にリダイレクトされていたファイル〟が読めなくなってしまうから，その前にもうひとひねりする。DOSコールdup(DUPlicateの意)を使うと，ファイルハンドルの複写ができるので，これを利用する。ここまでをまとめると，次のような感じだ。

1) ファイルハンドル0をdupで複写する。すると,新しいファイルハンドルが返され，いままで標準入力であったものが，ファイルハンドル0を使っても新しいファイルハンドルを使っても読み込めるようになる。

2) 1つの入力にファイルハンドルは2つもいらないので，すかさずファイルハンドル0をクローズする。標準入力はキーボードに戻る。

3) 以後，標準入力から読み込むかわりにコピーしたファイルハンドルを使う。

この改良を加えたのがリスト11だ。修正箇所はサブルーチンinitの中で標準入力のファイルハンドルを複写し，ワークrfnoに格納するようにした部分と，これにともない，サブルーチンfillbuffでのファイルハンドルの指定をrfnoで行うようにしたこと,プログラム終了時にオープンしたファイルハンドルをクローズするようにしたこと，これだけだ。

試しに，この第5版で

```
A>UPPER <UPPER.S
```

を実行し，今度は ^C, ^Sがきくことを確認してもらいたい。

リスト11
UPPER.S(その5)
その4からの追加
変更部分のみ

```
  1: *       英小文字→英大文字変換フィルタ　第5版
  2:
  3:         .include        doscall.mac
  4:         .include        const.h
  5: *
  6:         .text
  7:         .even
  8: *
  9: BUFFSIZE        equ     1024    *バッファの大きさ
 10: *
 11: ent:
 12:         lea.l   mysp,sp         *spの初期化
 13:
 14:         bsr     init            *入出力関係の初期化
 15:
 16:         bsr     do              *フィルタ本体
 17:
 18:         bsr     puteof          *ファイルエンドコードを出力
 19:
 20:         bsr     flushbuff       *書き出しバッファを吐き出す
 21:         tst.l   d0              *
 22:         bmi     werror          *
 23:
 24:         bsr     nl              *改行する（標準エラー出力）
 25:
 26:         DOS     _ALLCLOSE       *全ファイルクローズ
 27:
 28:         DOS     _EXIT           *終了

126: *
127: *       入力バッファを満たす
128: *
129: fillbuff:
130:         move.l  #BUFFSIZE,-(sp) *バッファにデータを読み込む
```

```
131:            pea.l   rbuff           *
132:            move.w  rfno,-(sp)      *
133:            DOS     _READ           *
134:            lea.l   10(sp),sp       *
135:
136:            move.l  #rbuff,rptr     *ポインタを再初期化
137:            move.l  d0,rctr         *カウンタを再初期化
138:            rts

189: *
190: *          入出力初期化
191: *
192: init:
193:            move.l  #rbuff,rptr     *読み込みバッファへのポインタ
194:            move.l  #wbuff,wptr     *書き出しバッファへのポインタ
195:            clr.l   rctr            *読み込み用カウンタ
196:            clr.l   wctr            *書き出し用カウンタ
197:
198:            move.w  #STDOUT,-(sp)   *標準出力の装置情報を取り出す
199:            clr.w   -(sp)           *
200:            DOS     _IOCTRL         *
201:            addq.l  #4,sp           *
202:
203:            move.b  d0,devflg       *devflgの第７ビットが１であれば
204:                                    *  キャラクタデバイス
205:
206:            move.w  #STDIN,-(sp)    *標準入力のファイルハンドルを
207:            DOS     _DUP            *  複製する
208:            addq.l  #2,sp           *
209:            tst.l   d0              *エラー？
210:            bmi     rerror          *  そうならエラー終了
211:
212:            move.w  d0,rfno         *複製したファイルハンドルをしまう
213:
214:            move.w  #STDIN,-(sp)    *標準入力をクローズして
215:            DOS     _CLOSE          *  キーボード(CON)に割り当てを戻す
216:            addq.l  #2,sp           *
217:            tst.l   d0              *エラー？
218:            bmi     rerror          *  そうならエラー終了
219:
220:            rts

274: *
275: *          ワークエリア
276: *
277:            .bss
278:            .even
279: *
280: rptr:      .ds.l   1               *読み込みポインタ（getcharで使用）
281: rctr:      .ds.l   1               *読み込みカウンタ（getcharで使用）
282: rfno:      .ds.w   1               *入力ファイルハンドル（getcharで使用）
283: wptr:      .ds.l   1               *書き出しポインタ（putcharで使用）
284: wctr:      .ds.l   1               *書き出しカウンタ（putcharで使用）
285: *
286: devflg:    .ds.b   1               *出力先フラグ（putchar,puteofで使用）
287:                                    *       bit7=0...ファイル
```

バッファにいつまでも残ることがわかる。ブレイクチェックではキーバッファの先頭しか調べないから，この後で ^Sを押してもチェックに引っかからないわけだ[14]。

■14) SHIFT+BREAKでも ^Sの入力ができる(マニュアルには明記されていないが)。しかも，同時にキーバッファをクリアしてくれるようだ。

そこで，プログラムの中で一定間隔でキーバッファをクリアすることを考える。キーバッファのクリア自体は，DOSコール＄FFのkflushを使って次のようにすればよい。

```
move.w    #-1, -(sp)
DOS       _KFLUSH
addq.l    #2, sp
```

これはマニュアルにあるとおりの書き方ではないが，たんにキーバッファをクリアするときの決まった形だと思ってもらいたい。

ただ，キーバッファのクリアをあまりしつこく行うとプログラム全体の実行速度が落ちてしまうので，UPPER.Xでは出力先がキャラクタデバイスのときにのみ，1行出力する直前にキーバッファを空にする処理を行うものとする。

もう1つの改良も使い勝手に関するものだ。フィルタはキーボードから入力して使うことはめったになく，通常はリダイレクションとともに使用される。一般には

```
A>UPPER <UPPER.S
```

のように〝<〟記号をつけて入力ファイルを指定するわけだ。ここで，この〝<〟を省略して，

```
A>UPPER UPPER.S
```

という指定のしかたが許されれば，わずか1文字ではあるが，タイプ量を減らすことができるし，見た目も自然である。調べてみると，Human68k上の他のフィルタ群MORE，FIND，SORTなども，この形式を許しているようだ。UPPERもこれに準ずることにしよう[15]。

■15) なお，コマンド行の〝<〟，〝>〟，〝|〟以下は，引数文字列には含まれないことになっている。

なお，MOREなどは入力ファイルのみがこの形式で指定可能であり，出力はあくまでリダイレクトに頼らなければならないが，UPPER.Xは出力も〝>〟なしに指定できるようにする。この新しい版のUPPER.Xの使い方は，次のようになる。

```
UPPER [入力ファイル [出力ファイル]]
```

[16]

■16) 〝[ ]〟で囲んだ部分は省略可能を意味する。

出力ファイルが指定されなかった場合は標準出力に書き出し，入力ファイルも指定されなかった場合は標準入力から入力して標準出力に書き出すものとする。出力ファイルのみを指定することができないのがイマイチのような気もするが，ま，妥当な線だろう。

さて，この改良を行うためには，COMMAND.Xのコマンド行から引数を受け取る方法を知らなければならない。『プログラマーズマニュアル』で調べるとわかるように，プログラム起動

時のa2レジスタがこのコマンドライン文字列を指している。ただし，1バイト目は文字列の長さを表し，2バイト目以降が実際のコマンドライン文字列だ。文字列は00_Hで終わる。この形式は，以前話した文字列の表現方法2種類を混ぜた形になっているのがわかる。つまり，先頭の1バイトを無視すれば，僕たちがいつも使っている形式の文字列であり，末尾の00_Hを無視すれば，文字数をべつに持った形式の文字列になる。好きなほうで処理できるわけだから，使い慣れた〝終端コード方式〟で処理することにしよう。

UPPER.Xのコマンドライン引数取り込み処理は，次のようになる。

1) a2に1を足す。a2は，コマンドライン文字列の先頭を指す。
2) 余分なスペースがあるかもしれないので，半角スペースのコード(20_H)およびTABコード(09_H)をスキップする(a2がスペースやTABを指している間，ポインタをどんどん進める)[17]。

■17) コマンド行でTABを区切りに使う人はめったにいないだろうが，バッチファイルの中ではTABを使うこともあるので，スペースキャラクタとTABコードを同列に扱う。

3) この時点で文字列が終わっていれば，パラメータはなにも指定されなかったことになるので，標準入出力を使ってデータをやりとりする。
4) まだ文字列があれば，それは入力ファイル名である。指定されたファイルを読み込み用にオープンする。
5) ファイル名の末尾にまでポインタを進め，またスペースをスキップする。
6) この時点で文字列が終わっていれば，出力ファイルが指定されなかったことになるので，標準出力を使って出力を行う。
7) まだ文字列があれば，それは出力ファイル名である。指定されたファイルを書き出し用にオープンし，以後の出力はこのファイルに対して行う。
8) ファイル名の末尾にまでポインタを進め，またスペースをスキップする。もし文字列がまだあれば，パラメータが多すぎるからエラーである。

ここで，8)の場合はたんにエラーメッセージを出すのではなく，かんたんな使用法を表示するようにする。また，Human68kのプログラムは

```
A>SORT /?
```

とか

```
A>SORT -?
```

のように，意味を持たないスイッチが指定された場合には使用法を表示して実行を終えるので，UPPER.Xもそれにしたがう。UPPER.Xには起動スイッチがまったくないから，とにかく〝/〟か〝−〟が指定されたら，使用法を表示するものとしよう。ついでに

```
A>UPPER ?
```

のように〝?〟が単独で指定された場合も同列に扱うものとする。このチェックはスペースをスキップした直後に行うことになるだろう。

完成:UPPER.X最終版

リスト12がUPPER.Xの最終版である。キーバッファのクリアはputcharの中で，パラメータの取り込みはサブルーチンinitとその下請けサブルーチンで行うようにしてある。その他，細かなところでは，サブルーチンflushbuff内のファイルハンドルが定数ではなく，wfnoで指定するように変更されている。

リスト12
UPPER.S(その6)
その5からの追加・
変更部分のみ

```
  1: *       英小文字→英大文字変換フィルタ　最終版
140: *
141: *       1バイト出力する
142: *
143: putchar:
144:         move.l  a0,-(sp)        * {レジスタを待避
145:
146:         andi.l  #$0000_00ff,d0  *上位ビットをマスクする
147:
148:         movea.l wptr,a0         *ポインタを取り出す
149:         move.b  d0,(a0)+        *バッファに1バイト追加する
150:         move.l  a0,wptr         *ポインタを更新する
151:         addq.l  #1,wctr         *カウンタを更新する
152:
153:         cmpi.l  #BUFFSIZE,wctr  *バッファが一杯になったか？
154:         bcc     putc0           *そうならバッファ内容を吐き出す
155:
156:         tst.b   devflg          *出力先はキャラクタデバイスか？
157:         bpl     putc1           *そうでなければリターン
158:         cmpi.b  #LF,d0          *出力データはLFコードか
159:         bne     putc1           *そうでなければリターン
160:
161:         move.w  #-1,-(sp)       *キーバッファを空にする
162:         DOS     _KFLUSH         *
163:         addq.l  #2,sp           *
164:
165: putc0:  bsr     flushbuff       *バッファ内容を吐き出す
166:
167: putc1:  movea.l (sp)+,a0        *} レジスタを復帰
168:         rts
169:
170: *
171: *       書き出しバッファ内容を吐き出す
172: *
173: flushbuff:
174:         tst.l   wctr            *バッファが空であれば
175:         beq     flush0          *  なにもしない
176:
177:         move.l  wctr,-(sp)      *バッファ内容を書き出す
178:         pea.l   wbuff           *
179:         move.w  wfno,-(sp)      *
180:         DOS     _WRITE          *
```

```
181:          lea.l   10(sp),sp       *
182:          tst.l   d0              *エラー？
183:          bmi     flush0          *  エラーコードを持って帰る
184:                                  *d0.l = 実際に書き出したバイト数
185:          sub.l   wctr,d0         *d0.l=wctr ... d0.l = 0  非エラー
186:                                  *d0.l<wctr ... d0.l < 0  エラー
187:
188:          move.l  #wbuff,wptr     *ポインタを再初期化
189:          clr.l   wctr            *カウンタを再初期化
190:
191: flush0: rts
192:
193: *
194: *       入出力初期化
195: *
196: init:
197:          move.l  #rbuff,rptr     *読み込みバッファへのポインタ
198:          move.l  #wbuff,wptr     *書き出しバッファへのポインタ
199:          clr.l   rctr            *読み込み用カウンタ
200:          clr.l   wctr            *書き出し用カウンタ
201:
202:          addq.l  #1,a2           *a2=コマンドライン
203:          bsr     ropen           *rfno=入力先のファイルハンドル
204:          bsr     wopen           *wfno=出力先のファイルハンドル
205:
206:          bsr     nextarg         *つぎの引数が
207:          tst.b   (a2)            *  あるか？
208:          bne     usage           *引数が多過ぎる
209:
210:          rts
211:
212: *
213: *       入力先ファイルハンドルを得る
214: *
215: ropen:
216:          bsr     nextarg         *つぎの引数の先頭アドレスを得る
217:          tst.b   (a2)            *文字列はまだあるか？
218:          beq     ropen0          *なければファイル名指定なし
219:
220: *ファイル名の指定があった場合
221:          bsr     getarg          *ファイル名をtempに抜き出す
222:
223:          move.w  #ROPEN,-(sp)    *指定されたファイルを
224:          pea.l   temp            *  リードオープンする
225:          DOS     _OPEN           *
226:          addq.l  #6,sp           *
227:          bra     ropen1
228:
229: *ファイル名の指定がなかった場合
230: ropen0: move.w  #STDIN,-(sp)    *標準入力のファイルハンドルを
231:          DOS     _DUP            *  複製する
232:          addq.l  #2,sp           *
233:
234: *d0には
235: *  ファイル名の指定があったときはオープンしたファイルハンドルが
236: *  指定がなかったときは標準入力を複製したファイルハンドルが入っている
237: ropen1: tst.l   d0              *エラー？
```

```
238:            bmi     rerror          *  そうならエラー終了
239:
240:            move.w  d0,rfno         *入力先ファイルハンドルをしまう
241:
242:            move.w  #STDIN,-(sp)    *標準入力をクローズして
243:            DOS     _CLOSE          *  キーボード(CON)に割り当てを戻す
244:            addq.l  #2,sp           *
245:            tst.l   d0              *エラー？
246:            bmi     rerror          *  そうならエラー終了
247:
248:            rts
249:
250: *
251: *       出力先ファイルハンドルを得る
252: *
253: wopen:
254:            move.w  #STDOUT,wfno    *仮に標準出力のハンドルをセットしておく
255:
256:            bsr     nextarg         *つぎの引数の先頭アドレスを得る
257:            tst.b   (a2)            *文字列はまだあるか？
258:            beq     wopen1          *なければファイル名指定なし
259:
260: *ファイル名の指定があった場合
261:            bsr     getarg          *ファイル名をtempに抜き出す
262:
263:            move.w  #ARCHIVE,-(sp)  *指定されたファイルを
264:            pea.l   temp            *  新規作成する
265:            DOS     _CREATE         *
266:            addq.l  #6,sp           *
267:            tst.l   d0              *エラー？
268:            bpl     wopen0          *  エラーがなければオープン完了
269:
270:            move.w  #WOPEN,-(sp)    *createでエラーが発生したときは
271:            pea.l   temp            *  openを使って
272:            DOS     _OPEN           *  もう一度ライトオープンしてみる
273:            addq.l  #6,sp           *
274:            tst.l   d0              *エラー？
275:            bmi     werror          *  そうなら今度こそエラー終了
276:
277: wopen0: move.w  d0,wfno          *出力先ファイルハンドルをしまう
278:
279: *ファイル名の指定がなかった場合は直接ここにくる（wfno = STDIN）
280: *ファイル名の指定があった場合はwfnoに出力先ハンドルが入っている
281: wopen1: move.w  wfno,-(sp)       *出力先の装置情報を取り出す
282:            clr.w   -(sp)           *
283:            DOS     _IOCTRL         *
284:            addq.l  #4,sp           *
285:
286:            move.b  d0,devflg       *devflgの第７ビットが１であれば
287:                                    *  キャラクタデバイス
288:            rts
289:
290: *
291: *       つぎの引数先頭までポインタを進める
292: *
293: nextarg:
294:            bsr     skipsp          *スペースをスキップ
```

```
295:
296:            cmpi.b  #'/',(a2)       *引数の先頭が
297:            beq     usage           *  /,-,?であれば
298:            cmpi.b  #'-',(a2)       *  使用法を表示して終了する
299:            beq     usage           *
300:            cmpi.b  #'?',(a2)       *
301:            beq     usage           *
302:
303:            rts
304:
305: *
306: *          コマンドライン先頭のスペースをスキップする
307: *
308: skpsp0:    addq.l  #1,a2           *ポインタを進め
309:                                    *繰り返す
310: skipsp:                            *サブルーチンはここから始まる
311:            cmpi.b  #SPACE,(a2)     *スペースか？
312:            beq     skpsp0          *  そうなら飛ばす
313:            cmpi.b  #TAB,(a2)       *TABか？
314:            beq     skpsp0          *  そうなら飛ばす
315:            rts
316:
317: *
318: *          引数１つ分を一時バッファにコピーする
319: *
320: getarg:
321:            lea.l   temp,a0         *a0=転送先
322: gtarg0:    tst.b   (a2)            *1)文字列の終端コードか
323:            beq     gtarg1          *
324:            cmpi.b  #SPACE,(a2)     *2)スペースか
325:            beq     gtarg1          *
326:            cmpi.b  #TAB,(a2)       *3)タブか
327:            beq     gtarg1          *
328:            cmpi.b  #'-',(a2)       *4)ハイフンか
329:            beq     gtarg1          *
330:            cmpi.b  #'/',(a2)       *5)スラッシュ
331:            beq     gtarg1          *
332:            move.b  (a2)+,(a0)+     *  が現れるまで転送を
333:            bra     gtarg0          *  繰り返す
334: gtarg1:    clr.b   (a0)            *文字列終端コードを書き込む
335:            rts
336:
337: *
338: *          使用法の表示・終了
339: *
340: usage:
341:            lea.l   usgmes,a0       *使用法
342:            bra     error
343:
344: *
345: *          カーソルが画面の左端になければ改行する
346: *
347: nl:
348:            move.w  #-1,-(sp)       *カーソル座標を取り出す
349:            move.w  #-1,-(sp)       *
350:            move.w  #3,-(sp)        *
351:            DOS     _CONCTRL        *
```

```
352:          addq.l   #6,sp            *
353:                                    *d0.l = xxxx_yyyy
354:          swap.w   d0               *d0.l = yyyy_xxxx
355:          tst.w    d0               *x座標は0か？
356:          beq      nl0              *  0ならなにもしない
357:
358: ltnl:    move.l   a0,-(sp)         * {
359:          lea.l    crlfms,a0        *改行する
360:          bsr      puterr           *
361:          movea.l  (sp)+,a0         *}
362:
363: nl0:     rts
364:
365: *
366: *         メッセージ表示（標準エラー出力へ）
367: *
368: puterr:  move.w   #STDERR,-(sp)    *標準エラー出力へ
369:          move.l   a0,-(sp)         *  文字列を
370:          DOS      _FPUTS           *  出力する
371:          addq.l   #6,sp            *スタック補正
372:          rts
373:
374: *
375: *         エラー終了
376: *
377: rerror:  lea.l    rerrms,a0        *読み込み時エラー
378:          bra      error
379: werror:  lea.l    werrms,a0        *書き出し時エラー
380:          *
381: error:   bsr      puterr           *メッセージを表示
382:
383:          move.w   #1,-(sp)         *終了コード1を持って
384:          DOS      _EXIT2           *  エラー終了
385:
386: *
387: *         メッセージデータ
388: *
389:          .data
390:          .even
391: *
392: rerrms:  .dc.b    CR,LF,'UPPER：うまく読み込めませんでした',CR,LF,0
393: werrms:  .dc.b    CR,LF,'UPPER：うまく書き出せませんでした',CR,LF,0
394: crlfms:  .dc.b    CR,LF,0
395: usgmes:  .dc.b    '使用法：UPPER [入力ファイル [出力ファイル] ] ',CR,LF
396:          .dc.b    '         半角英小文字を大文字に変換します',CR,LF
397:          .dc.b    '         出力ファイルが省略された場合は'
398:          .dc.b    '標準出力へ出力します',CR,LF
399:          .dc.b    '         入力ファイルも省略された場合は'
400:          .dc.b    '標準入力から入力します',CR,LF
401:          .dc.b    0
402:
403: *
404: *         ワークエリア
405: *
406:          .bss
407:          .even
408: *
```

```
409: rptr:    .ds.l   1               *読み込みポインタ（getcharで使用）
410: rctr:    .ds.l   1               *読み込みカウンタ（getcharで使用）
411: rfno:    .ds.w   1               *入力ファイルハンドル（getcharで使用）
412: wptr:    .ds.l   1               *書き出しポインタ（putcharで使用）
413: wctr:    .ds.l   1               *書き出しカウンタ（putcharで使用）
414: wfno:    .ds.w   1               *出力ファイルハンドル（putcharで使用）
415: *
416: devflg:  .ds.b   1               *出力先フラグ（putchar,puteofで使用）
417:                                  *       bit7=0...ファイル
418:                                  *            =1...キャラクタデバイス
419: *
420: temp:                            *一時バッファ（実体はrbuff）
421: rbuff:   .ds.b   BUFFSIZE        *読み込みバッファ（getcharで使用）
422: wbuff:   .ds.b   BUFFSIZE        *書き出しバッファ（putcharで使用）
423: *
424:          .stack
425:          .even
426: *
427: mystack:
428:          .ds.l   256             *スタック領域
429: mysp:
430:          .end
```

initでは202行以下，まずa2に無条件に1を足す。これは，文字数が格納されている1バイト分のスキップだ。つづいてサブルーチンropen，wopenを順次呼び出す。ropenは入力に使うファイルハンドルをrfnoに，wopenは出力用のファイルハンドルをwfnoに格納して戻るようにできている。ファイルを自前でオープンした場合と標準入出力を利用する場合の扱いの違いは，これらサブルーチンがすべて吸収している。

ropenは，最初にサブルーチンnextargを呼び出す(216行)。nextargはスペースをスキップしてからa2が指す1文字を調べ，〝/〟，〝-〟，〝?〟であればusageに分岐する。usageは使用法を表示して，即座にプログラムの実行を終える処理を受け持つ[18]。

■18) サブルーチンを数段階に呼び出した状態で強制終了させているのはあまりよいことではないが，この場合はいちいちエラーを返したりすると，プログラムがかえってこんがらがるような気がしてこのようにしてある。

nextargから戻った時点でa2は文字列の終わりか，もしあれば第1引数の先頭を指しているはずだから，tst命令で引数があるかどうか(a2が指しているのが文字列のエンドコード00_Hであるかどうか)を調べる(217行)。文字列が続いていれば，それはファイル名の指定だから，サブルーチンgetargを呼び出し(221行)，ファイル名だけをいったんラベルtempで示された領域へ切り出す(このとき，同時にa2がファイル名の直後を指すように更新される)。このコピーしたファイル名を使ってDOSコールopenによりファイルを読み込みモードでオープンする(223～226行)[19]。

■19) getarg内でファイル名を一時的に切り出す領域tempの実体は，データ読み込みバッファrbuffである。rbuffは，引数取り込みの途中

ではまだ本来の目的では使われていないのでどさくさに兼用している。

ファイル名の指定がない場合は，標準入力のファイルハンドルをコピーする(230～232行)。どちらの場合もファイルハンドルはrfnoに格納し(240行)，標準入力はクローズしてキーボードに割り当て戻す(242～244行)。これにより，とにかく入力先のファイルハンドルはrfnoに格納されることになり，以後の入力はこのrfnoに入っているファイルハンドルを使って行えばよい。

wopenでは，nextargを呼び出してスペースを飛ばして(256行)から出力ファイルが指定されているかどうかを調べる(257行)。あれば入力のときと同様にgetargでファイル名を切り出し(261行)，さらにDOSコールcreateでファイルを新規作成する(263～266行)。

ここからの動作はちょっと凝っている。createがエラーを返しても，すぐにエラー終了してしまわずに，DOSコールopenで書き込みモードでオープンし直す(270～273行)ようになっているのだ。

これは，

```
A>UPPER FILE1 CON
```

のように，出力ファイル名の指定が〝CON〟や〝PRN〟などのデバイス名だった場合に対する配慮だ。CONやPRNは，デバイスをファイルのように扱うための方便として用意されている名前であり，本当はファイルではないので〝新規作成〟できるはずがない。CONやPRNをcreateでオープンしようとすると，〝ファイル名の指定に誤りがある〟というエラー(エラー番号-13)が返ってくる。そこでcreateがエラーを返した場合には，DOSコールopenを使ってオープンし直すようにした。出力先がCONなどではなく，その他のエラーだった場合には，openでもう一度オープンしてもやっぱりエラーになるはずであり，この時点で真のエラーが検出できる[20]。

■20) 別の考え方としては，ファイル名がCONかどうかを文字列レベルで比較する方法もないではない。が，デバイス名は他にもあるし，今後いくらでも新しいデバイスが出てくることが考えられるので，万全の方法とはいえない。

さて，正常にオープンできたらファイルハンドルをwfnoにしまっておく(277行)。また，出力ファイル名が指定されていない場合には，254行でwfnoにSTDOUTを代入しているのがそのまま有効になる。どちらにしろ，以後の出力はwfnoに入っているファイルハンドルを使えばよいわけだ。

ropen，wopenから戻ったら，最後にふたたびスペースをスキップし，コマンドライン引数文字列が終わっているかどうか調べる。まだあるようなら，引数の数が多すぎることになり，使用法を表示するためにusageに飛ぶ。

課題

フィルタUPPER.Xもようやく完成した。完成版は400行を超え(といっても，かなりスカスカしたソースだが)，初期のいいかげんバージョンとは似ても似つかないものになった。もし最初からこれだけのものを作ろうとしていたら，途中で投げ出してしまったかもしれない。これは，今後もっと大きなプログラムを作るうえでの教訓になるだろう。

UPPER.Xは，フィルタとしてはかなりがっちり作ってある。1度作ったプログラムのパーツは後で別のプログラムにそのまま流用することができるわけであり，UPPER.Xもメイン部分(サブルーチンdo)だけを差し替えることで容易に別のフィルタに作り替えることができる。結局，この章で作ったのは，ある固定動作をするプログラムではなく，使い回しのきく〝フィルタの雛型〟であったのだ。

さて，いくつか課題を出しておくので，漠然とでもいいから考えてみてもらいたい。

1) リスト8をアセンブルすることで生成される実行ファイルは1244バイトもの大きさになる。これは，リスト8よりずっと複雑なUPPER.Xの最終版が1138バイトなのと比べると必要以上に大きく見える。理由を考え，対処しなさい。ちなみに，リスト8の実行ファイルは後1024バイト短くなるはずである。

2) 半角英大文字を小文字に変換するフィルタLOWER.Xを作りなさい。

3) UPPER.Xでは，プログラムの読みやすさや安全性を高めるためによけいな処理をしている部分が数多くある。これをぎりぎりまで最適化することでプログラムはどれだけ短くなるか，またどれだけ処理が速くなるかを検討しなさい。そもそも，そのような最適化は必要なのだろうか。

4) UPPER.Xのバッファの大きさ(記号定数BUFFSIZE)をいろいろな値に変更し，処理時間がどれだけ変わるか試しなさい。極端に大きなバッファを割り当てた場合，それに見合うだけの高速化ははかれたか。

5) UPPER.Xで使ったサブルーチンgetcharとputcharは入出力が複数系統ある場合には使えない。それはなぜか。また，どうすれば複数の入出力に対応できるか。

6) サブルーチンgetcharを使ってファイルから1行分入力するサブルーチンを作りなさい。どんな場合にも正確に1行分取り込むサブルーチンは作れたか。それともなにか制約はあるか。その制約は回避可能か。妥協するとしたらどこで妥協するか。

7) UPPER.Xと同等の処理を行うCのプログラムを作成し，そのソースの大きさ，オブジェクトの大きさ，制作にかかった時間と手間，完成したプログラムの実行時間をマシン語で作ったものと比べなさい。その結果，この規模のフィルタをマシン語で作ることに意味があるかどうかを考えなさい。

8) UPPER.Xのような完成した雛型を改造してべつにフィルタを作る場合はどうか。

9) プログラムを作る立場ではなく，ただ使うだけの立場であればどうか。

CHAPTER 7

コマンド作成〝基本〟作法

CHAPTER 7

コマンド作成"基本"作法

ASSEMBLER

前章では若干突っ走りすぎた(陳謝!)ので，この章では比較的軽め(かな？)の話題を取り上げよう。プログラムを〝使いにくくしないためにはどうすればいいか〟というノリで押してみようと思う(〝使いやすくする〟じゃないのが凄いでしょ)。プログラムにもいろいろあるわけだが，今回は，起動時に指示を全部与え，「ほら行け」と尻を叩くと勝手に処理をして帰ってくるようなプログラムを想定する。要するに，COMMAND.Xのプロンプトに続いてプログラム名と若干の引数を打ち込み，リターンキーを押すと後はユーザーに指示を仰ぐことなく処理を行って終了するようなプログラムだ[1]。

■ 1) あらかじめ指示を全部与えてプログラムを走らせることを〝非対話的〟とか〝バッチジョブ的〟という言葉で表現する。逆に，プログラムを立ち上げてからユーザーの指示とプログラムの動作が交互に行えるようなものは〝対話的〟という。対話的というのも変な言葉だが，interactiveの訳らしい。

この手のプログラムでは，ユーザーの指示はコマンドラインに打ち込んだ文字列という形でプログラムに伝えられる。プログラムはそれを解析・解釈することでユーザーが何をしたいのかを知る。コマンドライン文字列の解析処理ルーチンの作り方によっては使いにくいプログラムになってしまうこともありうるわけだ。そこで，このコマンドラインの扱い方を考えるのが，この章での話の中心になる。

さて，たかがコマンドライン文字列解析とはいうものの，それなりにいろいろと考えさせられる要素はある。マンマシンインタフェイスという言葉を持ち出せば，事の重大さがわかろうというものだ。

昔は自分の作ったソフトウェアを世に出そうと思ったら，雑誌に投稿するか，ソフトハウスに売り込むか，自分でソフトハウスを作って売るか，街頭でプログラムテープを配りまくるかぐらいしか手段がなかった。しかし，最近はパソコン通信などを通じて公開ソフトとして自由にばらまけるようになった。せっかく作ったプログラムを自分ひとりで使うのはもったいないし，X68000の環境を整える意味でも，機会があればどんどん自作ソフトをばらまいてもらいたいと思う。その場合は，ユーザーの立場に立った細かな配慮といったものも必要になってくるだろう。

そういう希望もあって，本章の話はいちおう自作ソフトを他人に使ってもらうことを前提で

進めたい。

こんなプログラムは使いたくない

コマンドライン文字列の解釈ルーチン作成は，そのプログラムを使ううえでの書式を設計することから始まる。ここでいう書式とは，どんなパラメータを，いくつ，どんな順番で，どんな形式で指定するかといったことだ。どのように書式を決めたら使いやすくなるかは一概にいえるものではなく，ユーザーが他にどんなプログラムを使っているかとか，キータイプの癖とか，好みの問題も絡んでくる。しかし，客観的に見て明らかに使いにくい形というのはいくつか思いつくから，〝そうしない〟ようにすれば，とりあえず〝使いにくくないプログラム〟にはなるだろう。

以下，よくない例を3つほど挙げてみたい。

1）OSの標準コマンドの書式と極端に違う

プログラムごとに使い方が全然異なるようではユーザーに不要な負担をかけることになるから，使用法はある基準にしたがって統一されることが望ましい。何を基準とするかは意見が分かれるところかもしれないが，OSのコマンドプロセッサ(Human68kならば COMMAND.X)の内部コマンドの書式や標準外部コマンドの使用法に準じるのがもっとも自然だろう。

たとえば，Human68kでは複数のパラメータがある場合は，スペースで区切って並べるようになっているのに，〝,〟で区切ることにしたり，スイッチは〝/A〟のように指定するのがふつうであるにもかかわらず〝$A〟などの見慣れない表記を用いることは避けるべきである。

また，複数のパラメータがある場合の語順にも同じことがいえる。入力ファイルと出力ファイルの両方を指定する場合，copyコマンドなどHuman68k上の大部分のプログラムでは，

```
A>PROG 入力 出力
```

の形式を用いる。しかし，これを，

```
A>PROG 出力 入力
```

のように通常と逆の順にすることはユーザーを混乱させるので避けたい。

2）キータイプ量が必要以上に多い

もともとキーボードから文字列を打ち込むという行為自体が面倒がられるものだから，入力する文字列の長さはなるべく短くてすむようにしたい。スイッチにはたいてい英単語の頭1文字が使われるが，これをフルスペルで書くように強制するとか，ファイル名をフルパスで指定しなければならないとか，扱うファイルの拡張子が限定されているにもかかわらず省略が許されないといったプログラムは，使っているうちにいらいらしてくるに決まっている。

3）スイッチが覚えにくい

スイッチがたくさんあると，どのスイッチがどんな意味だったかわからなくなるものだ。通常，スイッチには意味のある英単語の頭文字が使われることが多いものだ。いうまでもないこ

とだが，機能と関連のないでたらめな文字を使うのは避けたほうがよい。

また，趣味のプログラマはえてしてよけいなスイッチをつけたがる傾向があるが，それぞれのスイッチが本当に必要かどうかも一度真剣に検討してみる必要があるだろう。

ちなみに，それぞれのスイッチはプログラムのほんの細かな部分の動作を制御するために用い，1つのスイッチを指定することによってプログラムの動作が大幅に変わるのは好ましくない(非常に極端な例を挙げると，スイッチを何も指定しない状態ではファイルをコピーするプログラムなのに，/Dスイッチを指定するとファイルを消去するようになる，とか)。そういう場合は，機能別に2本のプログラムに分割したほうが使いやすいというものだ。

さらに，どんなにわかりやすく書式を設定したとしても，ユーザーが暗記するのを期待するのではなく，かんたんなヘルプ表示機能を用意することが望ましい。

こうやって言葉にしてみると当たり前のことばかりだが，以上の3点を〝べからず集〟の最上位にランクすることに読者も異存はないだろう。このべからず集に抵触しないことを，プログラムを使いにくくしないための最低限の基準にしたいと思う。

引数指定の注意点

書式を決めたら，後はそれにしたがって実際の引数処理ルーチンを作るわけだ。ここでの注意点は，とにかく〝融通をきかす〟ということにつきる。

たとえば，引数間の空白にしても，バッチの中から起動する場合などに備えてスペース文字(ASCIIコード20_H)だけではなく，TABコード(09_H)も空白と見なすとか，空白はただ1つしか許さないのではなく，いくつあってもかまわないように作るとか，逆にファイル名とスイッチの間のように区切りが明らかな場合は空白の省略を認める，といった細工の余地がある。

スイッチに関してもHuman68kの標準形式である〝/A〟だけではなく，〝-A〟の形式(これはHuman68kの準標準形式といえる)も認め，また，大文字・小文字のどちらで指定しても受け付けるように作りたい。

さらに，スイッチと他の引数とを並べる順序にも，ある程度の自由度が求められる。つまり，

```
A>PROG /A FILENAME
```

と指定しても，

```
A>PROG FILENAME /A
```

と指定しても受け付ける柔軟さがほしい。

後，さきほども少しふれたように，そのプログラムで扱うファイルの拡張子が限定されている場合には拡張子の省略を認めるといった配慮も必要だし，もっと一般的なファイルを扱うプログラムの場合はワイルドカードも使えたほうがいいだろう。

プログラミングの実際

さて，プログラム例に入る前に確認の意味でコマンドライン文字列の構造を確認しておく。

前章でも話したように，Human68kではコマンドライン文字列はレジスタを介してプログラムに渡され，プログラム起動時のa2レジスタがこのコマンドライン文字列を指している。先頭の1バイトが文字列の長さを表し，その直後から00_Hが現れるまでが実際の文字列だ。このことからわかるように，コマンドライン文字列の最大長は255バイト(+00_Hの1バイト分)になる。

文字列は，ユーザーがコマンドラインに打ち込んだものから先頭のコマンド名(ファイル名)とそれに続くブランクを除いた部分がそのまま入る。引数間の空白の数なども変わらず，末尾の余分なスペースすら残っている。なお，COMMAND.Xでリダイレクションやパイプラインを表すのに使われる文字，〝<〟，〝>〟，〝|〟以降はプログラムには渡されないことになっている。ただし，「〟」か「'」で囲んでおくと，その間は無条件にプログラムに渡されることになっており，これを利用すればリダイレクト記号なども引数に含むことができる[2)]。

■ 2) 引数を「〟」や「'」で囲んだ場合は，これらの記号そのものもコマンドライン文字列に含まれるので，プログラム側で取り除く必要がある。

リスト1は，こういったコマンドライン文字列の構造を確認するプログラムだ。与えられたコマンドライン文字列全体を〝[〟と〝]〟で囲んで表示する。図1の実行例を見てもらうと，リダイレクト記号以下が削られていることや，末尾のスペースがそのまま残っていることが確認できるだろう。コマンド名と最初の引数の間の空白が，その数にかかわらず削られているということも発見できる。

リスト1
PRINT.S

```
 1: *       コマンドライン引数の渡されかたを確認する
 2:
 3:         .include        doscall.mac
 4:         .include        const.h         *前章で用意した定数定義ファイル
 5: *
 6:         .text
 7:         .even
 8: *
 9: ent:
10:         lea.l   mysp,sp         *spの初期化
11:
12:         move.w  #'[',-(sp)      *[を表示
13:         DOS     _PUTCHAR        *
14:         addq.l  #2,sp           *
15:
16:         pea.l   1(a2)           *a2+1からの
17:         DOS     _PRINT          *  コマンドライン文字列を
18:         addq.l  #4,sp           *  表示
19:
```

```
20:          pea.l   endmes          *]を表示して改行する
21:          DOS     _PRINT          *
22:          addq.l  #4,sp           *
23:
24:          DOS     _EXIT           *終了
25: *
26:          .data
27:          .even
28: *
29: endmes:  .dc.b   ']',CR,LF,0
30: *
31:          .stack
32:          .even
33: *
34: mystack:
35:          .ds.l   256             *スタック領域
36: mysp:
37:          .end
```

```
A>print test TEST test↵
[test TEST test]
A>print    test   TEST   test↵
[test   TEST   test]
A>print test TEST test       ↵
[test TEST test      ]
A>print test              >CON↵
[test               ]
A>print                   >CON↵
[]
A>print "<test>"↵
["<test>"]
```

図1
リスト1実行例
(↵はリターンキーを押した位置)

この構造を頭に入れたうえで，以下，示すような引数の数に応じたコマンドライン引数取り込み処理ルーチン例を見てほしい。

引数を1個も必要としない場合(リスト2)

この場合は，コマンドライン文字列を頭っから無視してもかまわないわけだが，引数が何もないことを確認したほうがいろいろな意味で安全だと思われる。もしなんらかの引数がある場合にはプログラムの使用法を表示し，引数を必要としないことをユーザーに教えよう。ヘルプ内では，このプログラムが何をするプログラムで，どういう書式で使うか(この場合は，引数が何もなく，たんにコマンド名だけで起動できるということ)を示せば十分だ。

引数の有無がプログラムの実行に影響しない場合にも，わざわざ引数があるかどうかをチェックしているのには2つの意味がある。1つは上でも述べたように安全のためであり，ユーザ

一の勘違いやミスタイプがもたらす(かもしれない)事故を防ぐという目的，もう1つは他のプログラムとの統一性という問題だ。たいていのプログラムでは，意味のないスイッチ(よく使われるのは〝/?〟)が指定されると使用法を表示する慣例になっており，プログラムをはじめて使うときや使い方を忘れたときには無意識のうちに

```
A>PROG /?
```

とタイプする人も多いと思う。どんなプログラムでも〝/?〟が指定されたらヘルプを表示するように統一されていれば，ユーザーは安心してプログラムを使うことができるというわけだ。ただ，現実にはHuman68kに用意されたプログラムでも使用法を表示してくれないものもあるが[3]。

■ 3) Human68k上のプログラムでヘルプのない代表例はCOMMAND.Xで，起動スイッチのヘルプも内部コマンドのヘルプもない。COMMAND.Xの起動スイッチなどは，一度CONFIG.SYSで指定してしまえば後はめったに使うものではないが，だからこそヘルプが必要なのだ。また，ヘルプがあっても意味をなしていないのがED.Xで，起動スイッチのヘルプがどういうわけか編集コマンドのヘルプの奥底に埋もれている。基本的なところで勘違いしているといわざるをえない悪い見本である。

リスト2では，コマンドラインの解析はサブルーチンchkargで行っている。まず，a2に1を足して，a2が文字列の先頭を指すようにする。それからサブルーチンskipspを呼び出して引数文字列先頭の空白を読み飛ばす。リスト1で判明したように，実際には引数文字列先頭の空白はあらかじめ削られているはずだが，これは明文化された規則というわけではないので万が一に備えてこういった処理を挟み，空白が絶対にないことを保証するわけだ。

リスト2
ARG0.S

```
 1: *       コマンドライン引数解析
 2: *       引数を必要としない場合
 3:
 4:         .include        doscall.mac
 5:         .include        const.h
 6: *
 7:         .text
 8:         .even
 9: *
10: ent:
11:         lea.l   mysp,sp         *spの初期化
12:
13:         bsr     chkarg          *コマンドラインの解析
14:
15:         bsr     do              *メイン処理
16:
17:         DOS     _EXIT           *正常終了
18:
19: *
20: *       メイン処理（今は何もしない）
21: *
22: do:
```

```
23:          rts
24:
25: *
26: *        コマンドラインの解析
27: *
28: chkarg:
29:          addq.l  #1,a2           *a2=コマンドライン文字列先頭
30:          bsr     skipsp          *スペースをスキップする
31:          tst.b   (a2)            *余計な引数があるか？
32:          bne     usage           *  そうなら使用法を表示
33:          rts
34:
35: *
36: *        コマンドライン先頭のスペースをスキップする
37: *
38: skpsp0: addq.l  #1,a2            *ポインタを進め
39:                                  *繰り返す
40: skipsp:                          *サブルーチンはここから始まる
41:          cmpi.b  #SPACE,(a2)     *スペースか？
42:          beq     skpsp0          *  そうなら飛ばす
43:          cmpi.b  #TAB,(a2)       *TABか？
44:          beq     skpsp0          *  そうなら飛ばす
45:          rts
46:
47: *
48: *        使用法の表示＆終了
49: *
50: usage:
51:          move.w  #STDERR,-(sp)   *標準エラー出力へ
52:          pea.l   usgmes          *  ヘルプメッセージを
53:          DOS     _FPUTS          *  出力する
54:          addq.l  #6,sp           *スタック補正
55:
56:          move.w  #1,-(sp)        *終了コード１を持って
57:          DOS     _EXIT2          *  エラー終了
58:
59: *
60: *        メッセージデータ
61: *
62:          .data
63:          .even
64: *
65: usgmes: .dc.b   '機　能：○○を××します',CR,LF
66:          .dc.b   '使用法：ARGO',CR,LF
67:          .dc.b   0
68: *
69:          .stack
70:          .even
71: *
72: mystack:
73:          .ds.l   256             *スタック領域
74: mysp:
75:          .end
```

その後でa2が指している(スペースでもTABでもない)文字が文字列の終端コード00_Hかどうかをtst命令で調べる。Zフラグが立てば，文字列が何もないことになるから解析を終了し，メインルーチンに戻る。Zフラグが立たなかった場合は，なんらかの文字列があったことになるので，使用法を表示するためにラベルusageに分岐する。usage以下では，使用法を表示してプログラムを終了する。ヘルプメッセージは，ラベルusgmes以下に用意している。

ところで，ヘルプメッセージは標準エラー出力に出すようにした。標準出力がリダイレクトされた場合にも，ヘルプメッセージが画面に表示されるようにしてあるわけだ。しかし，現実には〝ヘルプメッセージをファイルに落としたい〟といった要求もあるから，ヘルプメッセージを標準出力に出すか標準エラー出力に出すかは考えどころかもしれない。馬鹿らしいほど些細なことではあるが，大勢の人の意見を聞いてみたい気もする。

また，使用法を表示した後，終了コード1を持って終了し，処理が正しく行われなかったことを呼び出し側に知らせるようにしてある。ヘルプはエラーではないという考え方もあるだろうが，一般に正しく処理が行われた場合のみ終了コード0で正常終了し，それ以外は1以上の終了コードを返したほうがユーザーのためになることが多いと思う。

引数を1個必要とする場合(リスト3)

必ずただ1個の引数(とりあえずファイル名ということにしよう)をともなって起動するプログラムの場合，チェック項目は3点ある。第1に引数があるかどうか，第2に引数が正当かどうか，第3によけいな引数がないかどうかだ。いずれの場合もチェックに引っかかったら，使用法か適切なエラーメッセージを表示して終了する。

引数があるかどうか調べるところまではリスト2と変わらない。引数があった場合は例によって〝/?〟かもしれないので，スイッチのチェックをする。このプログラムの場合は，スイッチが何もないわけだから〝/〟か〝−〟が指定されたら即使用法の表示に飛んでよいだろう。

次に，引数1つ分を適当なメモリ領域にコピーする。これは，引数の後によけいな空白などがついている可能性に備えてのことだ。なお，引数は最大255バイトだから，転送領域はそれにエンドコードの1バイトを足した256バイト以上を確保しておく必要がある。この引数の切り出し処理を行っているのがサブルーチンgetargで，このサブルーチンはa2が指すアドレスからスペースかTABか，〝/〟か〝−〟か00_Hが現れる直前までを1つの引数と見なして，a0でポイントしたメモリ領域(リストではラベルarg以下)へ転送する。扱いやすいように最後に文字列のエンドコード00_Hを書き込むのを忘れてはならない。

この時点でarg以下には引数1個分が切り出され，同時にa2は引数の直後を指すように更新されているので，a2の位置からさらにスペースを飛ばし，まだ引数があるかどうかを調べ，もしあるようなら(引数の個数が多すぎるので)，使用法の表示処理に分岐する。引数の個数があっていることが確認できたら，今度はその引数の正当性を調べる。どんな文字列を正当な引数と見なすかはプログラムによるわけだが，ここではファイル名である場合を取り上げる。

ある文字列がDOSの規則にしたがったファイル名かどうかを調べるのは真面目にやろうとするとたいへんなことなので，ここはDOSに頼ろう。指定された文字列をとにかくファイル名だと見なして，DOSコールで読み込みモードでオープンしてみるのはてっとり早い方法だ。ファイルがオープンできなかったらDOSコールのエラー番号を調べれば，ファイル名に異常があったかどうかがわかる。

リスト3
ARG1.S

```
 1: *       コマンドライン引数解析
 2: *       引数としてファイル名を１個必要とする場合
 3:
 4:         .include        doscall.mac
 5:         .include        const.h
 6: *
 7:         .text
 8:         .even
 9: *
10: ent:
11:         lea.l   mysp,sp         *spの初期化
12:
13:         bsr     chkarg          *コマンドラインの解析
14:
15:         bsr     do              *メイン処理
16:
17:         DOS     _EXIT           *正常終了
18:
19: *
20: *       メイン処理（今は与えられた引数を表示するだけ）
21: *
22: do:
23:         pea.l   arg             *引数を表示する
24:         DOS     _PRINT          *
25:         addq.l  #4,sp           *
26:         bsr     crlf            *改行する
27:         rts
28:
29: *
30: *       改行する
31: *
32: crlf:
33:         pea.l   crlfms
34:         DOS     _PRiNT
35:         addq.l  #4,sp
36:         rts
37:
38: *
39: *       コマンドラインの解析
40: *
41: chkarg:
42:         addq.l  #1,a2           *a2=コマンドライン文字列先頭
43:         bsr     skipsp          *スペースをスキップする
44:         tst.b   (a2)            *引数があるか？
45:         beq     usage           *  ないなら引数が足らない
```

```
 46:
 47:          cmpi.b   #'/',(a2)        *引数の先頭が
 48:          beq      usage            *  '/' か
 49:          cmpi.b   #'-',(a2)        *  '-' であれば
 50:          beq      usage            *  きっとヘルプが見たいのだろう
 51:
 52:          lea.l    arg,a0           *a0=引数切り出し領域
 53:          bsr      getarg           *引数１つをa0以降に取り出す
 54:
 55:          bsr      skipsp           *さらにスペースをスキップ
 56:          tst.b    (a2)             *引数があるか？
 57:          bne      usage            *  あるなら引数が多い
 58:
 59:          pea.l    nambuf           *DOSコールを使って
 60:          move.l   a0,-(sp)         *  ファイル名を
 61:          DOS      _NAMECK          *  展開してみる
 62:          addq.l   #8,sp            *
 63:          tst.l    d0               *d0が０でなければ
 64:          bne      usage            *  ファイル名の指定に誤りがある
 65:
 66:          rts
 67:
 68: *
 69: *        a2の指す位置から引数１つ分を
 70: *        a0の指す領域へコピーする
 71: *
 72: getarg:
 73:          move.l   a0,-(sp)         * {レジスタ待避
 74: gtarg0:  tst.b    (a2)             *1)文字列の終端コードか
 75:          beq      gtarg1           *
 76:          cmpi.b   #SPACE,(a2)      *2)スペースか
 77:          beq      gtarg1           *
 78:          cmpi.b   #TAB,(a2)        *3)タブか
 79:          beq      gtarg1           *
 80:          cmpi.b   #'-',(a2)        *4)ハイフンか
 81:          beq      gtarg1           *
 82:          cmpi.b   #'/',(a2)        *5)スラッシュ
 83:          beq      gtarg1           *
 84:          move.b   (a2)+,(a0)+      *  が現れるまで転送を
 85:          bra      gtarg0           *  繰り返す
 86: gtarg1:  clr.b    (a0)             *文字列終端コードを書き込む
 87:          movea.l  (sp)+,a0         *} レジスタ復帰
 88:          rts
 89:
 90: *
 91: *        コマンドライン先頭のスペースをスキップする
 92: *
 93: skpsp0:  addq.l   #1,a2            *ポインタを進め
 94:                                    *繰り返す
 95: skipsp:                           *サブルーチンはここから始まる
 96:          cmpi.b   #SPACE,(a2)      *スペースか？
 97:          beq      skpsp0           *  そうなら飛ばす
 98:          cmpi.b   #TAB,(a2)        *TABか？
 99:          beq      skpsp0           *  そうなら飛ばす
100:          rts
101:
102: *
```

```
103: *       使用法の表示＆終了
104: *
105: usage:
106:         move.w  #STDERR,-(sp)    *標準エラー出力へ
107:         pea.l   usgmes           *  ヘルプメッセージを
108:         DOS     _FPUTS           *  出力する
109:         addq.l  #6,sp            *スタック補正
110:
111:         move.w  #1,-(sp)         *終了コード１を持って
112:         DOS     _EXIT2           *  エラー終了
113:
114: *
115: *       メッセージデータ
116: *
117:         .data
118:         .even
119: *
120: usgmes: .dc.b   '機　能：指定ファイルを××します',CR,LF
121:         .dc.b   '使用法：ARG1 ファイル名'
122: crlfms: .dc.b   CR,LF,0
123:
124: *
125: *       ワークエリア
126: *
127:         .bss
128:         .even
129: *
130: arg:    .ds.b   256              *引数切り出し用バッファ
131: nambuf: .ds.b   91               *ファイル名展開用バッファ
132: *
133:         .stack
134:         .even
135: *
136: mystack:
137:         .ds.l   256              *スタック領域
138: mysp:
139:         .end
```

もう少しスマートにやりたければ，DOSコール＄FF29のnamestsか，＄FF37のnameckかを使う手がある。どちらのDOSコールもファイル名をドライブ名，パス名などに展開するもので，返ってくる情報の形が少し違う以外は同じような機能と考えてよい。展開の過程でファイル名の形式にあわない部分が見つかったらエラーを返すようになっているから，ファイル名の正当性を調べるのにも利用できる。ここではnameckのほうだけを解説しておこう。

nameckは，次のようにして呼び出す。

```
move.l    展開バッファアドレス,－(sp)
move.l    ファイル名へのポインタ,－(sp)
DOS       _NAMECK
addq.l    #8,sp
```

展開バッファは91バイト(以上)確保しておく。呼び出し後，このメモリ領域に図2に示すような形式でファイル名が展開される[4]。相対パスでファイル名を指定した場合も絶対パスに展

ドライブ名	2バイト	'A:','B:'のような形のドライブ名
パス名	65バイト	'¥'から始まり'¥'で終わる絶対パス＋00H
ファイル名	19バイト	18バイトまでのファイル名＋00H
拡張子	4バイト	'.'＋3バイトまでの拡張子＋00H

図2
nameckが返すファイル情報の形式

開されることになっている。このDOSコールはワイルドカードにも対応しており，〝*〟が指定されると適切な数の〝?〟に置き換える。

■ 4)図2を見ると，Human68kの(ドライブ名，ファイル名を除いた)パス名の最大長が64文字までであることがわかる。パス名がこれより長くなるような深い階層ディレクトリは構築できないということだ。Human68kのマニュアルには階層ディレクトリの深さの制限はないような書き方がしてあるが，現実にはこのような部分で目に見えない制限がある。これはサブディレクトリに短い名前をつけるか，長い名前をつけるかによって，階層の深さの限界が変わってくることを意味している。さっそく実験してみよう。

指定したファイル名が正しかったかどうかは，リターン時のd0.lを調べることでわかる。例によって，d0.lが負の数であればエラーであり，ファイル名が正しくなかったことを表す。また，d0.lがFF_H(=255)であれば，ファイル名が指定されなかった(たとえば，ドライブ名のみが指定されたとか)ことを表し，0であればファイル名がちゃんとあり，かつ，ワイルドカードも指定されなかったことを表す。d0.lが負でも，0でも，FF_Hでもなかった場合は，ワイルドカードが指定されたことを意味する。

結局，たんにファイル名の正当性を調べたい場合で，かつワイルドカードも認めないのであれば，d0.lが0かどうかだけを見ればよいことになる。

nameckの動作を確認する(リスト4)

さて，nameckにはこれからもお世話になるので，より細かい動作を調べておこう。リスト4がテスト用プログラムになっている。与えられたコマンドラインパラメータをそのままnameckに渡し，戻り値がいくつか(16進表記)，どのように展開されたかを表示する。

リスト4では初登場の命令を2つ使っている。最初のdbraはループ制御用の命令で，なぜ，いまごろ出てきたのかというほど使用頻度が高い命令だ。すかさず頭に入れておこう。というわけで，詳しくはコラム参照のこと。もう1つはrolという命令なのだが，これに関してはべつの機会に取り上げたほうがよさそうなので，今回はリスト中の注釈で勘弁してもらいたい。

リスト4
NAMETEST.S

```
1: *         DOSコールnameckの動作を確認するプログラム
2:
```

```
 3:          .include        doscall.mac
 4:          .include        const.h
 5: *
 6:          .text
 7:          .even
 8: *
 9: ent:
10:          lea.l   mysp,sp         *spの初期化
11:
12:          pea.l   namebuf         *与えられたコマンドライン引数を
13:          pea.l   1(a2)           *  ファイル名と見なし
14:          DOS     _NAMECK         *  nameckで展開する
15:          addq.l  #8,sp           *
16:
17:          lea.l   hexbuf,a0       *d0.lを16進８桁に変換し
18:          bsr     itoh            *  a0の指す領域へ格納しておく
19:
20:          lea.l   prttbl,a0       *a0=テーブルの先頭アドレス
21:
22:          moveq.l #4-1,d1         *以下を４回繰り返す
23:
24: loop:    move.l  (a0)+,-(sp)     *見出し部分を表示
25:          DOS     _PRINT          *
26:          addq.l  #4,sp           *
27:
28:          move.l  (a0)+,-(sp)     *対応する内容を表示
29:          DOS     _PRINT          *
30:          addq.l  #4,sp           *
31:
32:          pea.l   crlfms          *改行する
33:          DOS     _PRINT          *
34:          addq.l  #4,sp           *
35:
36:          dbra    d1,loop         *d1.wが-1になるまで繰り返す
37:
38:          DOS     _EXIT           *正常終了
39:
40: *
41: *       d0.lを16進８桁を表す文字列へ変換し(a0)以降に格納する
42: *               レジスタはみんな保存する
43: *
44: itoh:
45:          movem.l d0-d2/a0,-(sp)  * {レジスタ待避
46:
47:          moveq.l #8-1,d2         *以下を８回繰り返す
48:
49: itoh0:   rol.l   #4,d0           *d0.lを左に４ビット回転する
50:                                  *４ビットは16進１桁分！
51:                                  *たとえば
52:                                  *  $1234ABCD  →  $234ABCD1
53:          move.b  d0,d1           *d0の下位バイトをd1に取り出し
54:          andi.b  #$0f,d1         *  下位４ビットを残してマスクする
55:                                  *d1にはd0.lを16進で表したときの
56:                                  *  最上位桁が入っている
57:          addi.b  #'0',d1         *ここで数値から16進を表す文字へ
58:          cmpi.b  #'9'+1,d1       *  変換する
59:          bcs     itoh1           *  0～9の場合は'0'を足すだけだが
```

```
 60:          addq.b  #'A'-'0'-10,d1  *  A～Fの場合はさらに補正が必要
 61:
 62: itoh1:   move.b  d1,(a0)+        *変換した文字をしまう
 63:
 64:          dbra    d2,itoh0        *d2.wが-1になるまで繰り返す
 65:
 66:          clr.b   (a0)            *文字列終端コードを書き込む
 67:
 68:          movem.l (sp)+,d0-d2/a0  *} レジスタ復帰
 69:          rts
 70:
 71: *
 72: *        データ
 73: *
 74:          .data
 75:          .even
 76: *
 77: prttbl: .dc.l   mes1,hexbuf      *見出しとその内容の
 78:          .dc.l   mes2,drive      *  対応を示したテーブル（表）
 79:          .dc.l   mes3,name       *  見出しのアドレスと内容のアドレスが
 80:          .dc.l   mes4,ext        *  1組になっている
 81: *
 82: mes1:   .dc.b   'NAMACKの戻り値(d0.l):',0
 83: mes2:   .dc.b   '          パス名:',0
 84: mes3:   .dc.b   '      ファイル名:',0
 85: mes4:   .dc.b   '        拡張子:',0
 86: crlfms: .dc.b   CR,LF,0
 87:
 88: *
 89: *        ワーク
 90: *
 91:          .bss
 92:          .even
 93: *
 94: namebuf:                         *nameckで
 95: drive:  .ds.b   2                *  ファイル名が展開される
 96: path:   .ds.b   65               *  領域
 97: name:   .ds.b   19               *
 98: ext:    .ds.b   5                *計91バイト
 99: *
100: hexbuf: .ds.b   8+1              *itohで16進文字列を格納する領域
101: *
102:          .stack
103:          .even
104: *
105: mystack:
106:          .ds.l   256            *スタック領域
107: mysp:
108:          .end
```

COLUMN

dbra命令

dbra命令は,マシン語にしてはめずらしく,用途のはっきりした高級言語指向の命令で,一定回数繰り返すループを構成するのに用いられる。dbraの頭のdはDecrementのdであ

り，braは〝あの〟braである。

dbra　データレジスタ，分岐先

のようにして使い，指定したデータレジスタから１を引き(レジスタの内容を更新して)，結果が−1であればdbraの直後の命令の実行に移り，そうでなければ指定した分岐先に制御を移す。サイズはつねにワードであり，データレジスタの下位ワードのみが使用される。このため，dbra１つでは65536回までのループしか構成できない。

標準的な使い方は，たとえば次のようになる。

```
    move.w      #1000−1,d0
loop:       繰り返す処理
            :
    dbra        d0,loop
```

この例では，d0.wをループカウンタとして使い，1000回同じ処理を繰り返す。dbraはデータレジスタが−1になるまで分岐を繰り返すので，ループカウンタに使うレジスタはループ回数より１小さい値で初期化しておかなければならない。なお，上の例で

```
    move.w  #999,d0
```

ではなく，

```
    move.w  #1000−1,d0
```

という書き方をしているのは，1000回のループであることを強調しているのだと思ってもらいたい。

引数を２個必要とし，オプションとして/A，/Bの２つのスイッチがある場合(リスト５)

リスト5
ARG2.S

```
 1: *       コマンドライン引数解析
 2: *       引数としてファイル名を２個必要とし
 3: *       /A,/B２つのスイッチを持つ場合
 4:
 5:         .include        doscall.mac
 6:         .include        const.h
 7: *
 8:         .text
 9:         .even
10: *
11: ent:
12:         lea.l   mysp,sp         *spの初期化
13:
14:         bsr     chkarg          *コマンドラインの解析
15:
16:         bsr     do              *メイン処理
17:
18:         DOS     _EXIT           *正常終了
19:
20: *
21: *       メイン処理（今は何もしない）
22: *
```

```
23: do:
24:         rts
25:
26: *
27: *       コマンドラインの解析
28: *
29: chkarg:
30:         addq.l  #1,a2           *a2=コマンドライン文字列先頭
31:
32:         lea.l   arg1,a0         *a0=引数切り出し領域
33:
34:         moveq.l #2-1,d2         *以下を２回繰り返す
35:
36: ckarg0: bsr     nextarg         *スペースをスキップし
37:                                 *  スイッチがあれば処理する
38:         tst.b   (a2)            *引数があるか？
39:         beq     usage           *  ないなら引数が足らない
40:
41:         bsr     getarg          *引数１つをa0以降に取り出す
42:
43:         pea.l   nambuf          *DOSコールを使って
44:         move.l  a0,-(sp)        *  ファイル名を
45:         DOS     _NAMECK         *  展開してみる
46:         addq.l  #8,sp           *
47:         tst.l   d0              *d0が０でなければ
48:         bne     usage           *  ファイル名の指定に誤りがある
49:
50:         lea.l   256(a0),a0      *a0 = a0+256
51:
52:         dbra    d2,ckarg0       *d2.wが-1になるまで繰り返す
53:
54:         bsr     nextarg         *さらにスペースを飛ばす
55:         tst.b   (a2)            *引数があるか？
56:         bne     usage           *  あるなら引数が多い
57:
58:         rts
59:
60: *
61: *       スペースを飛ばしつぎの引数先頭までポインタを進める
62: *       スイッチがあれば処理してしまう
63: *
64: nextarg:
65:         bsr     skipsp          *スペースをスキップ
66:
67:         cmpi.b  #'/',(a2)       *引数の先頭が
68:         beq     nxarg0          *  /,-であれば
69:         cmpi.b  #'-',(a2)       *  スイッチ
70:         beq     nxarg0          *
71:
72:         rts                     *スイッチはもうない
73: *
74: nxarg0: addq.l  #1,a2           *'/'や'-'の分ポインタを進める
75:         move.b  (a2)+,d0        *１文字取り出す
76:         bsr     toupper         *大文字に変換しておく
77:         cmpi.b  #'A',d0         *Aスイッチ？
78:         beq     asw             *  そうなら分岐
79:         cmpi.b  #'B',d0         *Bスイッチ？
```

```
 80:          beq     bsw             *  そうなら分岐
 81:          bra     usage           *無効なスイッチが指定された
 82: *
 83: asw:     tst.b   Aflg            *Aスイッチの二重指定？
 84:          bne     usage           *  そうならエラー
 85:          move.b  #$ff,Aflg       *Aスイッチ0N
 86:          bra     nextarg         *つぎのスイッチがあるかもしれない
 87: *
 88: bsw:     tst.b   Bflg            *Aスイッチの場合と
 89:          bne     usage           *  やっていることは同じ
 90:          move.b  #$ff,Bflg       *
 91:          bra     nextarg         *
 92:
 93: *
 94: *        英小文字→英大文字変換
 95: *
 96: toupper:
 97:          cmpi.b  #'a',d0         *英小文字か？
 98:          bcs     toupr0          *
 99:          cmpi.b  #'z'+1,d0       *
100:          bcc     toupr0          *
101:          subi.b  #$20,d0         *小文字なら大文字に変換
102: toupr0: rts
103:
104: *
105: *        a2の指す位置から引数１つ分を
106: *        a0の指す領域へコピーする
107: *
108: getarg:
109:          move.l  a0,-(sp)        *｛レジスタ待避
110: gtarg0: tst.b    (a2)            *1)文字列の終端コードか
111:          beq     gtarg1          *
112:          cmpi.b  #SPACE,(a2)     *2)スペースか
113:          beq     gtarg1          *
114:          cmpi.b  #TAB,(a2)       *3)タブか
115:          beq     gtarg1          *
116:          cmpi.b  #'-',(a2)       *4)ハイフンか
117:          beq     gtarg1          *
118:          cmpi.b  #'/',(a2)       *5)スラッシュ
119:          beq     gtarg1          *
120:          move.b  (a2)+,(a0)+     *  が現れるまで転送を
121:          bra     gtarg0          *  繰り返す
122: gtarg1: clr.b    (a0)            *文字列終端コードを書き込む
123:          movea.l (sp)+,a0        *｝レジスタ復帰
124:          rts
125:
126: *
127: *        コマンドライン先頭のスペースをスキップする
128: *
129: skpsp0: addq.l  #1,a2            *ポインタを進め
130:                                  *繰り返す
131: skipsp:                          *サブルーチンはここから始まる
132:          cmpi.b  #SPACE,(a2)     *スペースか？
133:          beq     skpsp0          *  そうなら飛ばす
134:          cmpi.b  #TAB,(a2)       *TABか？
135:          beq     skpsp0          *  そうなら飛ばす
136:          rts
```

```
137:
138: *
139: *       使用法の表示＆終了
140: *
141: usage:
142:         move.w  #STDERR,-(sp)    *標準エラー出力へ
143:         pea.l   usgmes           *  ヘルプメッセージを
144:         DOS     _FPUTS           *  出力する
145:         addq.l  #6,sp            *スタック補正
146:
147:         move.w  #1,-(sp)         *終了コード１を持って
148:         DOS     _EXIT2           *  エラー終了
149:
150: *
151: *       データ
152: *
153:         .data
154:         .even
155: *
156: Aflg:   .dc.b   0        */Aスイッチon/offフラグ（=0...off,<>0...on)
157: Bflg:   .dc.b   0        */Bスイッチon/offフラグ（=0...off,<>0...on)
158: *
159: usgmes: .dc.b   '機　能：入力ファイルを××して'
160:         .dc.b                '出力ファイルに書き出します',CR,LF
161:         .dc.b   '使用法：ARG2 入力ファイル　出力ファイル',CR,LF
162:         .dc.b   '        /A      ○○を無視します',CR,LF
163:         .dc.b   '        /B      △△を□□と見なします'
164: crlfms: .dc.b   CR,LF,0
165:
166: *
167: *       ワークエリア
168: *
169:         .bss
170:         .even
171: *
172: arg1:   .ds.b   256              *引数切り出し用バッファ１
173: arg2:   .ds.b   256              *引数切り出し用バッファ２
174: nambuf: .ds.b   91               *ファイル名展開用バッファ
175: *
176:         .stack
177:         .even
178: *
179: mystack:
180:         .ds.l   256              *スタック領域
181: mysp:
182:         .end
```

よくある規模のプログラムといったところか。ちょっと複雑そうだが，大筋はリスト４と変わらず，２個の引数をdbraのループで処理しているくらいの差なので，ここではスイッチの扱いだけに注目してもらおう。とくによく見てもらいたいのは，スイッチのチェックをいつするかという点だ。

```
A>ARG2 /A FILE1 FILE2
A>ARG2 FILE1 FILE2 /A /B
```

のように，どこにスイッチを置いても正しく処理ができるように細工してある。

```
A>ARG2 /A FILE1 /B FILE2
```

というものまで可能だし，よく見てもらうと，

```
A>ARG2 /AFILE1/BFILE2
```

のようなべた書きすら通すようになっているのがわかるだろう(最後のパターンは副作用のようなものだが)。

スイッチのチェックを行うのはサブルーチンnextargだ。このサブルーチンは，先頭の空白を読み飛ばしたうえで，スイッチがあればその処理を行い，そうでなければそのまま戻るという動作をする。どちらにしろ，このサブルーチンから戻ったときにはa2レジスタは空白でもスイッチでもない文字(またはエンドコード00_H)を指すことになる。

空白を飛ばした後，まず先頭1文字が〝-〟か〝/〟かどうか調べる。そうでなければスイッチではないパラメータ(か00_H)だからそのまま戻る。〝-〟か〝/〟のどちらかであれば，スイッチのはずだからポインタを進め，次の1文字を取り出す。スイッチが大文字・小文字のどちらで指定されても困らないように，ここでサブルーチンtoupperを呼んで，この1文字を大文字に変換しておく。

それから順に/Aスイッチかな？ /Bスイッチかな？と調べ，どちらでもなければ使用法の表示に飛ぶ。どちらかのスイッチであれば，それぞれに対応した1バイトのワークにFF_Hを書き込む。このワークは，いわばスイッチがONかOFFかを表すフラグであり，0ならOFF，それ以外ならONを意味するものとする。後のメイン処理から必要に応じて

```
tst.b     Aflg
bne       /Aスイッチ がONの処理へ
OFFの処理～
```

というように参照されることになるだろう。

さて，実際にはフラグを立てる直前にすでにそのスイッチがONになっているかどうかを調べる処理が挟まっている。これは，

```
A>ARG2 /A /A FILE1 FILE2
```

のように，同じオプションが2度指定されるのをはじくためだ。故意に同じスイッチを複数回指定することは考えられないので，ユーザーの操作ミスと見なして早目に教えてあげるわけだ。また，このサンプルでは/Aスイッチと/Bスイッチを同時に指定してもかまわないものとしているが，2種類のスイッチの意味が矛盾するような場合は，同様のチェックを競合するスイッチに対しても行う必要があるだろう。

スイッチを1個処理したら，ふたたびnextargの先頭に飛ぶ。もちろん，これは複数のスイッチが連続して指定されても処理できるようにするためである。

今後の拡張の方針

ここまでのパターンが飲み込めれば，オプションやスイッチの数が変わっても，これらの応用で片付けることができるだろう。後は場合に応じた細かな配慮の問題になってくる。最後に1点だけ，かんたんな工夫の例を示そう。

拡張子の省略を許す(リスト6)

そのプログラムが，ある決まった拡張子のファイルを扱うのであれば省略することを認め，プログラム側で適当に補ってくれたほうがユーザーに優しい。ここで，本当に拡張子のないファイルを指定したい場合もあるので，

```
A>PROG FILE.
```

のように末尾に〝.〟をつけることで拡張子のないファイルを表し，

```
A>PROG FILE
```

のように〝.〟もない場合にかぎり，内部で拡張子を補うことにする。これはごくふつうに用いられている区別のしかただと思う。

拡張子があるかどうかを調べ，なければ補うという処理自体は単純な文字列操作になる。ファイル名の先頭から順に〝.〟を探し，見つからなければ文字列の後ろに用意しておいた拡張子を付け加えればよい。

拡張子の有無を調べるには，さきほども出てきたnameckを使うという手もある。nameckの動作テストプログラムNAMETEST.Xで次の2つのパターンを試してもらいたい。

```
A>NAMETEST TEST.
A>NAMETEST TEST
```

上のパターンのようにファイル名がピリオドで終わっている場合は，nameckで展開された後の拡張子は〝.〟1文字からなる文字列になっている。また，下の例のように拡張子がなく，ピリオドもない場合は展開後の拡張子は空文字列である。つまり，nameckにかけてから，ファイル名展開バッファの拡張子にあたる部分(バッファ先頭から86バイト目)を調べ，00_Hか〝.〟かで拡張子が省略されたかどうかを知ることができる。

これを利用した例をリスト6に挙げておく。このサブルーチンchkextは，リスト3のパラメータ解析処理の直後，メインルーチンの直前から呼び出すように作ってある。リスト3の113行あたりにでもこのサブルーチンを挿入し，また，リスト3の14行の空行を，

```
bsr     chkext
```

のように変更してアセンブルしてもらいたい。

リスト6

```
 1:          .text
 2:          .even
 3: *
 4: *        拡張子が省略されていたら
 5: *        適当な拡張子を補う
 6: *
 7: chkext:
 8:          tst.b    nambuf+86        *拡張子はあるか
 9:          bne      chkex0           *  あるなら何もしない
10:
11:          lea.l    arg,a0           *用意してある拡張子を
12:          lea.l    dext,a1          *  連結する
13:          bsr      strcat           *
14:
15: chkex0: rts
16:
17: *
18: *        文字列を連結する
19: *        a0=被連結文字列,a1=連結文字列
20: *
21: strcat:
22:          tst.b    (a0)+            *(a0)は0か?
23:          bne      strcat           *そうでなければ繰り返す
24:          subq.l   #1,a0            *行きすぎたから1つ戻る
25: strcpy:
26:          move.b   (a1)+,(a0)+      *1文字転送
27:          bne      strcpy           *終了コードまで繰り返す
28:          rts
29: *
30:          .data
31:          .even
32: *
33: dext:    .dc.b    '.$$$',0         *補う拡張子
```

サブルーチンchkextが呼ばれた時点では，すでにnameckの結果がnambuf以下に格納されているので，nambuf+86の1バイトをtst命令で調べ，00Hであったなら以下に切り出されているファイル名の末尾に〝.$$$〟という拡張子を付け加えている。文字列の連結処理には以前作ったサブルーチンを流用した。

ワイルドカードまで手が回らなかったのは残念だが，本章はこれくらいにしておこう。

さて，プログラムが使いにくくなるかならないかはほんのちょっとした部分の差でしかない。ならば，わずかな手間を惜しまないで使う側の立場に立ってプログラムをどんどん作ってばらまき，X68000のプログラム資産を増やして，みんなで幸せになろうじゃないか。

CHAPTER 10

サブルーチンに汎用性を

CHAPTER 8

サブルーチンに汎用性を

ASSEMBLER

この章では，比較的大規模なプログラムを作る際に知っておくと便利な(＝楽できる)技について話してみる。ユーザーライブラリの構築方法を中心に，その周辺技術までを紹介する。

ここでいうライブラリとは，汎用のルーチン群を指す。いつでもすぐに使えるようなプログラムのパーツ集だ。当然，いざ大きなプログラムを作ろうと思っても，その時点でライブラリを作り出すのでは手遅れである。日ごろから汎用性の高いルーチン群を集めておかなければ意味がない。ライブラリは作ろうと思って作るのではなく，本来はいつの間にかできている性格のものだ。１本のプログラムを作るたびに他のプログラムでも使えそうなパーツをピックアップしていくのがよい。もっとも，プログラムを書くときにある程度は意識していないと，なかなか使い回しのきくルーチンはできないというのもまた事実だ。

そういうわけだから，まずはその汎用性の高いルーチンの作り方あたりから始めてみる。

賢者(?)のサブルーチン

ある人がプログラムを作っていて，ふと「この処理は前に作ったプログラムにも出てきたぞ」と思ったとする。同じ処理を何度もプログラミングするのは非能率的なので，多少頭を巡らせて以前作ったルーチンを〝流用〟してやろうと考えた。ごく自然な発想だ。これを突きつめていけばライブラリに行き着くわけだが，彼はそこまで深くは考えない。ディスクの中からしばらく前に作ったプログラムのソースを引っ張り出してきて，作成中のプログラムといっしょにエディタに読み込み，カット&ペーストして必要な部分を抜き出せば，時間と労力の節約になるだろうぐらいに考えた。内心，自分はなんて冴えてるんだ，とか思う。

ところが，あいにくその〝必要な処理〟がメインルーチンの一部に埋め込まれていたとする。どこからどこまで抜き出せばよいのか探さなければならなくなった。作ってから１週間もたてば記憶もだいぶ薄れているだろうし，日ごろからコメントをマメにつけていなかったりすると手がかりがまったくない。肝心な部分が見つかるかどうかは疑わしい。結局，彼は諦めた。

ここで第１の教訓が得られる。汎用性云々以前の問題で，当たり前すぎて恥ずかしいのだが，こういうことだ。

教訓1：あるルーチンを他のプログラムでも使う可能性がある場合は，その部分を明確な処理単位であるサブルーチンにしておくのがよい。

もう1つ無理やり教訓をひねり出すと，

教訓2：コメントはないよりあるほうがマシ。

となる。

さて，しばらくしてその彼がまた同じような事態に陥った。彼も少しは学習したので，今度は必要な処理はちゃんとサブルーチンにしてある。どんな働きをするサブルーチンかもかんたんなコメントで示してあったので，目的のパーツはすぐに見つかった。彼は，そのサブルーチンを抜き出して作成中のプログラムに持ってくる。

これで終わりだと思ったら甘かった。そのサブルーチンでは内部でいくつかのレジスタを使用しているのだが，いま作っているプログラムのメインルーチンでも同じレジスタを使っていたのだ。気づいたからよかったものの，もし見落としていたらとんでもないことになるところだ。彼は，サブルーチンの先頭でmovemを使って内部で使用するレジスタをすべてスタックに待避し，rtsの直前で復帰するようにして事なきを得た。今回は，いちおううまくいったようだ。

ここでの教訓は，

教訓3：サブルーチンではレジスタを破壊しないようにするのが汎用性の第1歩。

ということにでもなるだろう。

一言付け加えておくと，あるプログラム中のサブルーチンが汎用であるためには，他のルーチンから独立し，切り離されていることが必要条件になる。他のルーチンについて〝知らなければ知らないほど〟よい。ここでは無知は美徳である。メインルーチンでどのレジスタを使っているかなんて知らないほうが楽ではある。サブルーチン内で使用するレジスタの値を待避するのはメインルーチンの顔を立てるためではない。サブルーチン側(とプログラマ)が楽をするためである。

では，架空の名もない彼にはここで退場していただいて，次にこの本で以前作ったサブルーチンの独立性・汎用性を試しに見てみよう。

サブルーチンの使用状況を考える

リスト1のprtdecは，5章で使った無符号数の10進表示サブルーチンだ。d0.wに表示したい数を入れて呼び出すと，その数を左詰めで標準出力に書き出す。サブルーチン内部で使用するレジスタの値はスタックに保存してあるし，いちおう使い回しがきくような作りになっている。このサブルーチンを作ったときには〝いちおう汎用性を狙ったので，かなりがっちり作ってある〟なんて偉そうなコメントがついていたりもする。

リスト1

```
 1: *        D0.Wを10進左詰めで表示するサブルーチン
 2:
 3:         .include        doscall.mac
 4: *
 5:         .text
 6:         .even
 7: *
 8: prtdec:
 9:         movem.l  d0/a0,-(sp)      * {d0,a0をスタックに待避
10:
11:         andi.l   #$0000ffff,d0    *上位ワードをクリア
12:         lea.l    bufend,a0        *ポインタ初期化
13: prtdec0:
14:         divu.w   #10,d0           *d0.lを10で割る
15:         swap.w   d0               *上位ワードと下位ワードを交換
16:         addi.w   #'0',d0          *0～9 → '0'～'9'
17:         move.b   d0,-(a0)         *1桁格納
18:         clr.w    d0               *次の除算に備える
19:         swap.w   d0               *上位ワードと下位ワードを交換
20:         bne      prtdec0
21:
22:         move.l   a0,-(sp)         *変換した文字列を表示する
23:         DOS      _PRINT           *
24:         addq.l   #4,sp            *
25:
26:         movem.l  (sp)+,d0/a0      *} d0,a0を復帰
27:         rts
28: *
29:         .data
30:         .even
31: *
32: buff:   .ds.b    5                *10進文字列格納領域
33: bufend: .dc.b    0                *文字列の終了コード
34:
35:         .end
```

たしかに，このサブルーチンが1つあれば，いつでも必要なときにd0.wの値を表示することができる。あたかも〝d0.wの値を表示する〟という命令があるかのように，だ。その目的においては便利このうえない。しかし，これをもって汎用性を謳うのは嘘もいいところだ。このサブルーチンがそのまま使える場面はそう多くはない。

たとえば，表示したい数がd0.wではなくd1.wに入っていたとする。あらたにd1.wの内容を表示するサブルーチンを作るのはいくらなんでも間抜けだから，次のようにd0.wにd1.wの値をいったん代入してから，このprtdecサブルーチンを呼び出すことになる。

```
move.w      d1,d0
bsr         prtdec
```

d0を介して引数をサブルーチンに引き渡すわけだ。

ところが，もしかするとd0.wは別の目的にすでに使われているかもしれない。その場合は，d0.wの値を一時的に待避しておく必要があるので，

```
move.w      d0, −(sp)
move.w      d1, d0
bsr         prtdec
move.w      (sp)+, d0
```

のようにしなければならなくなる。

これはサブルーチンの呼び出し手順としては煩雑すぎるし，明らかにサブルーチンの独立性を損っている。d0に値を入れてからサブルーチンを呼び出すという方法自体の問題だ。

また，値を標準出力ではなく，標準エラー出力に書き出したいという場合や，値を左詰めではなく，5桁の右詰めで表示したいという場合，あるいは値の符号を考慮して表示したい場合を考えてみよう。これらの要求に対しては，prtdecサブルーチンはまったく役に立たず，〝prtdecとほとんど同じだが，ごく一部が違うサブルーチン〟をそれぞれ別々に用意しなければならない。

これは，1つのサブルーチンとして切り出す処理単位の選び方がまずかったためだ。prtdecサブルーチンは，処理単位としては大きすぎたのである。汎用性を目指すのであれば，数値を文字列に変換する処理と，その文字列を表示する処理を切り離すべきだった。10進表示を行うサブルーチンだけがあるのと，数値→文字列変換ルーチンだけがあるのでは後者のほうがずっと幅広く使えるだろう。数値→文字列変換ルーチンを利用すれば10進表示ルーチンを作ることはできるが，逆はできないのだ[1]。

■ 1) ネジ回しと並んで，大が小を兼ねないということを示す典型例である。

そこで，

教訓4：汎用性を目指すなら，1個のサブルーチンにはただ1つの処理だけを行わせるようにしよう。

となる。

引数の渡し方

ここで，さきほど問題になったサブルーチンへ引数を渡す方法について，少し深く掘り下げてみよう。独立性という点だけではなく，速度効率・メモリ効率も比べてみたい。

リスト2にいくつかの例を示す。どの例もサブルーチンsubにロングワードデータ1つとワードデータ1つを渡すものとしよう。便宜上，ロングワードのほうを第1引数，ワードのほうを第2引数と呼ぶ。なお，サブルーチン中で使うレジスタの値を保存することは考えない。

レジスタを使う

リスト2-a)

```
 1: main:
 2:          move.l   #10000,d0          *第1引数セット
 3:          move.w   #$abcd,d1          *第2引数セット
 4:          bsr      sub                *サブルーチンコール
 5:
 6:           :
 7:
 8: sub:
 9:          *引数はもうd0.l,d1.wに入っている
10:
11:           :
12:
13:          rts
```

a)は，さきほどのprtdecのようにレジスタを介して値を渡す方法だ。もっとも直感的な方法といえる。レジスタに値をポンと入れてサブルーチンを呼び出すだけなので，処理ステップも短く，高速である。MPU68000にはレジスタがたくさんあるから，3つ4つの引数を渡すぐらいならわけはない。

しかし，さきに述べたように，引数の受け渡しに使用するレジスタとメインルーチンで使用するレジスタの競合をつねに気にしなければならず，サブルーチンの独立性は損われる。ある特定のプログラム中で頻繁に使われ(それゆえ呼び出しは高速に行われることが望ましい)，かつ，他のプログラムに流用することを考えない(独立性・汎用性は捨ててかかる)といった限られた場合にのみ採用するとよいだろう。

特定のワークエリアを使う

リスト2-b)

```
 1: main:
 2:          move.l   #10000,par1        *第1引数セット
 3:          move.w   #$abcd,par2        *第2引数セット
 4:          bsr      sub                *サブルーチンコール
 5:
 6:           :
 7:
 8: sub:
 9:          move.l   par1,d0            *第1引数取り出し
10:          move.w   par2,d1            *第2引数取り出し
11:
12:           :
13:
14:          rts
15:
```

```
16: par1:   .ds.l   1               *第１引数受け渡し領域
17: par2:   .ds.w   1               *第２引数受け渡し領域
```

b)は，レジスタのかわりに，ある特定のワークエリアに値を入れてからサブルーチンを呼び出すという方法だ。レジスタを使ったときのような〝ぶつかりあい〟を気にしなくてもすむし，まったく同じ引数で再度サブルーチンを呼び出す場合は，前回でワークにセットした値をそのまま利用するという手抜き技が使える。ただ，引数を受け渡すという目的のためだけにワークエリアを用意するのは，メモリの無駄遣いである。しかも，兼用しようなんて考えると今度はぶつかりあいを気にする必要が出てきてしまう。

ポインタで渡す

リスト2-c)

```
 1: main:
 2:         move.l   #10000,par1     *第１引数セット
 3:         move.w   #$abcd,par2     *第２引数セット
 4:         lea.l    pars,a0         *引数受け渡し領域
 5:         bsr      sub             *サブルーチンコール
 6:
 7:          :
 8:
 9: sub:
10:         move.l   (a0)+,d0        *第１引数取り出し
11:         move.w   (a0),d1         *第２引数取り出し
12:
13:          :
14:
15:         rts
16:
17: par1:   .ds.l   1               *第１引数受け渡し領域
18: par2:   .ds.w   1               *第２引数受け渡し領域
19:                                 *本当なこんな固定領域は不用
```

c)はb)と似たような方法で，適当なメモリ領域に引数をセットし，その領域へのポインタをサブルーチンに渡すというものだ。パラメータが複数ある場合は，連続したメモリに決まった順序でセットするようにすれば，やはりポインタは１つですむ。

パラメータの受け渡し用にメモリ領域が必要ではあるが，パラメータの所在はポインタで示されるので，空いているメモリならどこでも使える。アドレスレジスタをパラメータ受け渡し専用に１個つぶす覚悟があれば，効率も悪くはないだろう。しかし，最終的にアドレスレジスタでポインタを渡すかぎり，独立性という意味ではa)と大差がない。

プログラムの一部を使う

リスト2-d)

```
 1: main:
 2:          move.l   #10000,par1      *第1引数セット
 3:          move.w   #$abcd,par2      *第2引数セット
 4:          bsr      sub              *サブルーチンコール
 5: par1:    .ds.l    1                *第1引数受け渡し領域
 6: par2:    .ds.w    1                *第2引数受け渡し領域
 7: retn:    ～
 8:
 9:           :
10:
11: sub:
12:          movea.l  (sp)+,a0         *引数へのポインタ
13:          move.l   (a0)+,d0         *第1引数取り出し
14:          move.w   (a0)+,d1         *第2引数取り出し
15:          move.l   a0,-(sp)         *リターンアドレスセット
16:
17:           :
18:
19:          rts
```

d)は，引数をプログラムの途中に埋め込むという方法だ。サブルーチンを呼び出すbsrの直後に必要なだけのメモリを用意し，ここに引数をセットしておく。一見しただけでは，この引数の部分も命令と見なして実行してしまいそうに見えるが，そこはそれ，サブルーチン側でつじつまをあわせ，ちゃんとスキップするようにできている。

サブルーチンsubを呼び出した時点で，スタックトップにはサブルーチンからのリターンアドレスが積まれている。リスト2-d)の場合は，ラベルpar1で示されるアドレスがスタックトップにあるわけだ。これを適当なアドレスレジスタに取り出すと，そのアドレスレジスタはちょうどc)のパターン同様引数群を指すポインタとなる。このポインタをポストインクリメントし，順に引数を取り出していくと，全部の引数を取り出した時点で，このポインタは引数群の直後(リスト2-d)ではラベルretnで示されるアドレス)を指す。このアドレスは，〝サブルーチンから戻って最初に実行すべき命令〟のアドレスになっており，スタックに積み直したうえでrtsすれば，望みの位置から処理を再開できることになる。

この方法はパズルとしてはおもしろいが，プログラムサイズが引数受け渡し領域の分だけ間延びする，引数を取り出してリターンアドレスを再設定するまでの間，スタックがあまり自由に使えない(やってできないことはないのだが)などの欠点もあり，68000ではあまり使われることはない。

プログラム自体を書き換える

リスト2-e)

```
 1: main:
 2:          move.l   #10000,par1+2    *第1引数セット
 3:          move.w   #$abcd,par2+2    *第2引数セット
 4:          bsr      sub              *サブルーチンコール
 5:
 6:           :
 7:
 8: sub:
 9: par1:    move.l   #$aaaaaaaa,d0    *第1引数取り出し
10: par2:    move.w   #$bbbb,d1        *第2引数取り出し
11:
12:           :
13:
14:          rts
```

e)はもっと強引な方法だ。引数受け渡し領域はプログラム中に埋め込まれるが，よけいなメモリ領域を必要としない。どうやるかといえば，直接マシンコードのオペランド部分を書き換えるのだ。いわゆる〝自己書き換え〟に類するテクニックで，使用するにあたってはアセンブリ言語レベルだけではなく，マシンコードレベルの知識が必要だ。つまり，アセンブリ言語で書いたプログラムがどのようなコードに変換されるかを知っていなければならない。

リスト2-e)では，2行と9行，3行と10行が対になっている。2行，3行が引数を渡す部分で，9行，10行がその引数を受け取る部分だ。2行ではpar1+2で示されるアドレスにロングワードで引数をセットしている。+2がどこから出てきたのかは，9行の，

```
move.l  #$aaaaaaaa,d0
```

が，アセンブルすることによって，

```
2030 AAAA AAAA
```

という3ワードのコードに変換されることと関係がある。最初の1ワードは，

```
move.l  #～,d0
```

を意味し，残りの2ワードでロングワードの即値$AAAAAAAA_H$を表している。2行でこの後半2ワードを書き換えることで，任意の値をd0.lに代入する命令に組み換えることができるわけだ。

この方法の利点としては，場合によっては速度・メモリの効率がよい場合があるとか，引数のセットと実際のサブルーチン呼び出しが必ずしも連続していなくてもよいとかということがある。しかし，マシンコードレベルの知識が必要という最大の欠点のために，低機能なマイクロプロセッサならともかく，68000ではこんなことをするのはバカげている。68000にはもっとエレガントな方法が似つかわしい。

スタックに積む

リスト2-f)

```
 1: main:
 2:          move.w   #$abcd,-(sp)      *第２引数セット
 3:          move.l   #10000,-(sp)      *第１引数セット
 4:          bsr      sub               *サブルーチンコール
 5:          addq.l   #6,sp             *sp補正
 6:
 7:           :
 8:
 9: sub:
10:          move.l   4(sp),d0          *第１引数取り出し
11:          move.w   8(sp),d1          *第２引数取り出し
12:
13:           :
14:
15:          rts
```

f)が，その68000らしいエレガントな方法というやつだ。引数はスタックに積んでサブルーチンに渡す。DOSコールの呼び出し手順でおなじみの方法だろう。エレガントというよりは重厚さを感じさせる。はっきりいってレジスタで直接渡すような方法と比べると，速度的にはいささか不利ではある。半面，レジスタやワークエリアをいっさい使用しないので，汎用性を狙うにはベストの選択となる[2)]。

■ 2) スタックを使用するのだから，正確にはspを間接的に利用することになる。しかし，少なくとも表には出ないし，意識する必要もあまりない。

細かく見てみよう。サブルーチンsubが呼び出された時点でのスタックの状態は，図１のようになっている。スタックトップには，例によってサブルーチンからのリターンアドレスが積まれている。その次に，積んだ順序の逆順で引数が並ぶ。この点を考慮して，リスト2-fではサブルーチンに引数を渡す順序が他の例とは見かけ上反転している。

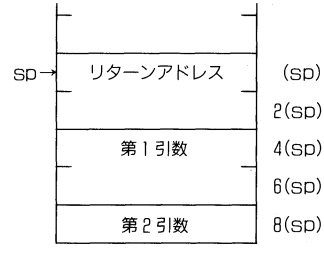


図1
スタックを使った
パラメータの受け渡し

一般には，スタックからデータを取り出すには，そのデータがスタックトップになければならない。第1引数を取り出すためにはリターンアドレスが邪魔だ。まともにやると，一度スタックトップに積まれたリターンアドレスを適当なレジスタなりメモリなりに取り出してから，引数を順次ポップすることになるだろう。

しかし，68000のspはそのへんによくあるマイクロプロセッサのスタックポインタとはわけが違う。68000のspはa7レジスタの別名であり，アドレスレジスタのうちの1本だ。当然，アドレスレジスタに関するアドレッシングモードもすべて使える。ここで，〝ディスプレースメント付きアドレスレジスタ間接アドレッシング〟を思い出してもらえれば話は早い。このアドレッシングモードは〝アドレスレジスタに変位(ディスプレースメント)を加えたアドレス〟を指定するものだった。アドレスレジスタが指しているアドレスそのものではなく，その前後に少しずれた位置を指定するのに利用する[3]。

■ 3) 補足しておくと，〝ディスプレースメント付き～〟で使用可能な変位は，符号付き16ビット数で表せる範囲(-32768～+32767)だ。変位をアドレスレジスタに加算して実効アドレスを求める際には，変位は32ビット長に符号拡張される。

そこで，もう一度図1を見てもらおう。第1引数は現在のspに4を，第2引数は8を足したアドレスに置かれている。それぞれのアドレスは，〝ディスプレースメント付き～〟を使えば〝4(sp)〟，〝8(sp)〟のように表現される。後は10行，11行のようにmove命令一発でレジスタに引数を取り出すことができる。

なお，この場合，サブルーチンから戻った時点でスタックに引数が積まれたままになっていることに注意したい。DOSコール呼び出しのときと同様，スタックポインタの補正が必要になる。

スタックにポインタを積む

リスト2-g)

```
 1: main:
 2:          move.l   #10000,par1      *第1引数セット
 3:          move.w   #$abcd,par2      *第2引数セット
 4:          pea.l    par1             *引数受け渡し領域
 5:          bsr      sub              *サブルーチンコール
 6:          addq.l   #4,sp            *sp補正
 7:
 8:           :
 9:
10: sub:
11:          movea.l  4(sp),a0         *引数へのポインタ
12:          move.l   (a0)+,d0         *第1引数取り出し
13:          move.w   (a0),d1          *第2引数取り出し
14:
15:           :
16:
```

```
17:         rts
18:
19: par1:   .ds.l   1              *第１引数受け渡し領域
20: par2:   .ds.w   1              *第２引数受け渡し領域
```

おまけとして，g)はc)とf)をあわせたようなものだ。引数は適当なメモリ領域にセットし，そのポインタをスタックに積むことでサブルーチンへと渡す。サブルーチンでの引数受け取り手順は，ポインタを取り出すところまでがf)と同じで，さらに実際の引数を取り出すにはc)と同様の手順を踏む。この方法は，大量の引数をスタックを介してサブルーチンに引き渡す際のヒントになっている。

以上，サブルーチンへ引数を渡す方法を何通りか比較検討してみた。結論としては，速度重視ならa)のレジスタを使う方法，サブルーチンの汎用性重視ならf)のスタックを使う方法ということになると思う。そのバリエーションとしてc)，g)のポインタを使う方法も併用されるだろう。

この章ではサブルーチンの汎用性を追求する主旨だから，いちおうスタックに引数を積むパターンを採用し，以下，さらに詳しく見ていくことにする。

スタックに積む場合の詳細

残念なことにリスト2-f)のままでは，プログラミング上の問題が残っている。試しにリスト2-f)にサブルーチン内で使用するレジスタの待避・復帰処理を付け加えてみてほしい。かりにd0～d7のデータレジスタすべてを保存するものとしよう。

リスト3-aのようにmovemを２ヵ所に挟み込めばいいように思えるところだが，そうではない。正しくはリスト3-bのように引数取り出し部分にも手を加える必要がある。待避したレジスタの分だけスタックポインタがずれてしまうからだ。

リスト3-a

```
 1: sub:
 2:         movem.l  d0-d7,-(sp)      *レジスタ待避
 3:
 4:         move.l   4(sp),d0         *第１引数取り出し？
 5:         move.w   8(sp),d1         *第２引数取り出し？
 6:
 7:         :
 8:
 9:         movem.l  (sp)+,d0-d7      *レジスタ復帰
10:         rts
```

リスト3-b

```
 1: sub:
 2:          movem.l  d0-d7,-(sp)       *レジスタ待避
 3:
 4:          move.l   4+4*8(sp),d0      *第1引数取り出し
 5:          move.w   8+4*8(sp),d1      *第2引数取り出し
 6:
 7:           :
 8:
 9:          movem.l  (sp)+,d0-d7       *レジスタ復帰
10:          rts
```

このことは，サブルーチンにわずかに手を入れることさえ難しくする。たとえば，サブルーチンに処理を付け加えた結果，あらたにa0レジスタも保存しなければならなくなったとしよう。movemのレジスタリストにa0を付け加えるときに，引数取り出し部分も変更する必要があるのを忘れる可能性は高い。こういうミスを後から見つけることはかなり難しいだろう。

また，リスト3-cのようにスタックに値をいくつか積んでからあらためて引数を取り出すような形になっていたりすると，さらに複雑さが増す。ある瞬間にスタックにどれだけのデータが積まれているかをいちいち計算しないとディスプレースメントが求められないのだ。

リスト3-c

```
 1: sub:
 2:          movem.l  d0-d7,-(sp)       *レジスタ待避
 3:
 4:          move.l   4+4*8(sp),d0      *第1引数取り出し
 5:          move.w   8+4*8(sp),d1      *第2引数取り出し
 6:
 7:           :
 8:
 9:          move.l   d2,-(sp)          *レジスター時待避
10:          move.l   4+4*8+4(sp),d2    *第1引数再取り出し
11:
12:           :
13:
14:          movem.l  (sp)+,d0-d7       *レジスタ復帰
15:          rts
```

どうしてこんなことになったかといえば，基準として選んだspが頻繁に変更されてしまうからだ。もちろん，スタックを極力使わないようにすればどうにかなるが，スタックを使わずにマシン語プログラムを書くなんて拷問に等しい。

そこで，スタックに変わる別の基準を作ることを考える。たとえば，リスト4-aのようにサブルーチンが呼び出された直後のspを適当なアドレスレジスタに入れてしまうというのはどうだろう。リスト4-aではa6を使ってみた。引数を取り出す部分はspがa6に変わっただけだし，以下，spがどんなに変動してもa6さえ固定しておけば，引数はいつでも〝同じ場所〟にあるから安心だ。

リスト4-a

```
 1: sub:
 2:          movea.l  sp,a6              *スタックフレーム形成
 3:          movem.l  d0-d7,-(sp)        *レジスタ待避
 4:
 5:          move.l   4(a6),d0           *第1引数取り出し
 6:          move.w   8(a6),d1           *第2引数取り出し
 7:
 8:           :
 9:
10:          movem.l  (sp)+,d0-d7        *レジスタ復帰
11:          rts
```

これはスタック上に枠組みを作ってやるようなもので，その意味で確保された領域のことを「スタックフレーム」と呼ぶ。また，スタックフレームのベースアドレス(基準となるアドレス)を保持するレジスタは「フレームポインタ」と呼ばれたりもする。

この方法を採ることでサブルーチンは作りやすく，修正しやすくなる。また，ディスプレースメントをラベル定義しておけば，プログラムの読みやすさも上がるというオマケもついてくる。弊害でアドレスレジスタが1本使えなくなるが，太っ腹の68000にとっては制限にすら感じられない。

実際にはリスト4-aのままではa6が破壊されてしまうから，少し細工してリスト4-bのようにしたほうがよい。あらかじめa6をスタックに待避してからspの値を代入するわけだ。この場合，a6をスタックに積んだ分〝a6に代入した時点でのsp〟は4バイトずれていることになる。それにともなってスタック上の引数の位置も変わってくるが，サブルーチンの中では一定だから問題ではない。

リスト4-b

```
 1: sub:
 2:          move.l   a6,-(sp)
 3:          movea.l  sp,a6              *スタックフレーム形成
 4:          movem.l  d0-d7,-(sp)        *レジスタ待避
 5:
 6:          move.l   8(a6),d0           *第1引数取り出し
 7:          move.w   12(a6),d1          *第2引数取り出し
 8:
 9:           :
10:
11:          movem.l  (sp)+,d0-d7        *レジスタ復帰
12:          movea.l    (sp)+,a6
13:          rts
```

サブルーチンの最初と最後に余分な命令がいくつも並ぶのは気持ちのよいものではない。わずかとはいえプログラムが長くなれば，それだけ間違いの入り込む余地も大きくなる。だからというわけではないのだろうが，68000にはスタックフレームの形成・解放を行う専用命令がある。もともとはC言語などの高級言語をサポートしやすいように用意されたもののようだ。スタ

ックフレームの生成はlink，解放はunlk(unlinkを詰めてある)という命令で行う。これを利用すると，リスト4-bはリスト4-cのようにすっきりしてくる。

リスト4-c

```
 1: sub:
 2:          link    a6,#0           *スタックフレーム形成
 3:          movem.l d0-d7,-(sp)     *レジスタ待避
 4:
 5:          move.l  8(a6),d0        *第1引数取り出し
 6:          move.w  12(a6),d1       *第2引数取り出し
 7:
 8:           :
 9:
10:          movem.l (sp)+,d0-d7     *レジスタ復帰
11:          unlk    a6
12:          rts
```

linkは，次のようにして使う。

link　アドレスレジスタ，#ローカルエリアサイズ

第2オペランドに即値がくるという，68000の命令としては例外的な語順になっていることに注意してほしい。

この命令は，ローカルエリアサイズに0を指定すると，リスト4-bの2～3行と同じような働きをする。具体的には，次のような動作だ。

1) 指定されたアドレスレジスタをスタックに待避する。

2) 1)の後，spの値を指定のアドレスレジスタに代入する(すでにアドレスレジスタをスタックに積んだ分だけspは更新されている)。

本当はこの後，

3) spに第2オペランドのローカルエリアサイズを加える。

という動作があるのだが，ローカルエリアサイズが0であれば同じことだ。なお，ローカルエリアに関してはコラムを参照のこと。

対するunlkは，

unlk　アドレスレジスタ

のようにして使う。アドレスレジスタは，通常linkで指定したのと同じものを指定する。たんにlinkで生成したスタックフレームを解放する命令だと思っていればまちがいない。いちおう，具体的な動作を示すと，

1) 指定されたアドレスレジスタの値をspに代入する。

2) スタックトップのロングワードデータを指定のアドレスレジスタにポップする。

という動きだ。

link，unlkで指定するアドレスレジスタはa7でさえなければなんでもよいのだが，慣例としてa6が使われる。以後はこれにならうことにする。

COLUMN

ローカルエリア

ローカル(local)というのは，X-BASICやCなどのローカル変数のローカルと同じ意味で，〝局所的〟と訳すことになっている。これと対になるのがグローバル(global：大域的・広域的)だ。どちらも(主として変数名やラベル名などの識別名の)通用範囲(＝スコープ)を表す用語だ。字面を見ればわかるように，局所的といえばごく狭い範囲でのみ通用するという意味で，大域的といえばもっとずっと広い範囲で通用することを表す。〝通用する〟でわからなければ，〝見える〟とか〝使える〟とか〝意味を持つ〟とか〝(論理的に)存在する〟とか適当に読み替えてみるとよい。

ついでに挙げておくと，同じような場面で使われる(が，意味は違う)言葉に〝静的〟，〝動的〟というのがある。変数のように〝実体(この場合，一定量のメモリ)をともなうもの〟が，ずーっと存在していれば「静的」，現れたり消えたりすれば「動的」という。

Cだと多少複雑になるからX-BASICでいうと，メインルーチンで宣言した変数は，プログラムの(ほとんど)どこからでも使えるから大域的であり，いつでも存在するから静的である。また，関数の中で宣言した変数と，関数の仮引数として宣言した変数は，一部の範囲(宣言した関数内)だけで使えるから局所的であり，必要に応じて作られ，用がすんだら消される運命にあるから動的だ。

大域的な変数はつねに静的であり，動的な変数はふつう局所的である。ちなみに，静的でも局所的な変数はありうる。物理的にはずっと存在していても一部からしか見えなければ，やはり局所的と呼ぶわけだ。

さて，linkのところで出てきたローカルエリアというのは，サブルーチンの中で一時的に使用するために確保するメモリ領域をいう。linkを実行した時点で確保され，unlkで解放されるわけだから，動的でかつ局所的である。

ローカルエリアを用いる第1の利点は，一時的な作業用にわざわざ固定のメモリ領域を用意しなくてもすむということだ。linkではスタック上にローカルエリアを確保するわけだから，結局は同じメモリを複数のサブルーチンで首尾よく共有することができる。同じメモリ領域を使うとはいっても，確保した範囲はその瞬間には他のルーチンで使っていないことが保証されているから(スタックの未使用部分をこっそり使うような悪いプログラムでないかぎり)ぶつかりあいを心配する必要もない。

また，第2の利点としては，他のサブルーチンに〝よけいなことを教えないですむ〟ということが挙げられる。一時作業用メモリのような〝自分だけが知っていればよいこと〟を他のサブルーチンにわざわざ教えてあげたり，〝見える〟ところにおいておく必要はないのだ。他のルーチンを〝無知〟にすることで自分の独立性を保ち，相手の独立を促すのである。

さて，図Aにスタック上にローカルエリアを確保する様子を示す。基本的にはスタックポインタを強引に移動して空間を作ってやるというだけのことだ。

図のa)の状態から，

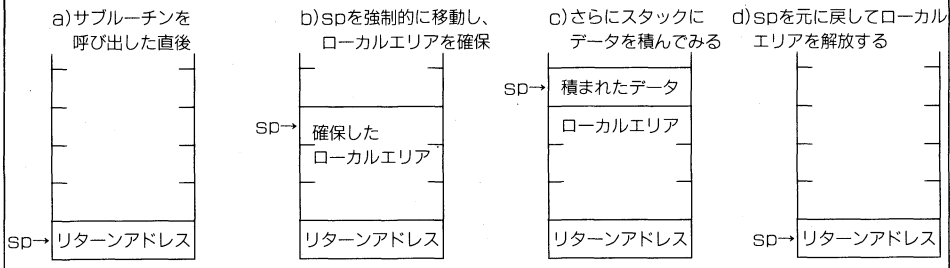
```
subq.l    #8, sp
```

とか，

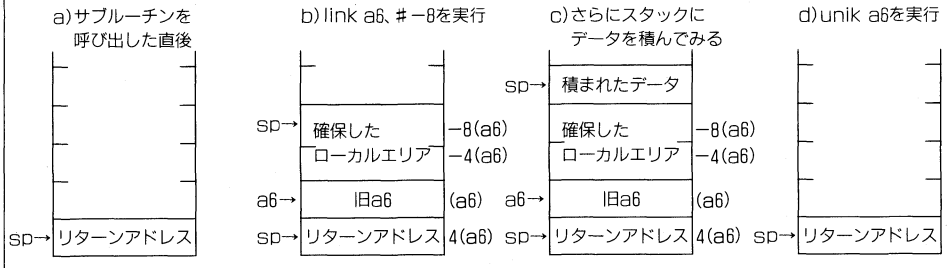
```
lea.l     −256(sp), sp
```

のようにスタックポインタを直接操作し減じれば，それぞれ8，256バイトのローカルエリア

図A　ローカルエリアの確保



図B　link, unlkによるローカルエリアの確保・解放



が確保される。b)を見ればわかるように，確保したローカルエリアはsp自身によってポイントされることになる。また，c)はローカルエリア確保後，スタックにデータを積んでもローカルエリアは破壊されたりしないことを，最後のd)はローカルエリアを解放するには，spを〝ローカルエリアを確保する前の値〟に戻してやればよいことを示している。

このように，link命令を使わなくてもローカルエリアは確保できるわけだが，linkの旨味は確保したローカルエリアをスタックフレームに取り込めるという点にある。図Bにlink命令を使ってスタック上にローカルエリアを確保した様子を示す。

```
link        a6,#−8
```

によって8バイトの領域を確保している。

a)の状態でlinkを実行すると，b)のようになる。直接spを操作したときと同様，この時点では確保したローカルエリアはspによってポイントされている。フレームポインタであるa6を基準にすれば，〝-8(a6)〟で示されるアドレス以降がローカルエリアである。

c)は，b)の状態からロングワードデータ1つをスタックに積んだ状態だ。spはすでにローカルエリアを指していないが，フレームポインタから見たローカルエリアは，同じ位置〝-8(a6)〟を保っている。

最後に，linkを使ってローカルエリアを確保する際の注意点を挙げておく。

上記の使用例を見ると気づくように，linkの第2オペランドは〝-(マイナス)確保するローカルエリアのバイト数〟になる。68000のスタックはアドレスの低いほうに成長していくから，ローカルエリアを確保するにはspの値を減じてやらなければならないわけだ。よって，linkの第2オペランドは必ず0か負の数になる。

また，第2オペランドは16ビットの符号付き数として扱われるので，指定可能な範囲は-32768～＋32767までになる。が，正の数は使う意味がないので実際には-32768～0の範囲

のみが使われる。このことはlinkで確保できるローカルエリアの大きさが最大32Kバイトまでであることを意味している。

さらに，すっかり話すのを忘れていた68000の謎の制約により，第2オペランドはつねに偶数でなければならない。奇数にすると，アドレスエラー攻撃を受けることになる。これに関しては，さらにコラム『.even疑似命令の意味』を参照してもらいたい。

COLUMN

.even疑似命令の意味

かんたんにいうと,.evenはアセンブル時に次の命令なり，データなりが偶数アドレスに置かれるように調整する働きをする。すでに次の命令が偶数アドレスに置かれる状態であれば何もせず，奇数アドレスに置かれそうになっていたら，1バイトの詰めものを入れる。固い言い方をすれば,〝ロケーションカウンタの値を偶数境界に整合させる〟命令である(ロケーションカウンタというのは，アセンブラが次に命令やデータをどこにおけばよいのかを数えるのに使うもので,AS.Xの場合はセクションごとに別々のロケーションカウンタが用意されている)。

なぜ,こんな疑似命令があるのかというと,68000ではメモリに対してワード単位,ロングワード単位でアクセスする場合には，そのアドレスが偶数でなければならないという制約があるからだ。多少いいかげんだが，68000が16ビットプロセッサだからだと思っていいだろう。16ビット幅のバスには奇数アドレスは〝きりが悪い〟のである。無理に実行しようとすると，〝アドレスエラー〟が発生する。これは，68000がプログラマに与えるほとんど唯一の〝頭の痛い〟制約である。これが同じ16ビットプロセッサの8086になると，もう少し制限は緩やかで，偶数境界をまたぐワードアクセスは内部で2回に分けて行うことになっている。つまり，実行速度が低下するだけですむわけだ。

メモリアクセスがすべて対象なのだから，プログラムカウンタの位置から命令コードを読み込むときもまた例外ではない。68000の命令コードはワード単位になっているからふつうは問題にならないが，プログラムの途中に奇数バイトのデータを挟んだりすると，やはりアドレスエラー発生の原因になる。プログラムカウンタはいつも偶数でなければならないのだ。

ものは考えようで，万が一pcがあらぬところを指しても，1/2の確率でアドレスエラーに引っかかって止まってくれるという見方もできる。これにメモリモードの保護機能が加わり，暴走しない，暴走してもダメージが少ないという安全なプログラミング環境が得られるというわけだ。

prtdecルーチンを改良する

ここでふたたびprtdecサブルーチンに立ち戻る。ここまでの話ではっきりした問題点を整理し，より汎用性のあるルーチンに改造してみた。結果はリスト5のようになった。深い意味はないが，サブルーチン名はputdecに変更されている。

リスト5

```
 1:          .include        const.h
 2:          .include        doscall.mac
 3: *
 4:          .text
 5:          .even
 6: *
 7: *        16ビット無符号整数を10進左詰めで表示する
 8: *
 9: *                  +----------------+
10: *        (a6)      |      旧a6      |      <-sp
11: *                  +................+
12: *                  |                |
13: *                  +----------------+
14: *        4(a6)     |リターンアドレス|
15: *                  +................+
16: *                  |                |
17: *                  +----------------+
18: *        8(a6)     |   表示する値   |
19: *                  +----------------+
20: *
21: value   =         8
22: *
23: putdec:
24:          link     a6,#0             *スタックフレーム形成
25:
26:          move.w   #STDOUT,-(sp)
27:          move.w   value(a6),-(sp)
28:          bsr      fputdec
29:          addq.l   #4,sp
30:
31:          unlk     a6
32:          rts
33:
34: *
35: *        16ビット無符号整数を10進左詰めでファイルに出力する
36: *        d0.lにDOSコールfputsの終了コードを持って戻る
37: *
38: *                  +----------------+
39: *        -6(a6)    |                |      <-sp
40: *                  +................+
41: *                  |                |
42: *                  +................+
43: *                  |                |
44: *                  +----------------+
45: *        (a6)      |      旧a6      |
46: *                  +................+
47: *                  |                |
48: *                  +----------------+
49: *        4(a6)     |リターンアドレス|
50: *                  +................+
51: *                  |                |
52: *                  +----------------+
53: *        8(a6)     |   表示する値   |
54: *                  +----------------+
55: *        10(a6)    |ファイルハンドル|
56: *                  +----------------+
```

```
 57: *
 58: buffsiz =       6
 59: temp    =       -buffsiz
 60: value   =       8
 61: fno     =       10
 62: *
 63: fputdec:
 64:         link    a6,#-buffsiz    *スタックフレーム形成
 65:                                 *＋ローカルエリア確保
 66:
 67:         pea.l   temp(a6)        *数値→文字列変換
 68:         move.w  value(a6),-(sp) *
 69:         bsr     utoa            *
 70:         addq.l  #6,sp           *
 71:
 72:         move.w  fno(a6),-(sp)   *変換後の文字列を出力
 73:         pea.l   temp(a6)        *
 74:         DOS     _FPUTS          *
 75:         addq.l  #6,sp           *
 76:
 77:         unlk    a6              *スタックフレーム解放
 78:         rts
 79:
 80: *
 81: *       16ビット無符号整数を文字列に変換する
 82: *
 83: *               +---------------+
 84: *       (a6)    |     旧a6      |     <-sp
 85: *               +...............+
 86: *               |               |
 87: *               +---------------+
 88: *       4(a6)   |リターンアドレス|
 89: *               +...............+
 90: *               |               |
 91: *               +---------------+
 92: *       8(a6)   |      値       |
 93: *               +---------------+
 94: *       10(a6)  | 文字列格納領域 |
 95: *               +...............+
 96: *               |   へのポインタ |
 97: *               +---------------+
 98: *
 99: value   =       8
100: buff    =       10
101: *
102: utoa:
103:         link    a6,#0           *スタックフレーム形成
104:         movem.l d0/a0-a1,-(sp)  * {レジスタ待避
105:
106:         moveq.l #0,d0           *d0.l=表示する数
107:         move.w  value(a6),d0    *
108:         movea.l buff(a6),a1     *a1=文字列格納領域先頭
109:         lea.l   5(a1),a0        *a0=文字列格納領域最終
110:         clr.b   (a0)            *終端コード書き込み
111:
112: utoa0:  divu.w  #10,d0          *d0.lを10で割る
113:         swap.w  d0              *上位ワードと下位ワードを交換
```

```
114:            addi.w  #'0',d0         *0～9 → '0'～'9'
115:            move.b  d0,-(a0)        *1桁格納
116:            clr.w   d0              *次の除算に備える
117:            swap.w  d0              *上位ワードと下位ワードを交換
118:            bne     utoa0
119:
120: utoa1:     move.b  (a0)+,(a1)+     *左詰めにする
121:            bne     utoa1           *
122:
123:            movem.l (sp)+,d0/a0-a1  *} レジスタ復帰
124:            unlk    a6              *スタックフレーム解放
125:            rts
126:
127:            .end
```

新しいputdecルーチンの機能自体は，基本的にこれまでと変わらない。16ビット無符号数を10進左詰めで標準出力に書き出す。ただ，以前はd0.wで渡していたパラメータはスタックを介して渡すように修正されている。

リストを見ると，いままでは単独のサブルーチンであったものが3つのサブルーチンに分離されており，見かけ上の複雑さは増したように見えるかもしれない。こうなってしまったのは，処理単位を細かく一般化して，処理の階層関係をはっきりさせたためだ。数値を標準出力に書き出すという処理は，数値を指定のファイルハンドルに書き出す処理というもっと一般的なケースの一部だ。そして，この処理は数値を文字列に変換する処理と，その文字列をファイルハンドルに書き出す処理に分離することができる。

23行以下のputdecは，表示する値を取り出したら，「この値をここに出力してね」といいながら，値と標準出力のファイルハンドルをパラメータとしてfputdecというサブルーチンを呼び出す[4]。彼の仕事はこれで終わりだ。無知とはいっても，〝自分の仕事を誰に頼めばうまくやってくれるか〟だけは知っていたわけだ。

■ 4) スタックフレーム上のパラメータの位置は21行でラベル定義している。〝=〟はsetの省略形で，equのようにラベルを定義するのに使う疑似命令だ。equで定義したラベルは再定義することができないが，setで定義した場合は何度でも定義しなおすことができる。プログラムの一部でのみ使用するようなラベルを定義するのに利用する。

fputdecは，数値を文字列に変換する処理をutoa[5]というサブルーチンにさせてから，変換後の文字列を指定のファイルハンドルに出力している。数値を変換してできる文字列を格納するために一時的な作業領域が6バイト(最大5桁+終端コード)必要だから，秘密主義にしたがってlinkでローカルエリアを用意している。このときのスタックフレームの様子は，リスト中の注釈を参考にしてもらいたい。

■ 5) utoaは，Unsigned integer TO Array of characterぐらいのつもりで，無符号整数を文字の配列状データ(要するに，文字列)へ変換するという意味での命名だ。

数値→文字列変換処理を行うutoaは，変換すべきワードデータと変換後の文字列を格納する

領域へのアドレスをパラメータとして必要とする。変換処理のアルゴリズムは以前のputdecに埋め込まれていたものと同じだが，わけあって多少複雑さを増し，文字列へ変換してから左詰めにする処理が入っている。

ソースは分割して利用しよう

ようやく汎用性の高いサブルーチンの作り方が固まった。しかし，いまのところ，せっかく作った汎用ルーチンは，ソースレベルでプログラムに組み込んでアセンブルしなければ使えない。putdecのように1つの機能が複数のサブルーチンにばらされているような場合は，それぞれのサブルーチンをいっしょに移動させる必要がある。これではあんまりなので，実際に使うときの手間暇を減らす工夫をしてみたい。

ここで分割アセンブルの考え方を導入する。1本のプログラムを複数のソースに分割しておき，個別にアセンブルしたうえで，最終的にリンク時に結合するという考え方だ。長いプログラムを十分見通しがきく大きさのソースに分けて開発したり，数人でプログラムを作るようなときにも便利だ。修正が入らないかぎり，各ソースはただ1度アセンブルしておけばよいわけで，開発時間の短縮にもつながる。

この応用で，汎用のサブルーチンは個別にアセンブルしておくことにし，アセンブル前ではなく，リンクするときに組み込むことを考える。ソースの切り貼りをする手間と，すでにできているサブルーチンを何度もアセンブルしなおす時間をなくそうという魂胆だ。

ところが，通常，識別子(ラベル)を使うためには，同一のソース内のどこかで宣言しておかなければならないことになっている。経験済みと思うが，宣言されていないラベルを使おうとすると，アセンブラは困って停止してしまう。

そこで，アセンブラに〝このラベルはこのソースにはないよ〟ということを教えるための疑似命令.xrefと，〝このラベルは別のソースでも使うよ〟という疑似命令.xdefを使う。

.xref, .xdefは，

```
.xref    ラベル名
.xdef    ラベル名
```

のようにして使用する。それぞれ〝外部参照名〟の定義，〝外部定義名〟の宣言を行う疑似命令だ[6]。ラベル名は〝,〟で区切って複数並べることができる。

■ 6) 逆に，外部定義しない他のすべてのラベルは，そのソース内でのみ有効になる。他のソースに同名のラベルがあってもかまわないということだ。

使い方は単純で，ソース外のラベルを参照するとき(サブルーチン呼び出しを含む)には，そのラベル名を.xrefで宣言し，参照される側では同名のラベルを.xdefで宣言すればよい。

.xrefと.xdefの使い分けが面倒であれば，.glbl疑似命令を使ってもかまわない。これは大

域的なシンボルの宣言を行う命令で，ちょうど.xrefと.xdefの機能をあわせたものだと思えばよいだろう。外部参照と外部定義のどちらも.globlで宣言してしまうことができる。

リスト6に.xrefと.xdefの使用例を示す。a)をリスト5の頭につけてアセンブルし，b)のMAIN.Sは単体でアセンブルする。できた2つのオブジェクトファイルを，

```
A>LK MAIN PUTDEC
```

によってリンクすれば，2つのオブジェクトが組み合わされ，MAIN.Xが生成される。

リスト6-a

```
1:          .xdef   putdec
2:          .xdef   fputdec
3:          .xdef   utoa
```

リスト6-b

```
1:          .xref   putdec          *外部参照定義
2: *
3:          .include        doscall.mac
4: *
5:          .text
6:          .even
7: *
8: ent:
9:          lea.l   mysp,sp         *spの初期化
10:         move.w  #12345,-(sp)    *引数を積み
11:         bsr     putdec          *サブルーチンを呼ぶ
12:         addq.l  #2,sp           *スタック補正
13:
14:         DOS     _EXIT
15: *
16:         .stack
17:         .even
18: *
19: mystack:                        *スタック領域
20:         .ds.l   256             *ローカルエリアの分も
21: mysp:                           *計算に入れる
22:         .end
```

ここで，分割アセンブルに関連し，コラム『セクション』でいままで枕詞的に使ってきた.textなどの疑似命令の意味についてまとめておく。

COLUMN

セクション

これまで詳しい説明もなく使ってきた.text,.data,.bss,.stackは，セクションを指定する疑似命令だ。それぞれ，テキストセクション，データセクション，ブロックストレージセクション，スタックセクションの始まりを表す。セクションの選択は，次にこれらの疑似命令が現れるまで有効だ。

テキストセクションには実際に実行するプログラム，データセクションには〝初期値付き〟データ部(.dc)，ブロックストレージセクションには〝初期値なし〟データ部(.ds)，

スタックセクションにはスタック領域を置く。これらは必ずしも守らなければならないというものではなく，べつにテキストセクションにデータを置いたりしてもかまわない。実際のところ，Human68kではセクションはあまり深い意味を持たないといってもいいだろう。慣例というか，メーカーの顔を立てるぐらいのつもりでいればよい。

ただ，以前もふれたように，ブロックストレージセクションとスタックセクションに置かれた.ds命令は，実行ファイルの大きさに影響を与えないという利点がある。テキストセクションに.dsで大きな領域を確保すると，実行ファイルにはその数だけの0が埋め込まれるが，ブロックストレージセクション，スタックセクションでは〝何バイト確保されたか〟という情報のみが残り，メモリに読み込まれた時点で展開される。

また，1つのプログラムの中に複数の同一セクションがある場合は，最終的にはそれらはみんな1つにまとめられることになっている。だから，どうなんだといわれても困るが。

なお，いつも.textたちとセットで使ってきた.even疑似命令については，別コラム参照のこと。

ライブラリの作成

もう1歩突っ込もう。いまここに，リスト5＋6-aをアセンブルしてできたオブジェクトファイルがある。この中には3つのサブルーチンが含まれている。それぞれは独立しても使用できる(実際には下位のサブルーチンも必要だが)ような作りになっているから，utoaだけを使いたいという状況も考えられる。ところが，リンクはファイル単位で行われるので，utoaだけをリンクしようとしても残り2つのサブルーチンがくっついてくる。この余分なサブルーチンは，実行ファイルのサイズを大きくすることのみにしか貢献しない[7]。

■ 7) プログラムに含まれてはいるが，使われることのない部分を意味する〝dead code〟という言葉があるらしい。

こういうことが起こらないようにするためには，1つのオブジェクトファイルにはサブルーチンが1つしか存在しないように分割する必要がある。リスト5＋6-aの場合だと，サブルーチンごとにputdec.s，fputdec.s，utoa.sの3つに分割し，個別にアセンブルしておくことになるだろう(外部定義・参照に気をつけて実際に分割してみよう)。

ところが，こうしてしまうと，putdecを使いたいときに残りの2つも忘れずにリンクしてやらなければならなくなる。こんな感じだ。

```
A>LK MAIN PUTDEC FPUTDEC UTOA
```

このうち1つをリンクし忘れたりすると，LK.Xはのろのろとしばらく待たせたあげく，〝知らないラベルがある〟と怒り出すわけだ。

ここで，登場するのがライブラリファイルだ。これはいくつものオブジェクトファイルを1つにまとめたものである。上記の3つのオブジェクトファイルをまとめたライブラリファイル

(かりに〝USERLIB.L〟)を作っておけば，リンク作業は，

```
A>LK MAIN USERLIB.L
```

だけですむ[8]。LK.Xは，このライブラリファイルの中からMAIN.Oで使われているものだけを抜きとり，必要ならさらに下位のサブルーチンも探してリンクしてくれる。ライブラリの中に今回はまったく使わない他のオブジェクトがあっても無駄は生じない。LK.Xは不要なオブジェクトをリンクの対象から外すからだ。

■8) Human68kでは，ライブラリファイルの拡張子は〝.L〟または〝.A〟となる。

後は具体的なライブラリファイルの作り方を示せば，本章は終わりだ。

一般に，ライブラリはライブラリアンと呼ばれるプログラムによって作成される。もっとも，Human68kにライブラリアンLIB.Xがサポートされたのは XCのVer.2.0からで，それ以前はアーカイバAR.Xで代用していた。AR.Xはたんに複数のファイルを詰めて1つのファイルにまとめるプログラムであり，リンクの効率はあまり考えられていない。AR.Xで作成した〝ライブラリファイルもどき(拡張子はA)〟の中から必要なものを探すためには，頭から順に見ていくしかないので，リンクによけいな時間がかかることになる。対して，LIB.Xのほうはもう少しリンク作業が楽になるような形式のライブラリファイル(拡張子はL)を作ってくれる。

ライブラリアンLIB.Xを使ってライブラリファイルを作るには，

```
A>LIB　ライブラリファイル名　オブジェクトファイル名…
```

というようにライブラリファイル名の後ろにずらずらオブジェクトファイル名を並べればよい。これによって，指定されたオブジェクトファイルすべてをまとめたライブラリファイルが生成される。もしすでに同名のライブラリファイルがあった場合は，オブジェクトの追加・置き換えが行われる。

LIB.Xが手元になく，AR.Xで代用する場合には，

```
A>AR ライブラリファイル名　オブジェクトファイル名…
```

となる。プログラムの使い方自体はLIB.XもAR.Xも大差はない。

その他，LIB.X，AR.Xにはスイッチによってライブラリから一部を抽出したり，削除したり，ライブラリファイルに含まれるファイルの一覧を表示したりといった一通りの機能が揃っている。必要に応じて『アセンブラマニュアル』で調べてもらいたい。

この章では〝後で楽するために，いま苦労する〟法としてのユーザーライブラリ作成技術を紹介し，そのための汎用サブルーチンの作り方を説明してきた。汎用性・独立性にこだわると，どうしても速度・メモリ使用の効率は下がってしまうが，大きなプログラムも比較的楽に作れる手段が得られるのは大きい。

かといって，あまりに開発効率にこだわると，マシン語っぽさがなくなってしまうこともありそうだ。なんとなくできそこないの高級言語を使っているような気分になるかもしれない。

使い回しのききそうなルーチンはがっちり作ってどんどんライブラリに加えるのと同時に，必要なら独立性は捨て，思いきり実行速度にこだわって趣味の世界に走るという楽しみ方も忘れてはいけないような気がする。

COLUMN

サブルーチンからの戻り値

本文ではサブルーチンへ引数を渡すほうばかりを解説したが，逆にサブルーチンからメインルーチンへ値を返す方法も検討しておく必要があるだろう。結論からいうと，サブルーチンへ引数を渡す方法のほとんどを，戻り値を返す方法として利用することができる。ここでも独立性を追求するならスタックに値を積んで返す方法がよいのだが，これには余分なスタック操作が必要となる(考えてみるように)。

そのためかどうかは知らないが，現実にはメモリ効率や実行速度を優先し，戻り値はレジスタで返す場合が多い。この場合，サブルーチンの独立性が下がることになるが，つねに同じレジスタで値を返すといった自分なりの規則を作れば，それほどひどいことにはならない。

CHAPTER 9

「プロセス操作」という世界

CHAPTER 9

「プロセス操作」という世界

ASSEMBLER

この章では，〝プロセス〟関連の話題を取り上げる。プロセスという言葉自体は〝チャイルド(子)プロセス〟などの言葉でおなじみだろう。コンピュータ用語のプロセスは〝一連の工程〟ぐらいの意味で，あるひとまとまりの処理単位を指す言葉だ。Human68kでは，基本的に1本のプログラム＝1つのプロセスということになっている。プログラムはHuman68kによって確保されたメモリにロードされ，もろもろの管理情報を与えられた時点でHuman68kの管理下に置かれ，プロセスとなる。

Human68k上で一度に動けるプロセスはただ1つだけだが，動作中のプロセスは別のプロセスを生成・起動することができる。いわゆるチャイルドプロセスだ。あまり意識されないが，COMMAND.XやVS.X上から実行しているプログラムも，結局はみんなチャイルドプロセスである。

では，さっそく本題に入ろう。ちなみに，本章の参考資料は『プログラマーズマニュアル』第7章だ。

メモリ管理はOSの役目

メモリ上に複数のプロセスが共存するためには，1つの大前提がある。プロセス間でメモリの干渉が起きないように，誰かがメモリの使用状況(どこが使用中で，どこが空いているのか)を把握・管理していなければならない。当然，それはOSの役目だ。メモリの管理は，OSに求められるもっとも基本的な機能といえる。

Human68kでは，メモリ管理ポインタと呼ばれる双方向リスト構造をしたデータを使ってメモリを管理している。かんたんにいってしまうと，確保したメモリブロック(ひとかたまりのメモリ)の先頭にそのブロックの大きさや，前後のメモリブロックの位置を示す情報を付け加え，それによって管理するという方法だ。Human68kのメモリ管理ポインタは4ロングワード＝16バイトからなり，表1に示すような構造をしている(表中，Lはロングワードの意味)。メモリ管理ポインタは，必ずxxxxx0_Hのアドレスに置かれる。これは，実質的にメモリ確保の最小単位が16バイトであることを意味する。

+00H	1L	ひとつ前のメモリ管理ポインタのアドレス（0…前はない）
+04H	1L	このメモリブロックを確保したプロセスのメモリ管理ポインタのアドレス（0…親はない）
+08H	1L	このメモリブロックの終わり＋1のアドレス
+0CH	1L	次のメモリ管理ポインタのアドレス（0…次はない）

表1
メモリ管理ポインタ
(計16バイト)

メモリ管理ポインタ内には直前と直後のメモリ管理ポインタのアドレスが格納されているから，これを利用してメモリ管理ポインタを正逆両方向にたどることができる。a0レジスタにメモリ管理ポインタのアドレスが入っている状態から

```
movea.l (a0), a0
```

を実行すれば，a0は直前のメモリ管理ポインタを指し，

```
movea.l 12(a0), a0
```

を実行すれば，次のメモリ管理ポインタを指すようになる。

このメモリ管理ポインタのリンク(つながり)を直前，また直前というようにずっと前にたどっていくと，OS内部にある最初のメモリ管理ポインタに突き当たる[1]。このメモリ管理ポインタは，OSそのものが収められたメモリブロックを表していると考えられる。『プログラマーズマニュアル』の表記にしたがい，以下，この最初のメモリ管理ポインタの置かれたアドレスを〝fstmem〟と呼ぶことにする。なお，fstmemの位置はHuman68kのバージョンによって異なる[2]。

■ 1) Human68kのメモリ管理ポインタは，必ずメモリの低位アドレスから順にリンクしているので，あっちに行ったり，こっちに行ったりするようなことはない。

■ 2) 古い版の『プログラマーズマニュアル』には，このアドレスがさも固定であるかのように具体的なアドレスで示してあるが，無視すること(しかも，示されているアドレスはまちがっている)。

Human68kにおける，メモリ管理ポインタによるメモリ管理のイメージを次ページの図1に示す。最初のa)では，A～Cの3つのメモリブロックが確保されている。図1ではとくに明記していないが，各メモリブロックの頭にはそれぞれメモリ管理ポインタがある。いま，それぞれのメモリ管理ポインタは，

```
0←A⟷B⟷C→0
```

というリンクを形成している(0はリンクの末端を示すものとする)。Cの後ろが空きメモリである。空きメモリは，どのメモリ管理ポインタにも属さない部分であり，どこまでが空きメモリかは別の情報(図ではmemendとした)で示す必要がある[3]。

■ 3) 余談ながら，Human68kのRAMディスクドライバは，memendに相当するHuman68k内部のワークエリアを操作することで，メインメモリの後ろからメモリを削り取って使っている。

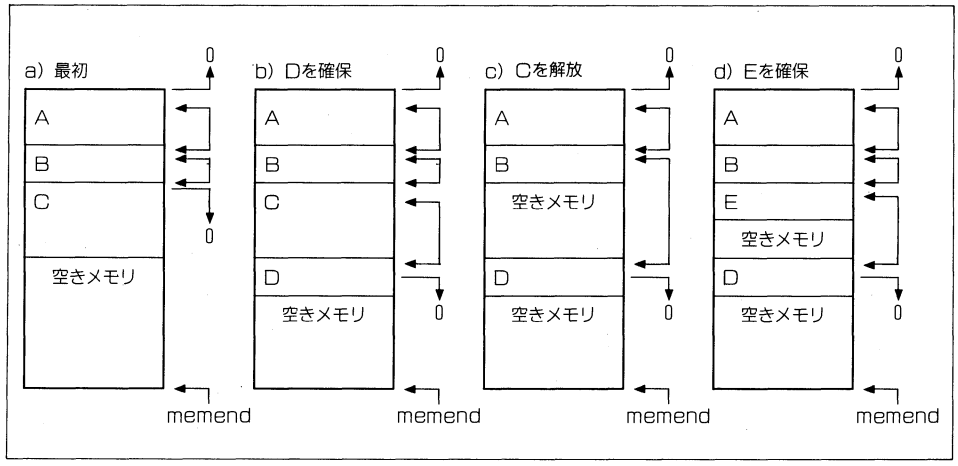


図1
メモリ管理のイメージ

この状態からあらたなメモリブロックDを確保すると，b)のようになる。空きメモリの先頭を削ってDに割り当てた形だ。その後Dを解放すれば，a)に戻るわけだが，Dを確保したままでCだけを解放することもできる。c)の状態がそれで，BとC，CとDの間のリンクを断ち切って，つなぎあわせた格好をしているのがわかる。

空いてしまったメモリは無駄にはならない。次に確保するメモリがここに収まる大きさであれば，d)のようにこの隙間を埋める形でメモリの確保が行われる。さらに残った隙間もまた後で使われるチャンスがある。Human68kはメモリの確保を行うときに，必ず，メモリブロックの隙間に収まるかどうかをチェックする[4]。

■ 4) 表1で確認してもらいたいのだが，〝次のメモリ管理ポインタのアドレス〟から〝このメモリブロックの最終アドレス＋1〟を引けば，隙間の大きさが求められる。

ただ，Human68kの場合，隙間のチェックをメモリの低位から順に行うので，必ずしも最高の効率でメモリが使用できるわけではない。たとえば，20K，10Kの順序で隙間が存在するときに，10Kのメモリを確保しようとすると，最初に見つけた20Kの隙間を使ってしまい，結局10Kの隙間2つが残るという結果になる。

メモリブロックの表示

メモリ管理ポインタは通常のプログラムではとくに気にする必要がないのだが，ついでなので，リスト1にメモリ管理ポインタのリンクをたどり，fstmem以降のメモリブロックのアドレスをすべて表示するプログラムMB.Xを示す。XCやHuman68k　Ver.2.0などについている

PROCESS.Xをずっとかんたんにしたようなものだ(実行結果を見比べてみるとよい)。リスト2のITOH.Sとリンクして実行形式ファイルを作成するようになっている。ITOH.S中のサブルーチンitohは，以前作った数値→16進文字列変換サブルーチンを，スタックを介してパラメータを渡すように作り直したものだ。これからもしばしば使うのでライブラリに加えておいてもらいたい。

リスト1
MB.S

```
1: *       メモリ上の全メモリブロックの位置を表示する
2: *
3: *       作成法：as mb
4: *               as itoh
5: *               lk mb itoh
6: *
7:         .include        doscall.mac
8:         .include        const.h
9: *
10:        .xref   itoh            *外部参照
11: *
12: *      メモリ管理ポインタの構造
13: *
14: PREVMEM        equ     0
15: OWNERPROC      equ     4
16: MBEND          equ     8
17: NEXTMEM        equ     12
18: *
19:        .text
20:        .even
21: *
22: ent:
23:        lea.l   mysp,sp         *spの初期化
24:
25:        clr.l   -(sp)           *スーパーバイザモードに
26:        DOS     _SUPER          *  切り替える
27:                                *d0 = ssp
28:        move.l  d0,(sp)         *sspを待避
29:
30: loop1: move.l  PREVMEM(a0),d0  *d0 = 直前のメモリ管理ポインタ
31:        beq     loop2           *0なら先頭
32:        movea.l d0,a0           *a0 = 直前のメモリ管理ポインタ
33:        bra     loop1           *先頭に達するまで繰り返す
34:
35: loop2: pea.l   buff            *メモリ管理ポインタの
36:        move.l  a0,-(sp)        *  先頭アドレスを
37:        bsr     itoh            *  16進文字列変換して
38:        addq.l  #8,sp           *  buff以降にセットする
39:
40:        move.b  #'-',buff+8     *つなぎの'-'を書き込む
41:
42:        pea.l   buff+9          *メモリブロックの
43:        move.l  MBEND(a0),d0    *  最終アドレスを
44:        subq.l  #1,d0           *  16進文字列に変換して
45:        move.l  d0,-(sp)        *  buff+9以降にセットする
46:        bsr     itoh            *
47:        addq.l  #8,sp           *
```

```
48:
49:             pea.l   buff            *XXXXXXXX-XXXXXXXXまでを
50:             DOS     _PRINT          *  まとめて表示する
51:             addq.l  #4,sp           *
52:
53:             pea.l   crlfms          *改行する
54:             DOS     _PRINT          *
55:             addq.l  #4,sp           *
56:
57:             move.l  NEXTMEM(a0),d0  *d0 = つぎのメモリ管理ポインタ
58:             movea.l d0,a0           *←注意：フラグは変化しない
59:             bne     loop2           *d0が０でなければ繰り返す
60:
61:             DOS     _SUPER          *ユーザーモードに
62:             addq.l  #4,sp           *  戻す
63:
64:             DOS     _EXIT           *終了
65: *
66:             .data
67:             .even
68: *                    01234567890123456
69: buff:       .dc.b   '12345678-12345678',0   *表示用文字列格納領域
70: crlfms:     .dc.b   CR,LF,0                 *改行コード
71: *
72:             .stack
73:             .even
74: *
75: mystack:                            *スタック領域
76:             .ds.l   1024            *
77: mysp:
78:             .end    ent             *実行開始アドレスはent
```

リスト2
ITOH.S

```
 1:             .xdef   itoh            *外部定義
 2: *
 3:             .text
 4:             .even
 5: *
 6: *itoh(value,buff)
 7: *機能： 32ビット整数を16進８桁の文字列に変換する
 8: *レジスタ破壊：ccr
 9: *
10: *           ex)
11: *                   pea.l   buff
12: *                   move.l  #$12345678,-(sp)
13: *                   bsr     itoh
14: *                   addq.l  #8,sp
15: *                     :
16: *           buff:   .ds.b   8+1
17: *
18: *
19: value       =       8
20: buff        =       12
21: *
22: itoh:
```

```
23:          link     a6,#0
24:          movem.l  d0-d2/a0,-(sp)
25:
26:          move.l   value(a6),d0      *値
27:          movea.l  buff(a6),a0       *文字列格納アドレス
28:
29:          moveq.l  #8-1,d2           *以下を８回繰り返す
30:
31: itoh0:   rol.l    #4,d0             *d0.lを左に４ビット回転する
32:          move.b   d0,d1             *d0の下位バイトをd1に取り出し
33:          andi.b   #$0f,d1           *  下位４ビットを残してマスクする
34:          addi.b   #'0',d1           *ここで数値から16進を表す文字へ
35:          cmpi.b   #'9'+1,d1         *  変換する
36:          bcs      itoh1             *  0～9の場合は'0'を足すだけだが
37:          addq.b   #'A'-'0'-10,d1    *  A～Fの場合はさらに補正が必要
38:
39: itoh1:   move.b   d1,(a0)+          *変換した文字をしまう
40:
41:          dbra     d2,itoh0          *繰り返す
42:
43:          clr.b    (a0)              *文字列終端コードを書き込む
44:
45:          movem.l  (sp)+,d0-d2/a0
46:          unlk     a6
47:          rts
```

MB.Xのアルゴリズムは，fstmemから順次メモリブロックの先頭アドレスと最後のアドレスを表示してリンクをたどる処理を繰り返すというだけの単純なものだ。最後のメモリブロックに到達したかどうかは，メモリ管理ポインタ内の〝次のメモリ管理ポインタのアドレス〟が0かどうかで判断する。ただ，fstmemが何番地か決められないため，メインの処理に先立ってfstmemの位置を探さなければならない。これは，メモリ管理ポインタをメモリ低位方向へたどることで行う。

問題はもう１つある。fstmemはOS内であり，ユーザー空間の外側のスーパーバイザ空間に位置している。通常のアプリケーションの走行環境であるユーザーモードではアクセスできない場所にあるわけだ[5]。そこで，一時的にスーパーバイザモードに移行する必要が出てくる。これにはDOSコールsuperを使う。リスト１の25行のようにスタックにロングワードで０を積んで[6]superを呼び出すと，スーパーバイザモードに移行することができる。superから戻った時点で，d0.lには直前までのssp(スーパーバイザスタックポインタ)が入っており[7]，これは後でユーザーモードに戻るときに必要な情報なので，どこかに保存しておかなければならない。リスト１では素直にスタックにしまっている。

■ 5) 強引にアクセスしようとしても，ユーザーモードから見れば〝そこにはメモリが実装されていないのと同じこと〟なので，バスエラーが発生する。
■ 6) 実際にはclr.lを使っている。
■ 7) すでにスーパーバイザモードなのにふたたびスーパーバイザに切り替えようとするとエラーになり，d0.lは負の値を採る。

スーパーバイザモードに移行し，安心して悪さができるようになったところで，メモリ管理ポインタをたぐってfstmemを探す。プログラムが起動した時点で，a0レジスタにはプログラムの置かれたメモリブロックのメモリ管理ポインタが入っていることになっているから，そこを基点にしてメモリ管理ポインタをひたすら前にたどっていく(30～33行)。

fstmemが見つかったら，16進変換サブルーチンを適当に使ってメモリブロックの最初と最後のアドレスを表示する。1つのメモリ管理ポインタに対する処理がすんだら，リンクをたどって続くメモリブロックを処理する(35～59行)。

もし続くメモリ管理ポインタがなければ，それで処理はおしまいだ。終了直前に礼儀正しくスーパーバイザモードからユーザーモードに戻しておきたい。この切り替えは，さきほどスーパーバイザモードに移行したときにDOSコールsuperから戻ってきたsspの元の値をスタックに積んで再度superを実行することで行う[8]。

■ 8) リスト1ではいきなりsuperを呼び出しているように見えるが，28行ですでにsspをスタックに積んだ形になっているので，これでよい。

メモリの確保と解放

OSがメモリを管理している以上，ユーザープログラムはOSに与えられたメモリ以外に手を出してはいけない。もし，実行中により多くのメモリが必要になったら，OSに要求し，確保してから使う。Human68kには，メモリ確保に関するDOSコールとしてmalloc, free, setblockの3つが用意されている。それぞれ，メモリブロックの確保，メモリブロックの解放，メモリブロックのサイズ変更を行うものだ。

●malloc

mallocは，次のようにして呼び出す。

```
move.l    確保したいバイト数, －(sp)
DOS       _MALLOC
addq.l    #4, sp
```

バイト数にメモリ管理ポインタの分を入れる必要はない。自動的にメモリ管理ポインタで使う16バイトだけよけいにメモリが確保される。また，バイト数は下位の24ビットのみが有効だ(上位8ビットは0と見なされる)。X68000のメインメモリは最大で12Mバイトだから，これで十分なのはいうまでもない。

メモリの確保に成功した場合は，d0.lに確保して使えるようになったメモリのアドレスを入れて戻る。これはメモリ管理ポインタの直後にあたり，実際に使えるメモリの先頭アドレスだ。

メモリ不足で指定しただけのメモリが確保できなかった場合は，d0.lに81000000_H＋確保できる最大のバイト数が返ってくる[9]。また，完全に確保不可能な場合は戻り値は$8200000x_H$になっている(xは不定だそうだ)。メモリ管理ポインタを格納するための16バイトすらとれなかっ

たということだ。

■ 9) mallocで一度に確保できる最大のバイト数は，空きメモリの合計ではなく，最大の大きさの空きメモリブロックの大きさであることに注意したい。

どちらにしろ，8xxxxxxx$_H$は2の補数として見れば負の値であり，他のDOSコール同様，d0.lが負であればエラーと判断してよいことになる。

mallocの返すエラーを積極的に利用すると，リスト3[10]のようなこともできる。ここに示すサブルーチンallmemはとれるかぎりの大きなメモリブロックを確保し，その先頭アドレスをd0.lに入れて戻る。確保できなかったら，d0.lは負の値になる。

■10) リスト3のALLMEM.SはITOH.S同様ライブラリとして使うためのものだから，単独で実行形式ファイルにしても意味がない。

リスト3
ALLMEM.S

```
  1:          .include        doscall.mac
  2: *
  3:          .xdef    allmem          *外部定義
  4: *
  5:          .text
  6:          .even
  7: *
  8: *allmem()
  9: *機能：  確保できる最大のメモリブロックを確保する
 10: *        d0.lに確保した先頭アドレスを持って戻る
 11: *        d0.lが負の場合はエラー
 12: *レジスタ破壊：d0.l,ccr
 13: *
 14: *        ex)
 15: *                bsr     allmem
 16: *                tst.l   d0
 17: *                bmi     error
 18: *
 19: allmem:
 20:          move.l  #$ffffff,-(sp)
 21:          DOS     _MALLOC
 22:          andi.l  #$ffffff,d0
 23:          move.l  d0,(sp)
 24:          DOS     _MALLOC
 25:          addq.l  #4,sp
 26:          rts
 27:
 28:          .end
```

allmemでは，最初に絶対確保できない大きさのメモリを確保しようとする。当然エラーになるから，戻り値のd0.lは81xxxxxx$_H$か8200000x$_H$のどちらかだ。81xxxxxx$_H$だった場合は下位24ビットを残して上位ビットをマスクすると，確保できる最大バイト数が得られる。この値を使ってもう一度mallocを実行すれば，まちがいなく最大の大きさのメモリブロックが確保できるわけだ。8200000x$_H$だった場合は，上位をマスクしてもう一度mallocしてもやはりエラー

になるわけで，サブルーチンから戻った時点でメインルーチン側でエラーチェックすれば判別できる。

●free

mallocで確保したメモリブロックを解放するにはfreeを使う。mallocの戻り値をスタックに積んで呼び出すだけだ。

```
move.l    確保したメモリのアドレス,－(sp)
DOS       _FREE
addq.l    #4,sp
```

変なアドレスを指定した場合など，エラーのときはd0.lに負の値を返すが，mallocの戻り値をそのまま使えば，エラーが起こることは考えられない。

なお，freeを使って明示的に解放しなくても，exitやexit2でプロセスが終了したときに，そのプロセスが確保したメモリはすべて解放されることになっている。だが，礼儀としては確保したメモリはきちんと解放してから終了すべきだろう。

●setblock

残るsetblockは，mallocで確保したメモリブロックの大きさを変更するのに使う。小さくするのにも大きくするのにも使える。呼び出し方法は次のとおりだ。

```
move.l    変更したいバイト数,－(sp)
move.l    確保したメモリのアドレス,－(sp)
DOS       _SETBLOCK
addq.l    #8,sp
```

setblockの実行がうまくいけば，そのメモリブロックは指定した長さに変更されている。メモリブロックを小さくする分にはエラーは起きようもないが，大きくしようとしてできなかった場合はmalloc同様の値でd0.lにエラーを返してくる。

さて，Human68kのチャイルドプロセス起動時のメモリ割り当てには変な癖がある。もっとも大きな空きメモリブロックを全部確保して，プログラムを読み込む領域に割り当ててしまうのだ[11]。メモリ上にHuman68kとCOMMAND.Xだけがある状態からユーザープログラムを起動すると，そのプログラムにはもっとも大きな空きメモリブロック，つまり，フリーエリアの残り全部が割り当てられる。家を買ったら広い庭がオマケについてきたようなもので，プログラム本体はその頭の部分にちょこんと置かれた形になっている。この〝庭〟はmallocなんかしなくても，当然自由に使ってよい。すでに確保されているのだ。

■11) リスト1のMB.Xを走らせた人なら，最後にやけに大きなメモリブロックがあるのに気づいたはずだ。

しかし，広いメモリを与えられたいうことは，そのメモリの管理がユーザープログラムにまかされたことを意味する。どこを使っているのかを自分で把握している必要があるわけだ。OSの苦しみをいやというほど味わわされることになるだろう。そこで，いったん余分なメモリを

OSに返却してしまい，それから必要に応じてmallocで確保して使うということがよく行われる。「こんなメモリはいらない」と戻しておいて，「でも，ちょっとちょうだいね」と少しずつ取り返すのだ。

余分なメモリの切り離しには，さきほどのsetblockを使い，〝プログラム自身が置かれているメモリブロック〟を〝プログラム本体を収めるぎりぎりの大きさ〟に変更する。リスト4のサブルーチンmemoffが，この場合の常套的な手順だ。

リスト4
MEMOFF.S

```
 1:          .include        doscall.mac
 2: *
 3:          .xdef   memoff          *外部定義
 4: *
 5:          .text
 6:          .even
 7: *
 8: *memoff()
 9: *機能：  プログラム本体以降の余分なメモリを開放する
10: *       注意：プログラム起動直後、a0,a1が破壊される前に
11: *             呼び出すこと
12: *レジスタ破壊：a0.l,a1.l,d0.l,ccr
13: *
14: memoff:
15:          lea.l   16(a0),a0       *a0 = メモリ管理ポインタの直後
16:                                  *a1 =  プログラム本体の直後
17:          suba.l  a0,a1           *a1 - a0 = プログラムに必要な
18:                                  *          メモリサイズ
19:          move.l  a1,-(sp)        *メモリブロックの新サイズ
20:          move.l  a0,-(sp)        *メモリブロックの先頭アドレス
21:          DOS     _SETBLOCK       *メモリブロックサイズ変更
22:          addq.l  #8,sp           *スタック補正
23:
24:          rts
25:
26:          .end
```

setblockに渡すパラメータとして，〝プログラム自身が置かれているメモリブロックのメモリ管理ポインタの直後のアドレス〟と，〝プログラム本体の大きさ〟が必要だから，まず，これらを求める。前者はプログラム起動時のa0レジスタ(＝プログラムの置かれたメモリブロックのメモリ管理ポインタの先頭アドレス)に，メモリ管理ポインタで使っているメモリサイズ16を加えれば求められる。リスト4では16行の

```
lea.l     16(a0),a0
```

によって，このアドレスをa0に求めている。また，起動時のa1レジスタにはプログラム本体の終わり直後のアドレスが入っているので，a1からいま求めたa0の値を引けば，プログラムで使っているメモリの大きさが求まる。後はこれらをsetblockに渡すだけだ。

プロセス管理とは

Human68kでは，プロセスは〝プロセス管理ポインタ〟[12]と呼ばれる240バイトの構造体状のデータでプロセスを管理している。メモリ管理ポインタのように相互にリンクするようなことはなく，たんに管理情報がひとまとめにしてあるだけのものだ。

■12)『プログラマーズマニュアル』を見ると，プロセス管理ポインタの同義語として〝PSP〟，〝PDB〟という2つの略語が未定義のまま使われているようだ。前者はMS-DOSで使われている言葉で〝Program Segment Prefix〟の略だったと記憶している。〝プログラムが格納された領域の頭書き〟ぐらいの意味だ。後者はマニュアルではひとこともふれられていないが，きっと〝Process(Program?)Data Block〟かなんかだろう(自信はないが)。

プロセス管理ポインタは表2のような構造をしており(表中，L，W，Bはそれぞれロングワード，ワード，バイトを意味する)，メモリ管理ポインタの直後に置かれる。慣例にしたがい，プロセス管理ポインタ内のデータはメモリ管理ポインタの先頭からの相対アドレス(アドレスの差)で示してある。プロセス管理ポインタの直後から実際のプログラムが格納される。プログラムの先頭にentというラベルをつけたとすると，プロセス管理ポインタの先頭はent−F0$_H$で直接指定できる[13]。

■13)メモリ管理ポインタも同様にent−100$_H$で指定できる。これはプログラム起動時のa0レジスタと同じ値になるはずである。

ここでは，プロセス管理ポインタの中身自体を理解する必要はない。「プログラムの頭にはこんな情報がついているんだなー。プロセスを管理するのに必要そうな情報が並んでいるなー」という程度の認識で十分だ。実際，この内容を直接書き換えることはユーザープログラムには許されていないし，読んでみたところで役に立つような情報も少ない。

ただ，プロセス管理ポインタの80$_H$番地目以降に起動されたプログラム自身のドライブ名，パス名，ファイル名が格納されているのはちょっとおいしい。この情報は，プログラムの実行に必要なデータファイルがある場合に重宝する。データファイルを実行ファイルと同じディレクトリに置いておくことにすれば，このプロセス管理ポインタ内の情報からデータファイルのあるディレクトリ(結局は，実行ファイルのあるディレクトリ)がすぐ求められる。これで，〝××というデータファイルのあるディレクトリに移動してから実行してください〟などという間抜けなプログラムを作らなくてすむだろう。

●exec

プロセスに関するDOSコールはいくつかあるが，そのうちもっとも利用価値の高いのがプログラム中から他のプログラムを起動するexecだ。このDOSコールひとつで子プロセスの生成が比較的かんたんに行える。

●メモリ管理ポインタ		
+00H	4L	
●プロセス管理ポインタ		
+10H	1L	プロセスの環境変数領域アドレス（＝起動時のa3）
+14H	1L	プロセスが終了した際の戻りアドレス
+18H	1L	プロセスがCTRL+Cによって中断された際の戻りアドレス
+1CH	1L	プロセスがエラーによって中断された際の戻りアドレス
+20H	1L	プロセスに与えられたコマンドラインのアドレス（＝起動時のa2）
+24H	12B	プロセスのファイルハンドルの使用状況
+30H	1L	プロセスのbssの先頭アドレス
+34H	1L	プロセスのヒープの先頭アドレス（bssと同じ）
+38H	1L	プロセスの初期スタックアドレス（＝プログラムの終わり＋1＝起動時のa1）
+3CH	1L	親プロセスのuspの値
+40H	1L	親プロセスのsspの値
+44H	1W	親プロセスのsrの値
+46H	1W	中止時のsrの値
+48H	1L	中止時のsspの値
+4CH	5L	trap10～14の例外ベクタ
+60H	1L	プロセスのフラグ（0…親，－1…OSから起動された）
+64H	28B	未使用（Ver.2.0では一部使用）
+80H	1L	このプロセスに対応するファイルのドライブ名（ドライブ名＋':'）
+82H	1L	このプロセスに対応するファイルのパス名　('¥～¥',0)
+C4H	1L	このプロセスに対応するファイルのファイル名　('～.X',0)
+DCH	36B	未使用
●プログラム本体		
+100H	～	

表2
プロセス管理ポインタ

なお，execで子プロセスを起動するときにはHuman68k内部でmallocが実行され，子プロセス用のメモリが確保される。このため，子プロセス起動の前に，さきほどのmemoffのような処理で子プロセス用の空きメモリを作っておかなければならない。

execには0～4のモードが用意されており，モードによってパラメータの個数や意味が異なる。Human68k Ver.2.0では機能が若干拡張されているが，いまはVer.1.0と共通の部分だけを取り上げる。

モード0はプログラムのロード・実行をいっきに行う。

```
move.l      環境変数領域のアドレス,－(sp)
move.l      コマンドラインパラメータ,－(sp)
move.l      起動ファイル名,－(sp)
clr.w       －(sp)          ＊モード0
DOS         _EXEC
lea.l       14(sp),sp
```

環境変数領域のアドレスに0を指定すると，子プロセスは親の環境をそのまま引き継ぐ。特別な環境で子プロセスを走らせようというのでないかぎり，いつも0にしておけばよいだろう。コマンドライン引数は任意の文字列(の先頭アドレス)で指定する。子プロセス側では，このア

ドレスを起動時のa2レジスタで受け取ることになる。起動ファイル名はやはり適当な領域にファイル名をセットして，その先頭アドレスで指定する。最後に，スタックに積んでいる0はモード番号の0だ。

リターン時のd0.lが負であればエラーが発生した(子プロセスが起動できなかった)ことを意味する。d0.lが正であれば，それは子プロセスの終了コードである[14]。

■14) 子プロセスの終了コードはDOSコールwaitを使っても取得できる。waitはたんに
DOS _WAIT
で呼び出し，最後に実行した子プロセスの終了コードをd0.lに返す。

モード1はロードのみを行い，実行アドレスをd0.lに返す。呼び出し時のパラメータは，モード番号を除けばモード0と変わらない。

```
move.l      環境変数領域のアドレス,－(sp)
move.l      コマンドライン引数,－(sp)
move.l      起動ファイル名,－(sp)
move.w      #1,-(sp)        ＊モード1
DOS         _EXEC
lea.l       14(sp),sp
```

モード1でロードした後実行するには，モード4を使う。モード1の戻り値をスタックに積んで渡すだけだ。

```
move.l      実行アドレス,－(sp)
move.w      #4,－(sp)       ＊モード4
DOS         _EXIT
addq.l      #6,sp
```

モード2は，モード0，1実行前の下準備に使う。具体的にはコマンドラインをファイル名と引数に分解し，同時に指定した環境変数領域内の環境変数pathを参照してファイルをディスクから検索する。ファイル名は，この検索結果にしたがって絶対パスで返される。

```
MORE CONFIG.SYS
```

のような文字列を，

```
A:¥BIN¥MORE.X
CONFIG.SYS
```

に分けてくれるわけだ。モード2の結果を使ってあらためてモード0か1を実行すれば，pathが通っているディレクトリにあるプログラムならどれでもかんたんにロード・実行できる。

```
move.l      環境変数領域のアドレス,－(sp)
move.l      コマンドライン引数の格納領域,－(sp)
move.l      与えるコマンドライン兼ファイル名格納領域,－(sp)
move.w      #2,－(sp)       ＊モード2
```

```
DOS     _EXEC
lea.l   14(sp),sp
```

環境変数領域は，モード0，1同様，0を使った手抜き指定が可能だ。また，与えるコマンドラインがファイル名で上書きされる点には注意してもらいたい。

残るモード3はメモリの範囲を直接指定して，そこにプログラムをロードする。あまり使われないだろうから，ここでは相手にしない。X-BASICのFNCファイルの読み込みなどに利用されているようだ。

子プロセスの起動

execはこのまま使ってもそれなりに便利なのだが，若干手間がかかるので，ここでコマンドライン文字列を与えると，pathの検索からロード・実行まで行うようなサブルーチンを作っておく。サブルーチン名はchildとしよう。リスト5だ。大筋はexecのモード2と0を連続実行しているだけで，変なことはしていないつもりだから，しっかり読み切ってもらいたい。

リスト5
CHILD.S

```
1:          .include        doscall.mac
2: *
3:          .xdef   child           *外部定義
4: *
5: *        EXECモード
6: *
7: LOADEXEC         equ     0
8: PATHCHK          equ     2
9: *
10:         .text
11:         .even
12: *
13: *child(cmd)
14: *機能： 与えられたコマンドラインに従って
15: *       プログラムをチャイルドプロセスとして起動する
16: *       d0.lにEXECの終了コードを持って戻る
17: *       d0.lが負の場合はエラー
18: *レジスタ破壊：d0.l,ccr
19: *
20: *       ex)
21: *               pea.l   cmd
22: *               bsr     child
23: *               addq.l  #4,sp
24: *               tst.l   d0
25: *               bmi     error
26: *                   :
27: *       cmd:    .dc.b   'command /cdir',0
28: *
29: nambuf  =       -512
30: cmdlin  =       -256
31: str     =       8
```

```
32: *
33: child:
34:         link    a6,#-512        *512バイトのローカルエリア
35:         movem.l d1-d7/a0-a6,-(sp)
36:
37:         movea.l str(a6),a1      *与えられた文字列を
38:         lea.l   nambuf(a6),a0   *  ローカルエリアに
39:         move.w  #255-1,d0       *  最大255バイト
40: chld0:  move.b  (a1)+,(a0)+     *  コピーしておく
41:         dbeq    d0,chld0        *∵上書きされるから
42:         clr.b   (a0)            *念のための終端コード
43:
44:         clr.l   -(sp)           *自分の環境
45:         pea.l   cmdlin(a6)      *パラメータ部格納領域
46:         pea.l   nambuf(a6)      *コマンドライン兼
47:                                 *  フルパス名格納領域
48:         move.w  #PATHCHK,-(sp)  *PATH検索
49:         DOS     _EXEC           *
50:         tst.l   d0              *d0.lが負なら
51:         bmi     chld1           *  エラー
52:
53:         move.w  #LOADEXEC,(sp)  *ロード&実行
54:         DOS     _EXEC           *
55: chld1:  lea     14(sp),sp       *スタック補正 4*3+2バイト
56:
57:         movem.l (sp)+,d1-d7/a0-a6
58:         unlk    a6
59:         rts
60:
61:         .end
```

まず，サブルーチンの頭で512バイトのローカルエリアを用意している[15]。execモード2で分解されるファイル名とコマンドライン引数を格納するのに256バイトずつ使う。リスト中では，それぞれの先頭アドレスはスタックフレーム上の位置〝nambuf(a6)〞，〝cmdlin(a6)〞で表されている。以下，これらをたんにnambuf，cmdlinと表記する。

■15) childを呼び出すメインプログラム側では，この512バイトも計算に入れてスタック領域を確保する必要がある。

ローカルエリア確保後，d0とspを除くレジスタをスタックに待避する[16]。それからサブルーチンに渡されたコマンドライン文字列の先頭アドレスをa0に取り出し，その文字列をnambufにコピーする。execモード2では，与えた文字列を上書きする形でファイル名が返されるため，メインから渡された文字列を保存する意味でこういうことをしている。このコピーならかりに上書きされて失われても，メインルーチンに影響は出ない。

■16) execで子プロセスを実行したときには，sp，sr以外のレジスタは破壊される。

nambufへの文字列コピーは39～41行のループで行っている。nambufが256バイトしかない

ので，それ以上は絶対に転送しないように細工してある。

ここで使っているdbeqはdbraのバリエーションで，eqはbeqのeqと同じ〝もしZフラグが立っていれば〟という条件を表す[17]。dbeqは，次のような動作をする。

■17) 当然，dbneやdbccやdbcsなどもある。

1) Zフラグが立っていれば，即座にループを抜け，dbeqの直後の命令の実行に移る。

2) そうでなければ，指定されたデータレジスタを1減らす。

3) 結果が−1であればループを抜ける。

4) そうでなければ，指定アドレスに分岐し，ループを繰り返す。

2)〜4)はdbraの動作と同じで，その前にフラグによる条件判断がついたものと思えばよい。感じとしては，

```
        beq     skip
        dbra    dn, ～
skip:   ～
```

をひとまとめにしたようなものだ。〝ある処理を一定回数繰り返すが，途中で特定の条件が成り立っていたらループを抜ける〟のに便利な命令だ。ここでは〝メモリ転送を255回繰り返すが，途中で文字列の終端コードを見つけたら，それ以上転送せずにループを抜ける〟ために利用している。

nambufにコマンドライン文字列をセットしたら，以下，環境変数領域(手抜きの0)，cmdlin，nambufをスタックに積んで，execモード2，0を連続実行する。モード2の呼び出しでスタックに積んだ引数がモード0でもそのまま使えるので，途中のスタック補正は省略し，スタックトップのモード番号を直接2から0に置き換えている(53行)。

execからの戻り値d0.lは，そのままサブルーチンからの戻り値としてメインルーチンに返す。メイン側ではd0.lの正負からエラーの有無を判定することになる。

childの使用例をリスト6に示す。このプログラムCHLDTEST.Xは，たんにチャイルドプロセスとしてATTRIB.Xを起動するだけのものだ。起動するコマンドはプログラム中に埋め込まれているので，適当に変更してみるとよいだろう。メモリの切り離しにリスト4のmemoffを使っているので，実行ファイルは，

```
A>LK  CHLDTEST  CHILD  MEMOFF
```

で作成すること。

リスト6
CHLDTEST.S

```
1: *       チャイルドプロセスを単に起動してみる
2: *
3: *       作成法：as chldtest
4: *               lk chldtest child memoff
5: *
6:         .include        doscall.mac
7:         .include        const.h
```

```
 8: *
 9:          .xref   child           *外部参照
10:          .xref   memoff          *
11: *
12:          .text
13:          .even
14: *
15: ent:
16:          lea.l   mysp,sp         *spの初期化
17:
18:          bsr     memoff          *余分なメモリを開放する
19:
20:          pea.l   cmd             *チャイルドプロセス起動
21:          bsr     child           *
22:          addq.l  #4,sp           *
23:          tst.l   d0              *エラー？
24:          bmi     error           *  そうならエラー終了
25:
26:          DOS     _EXIT           *終了
27: *
28: error:                           *エラー終了
29:          pea     errmes
30:          DOS     _PRINT
31:          addq.l  #4,sp
32:
33:          move.w  #1,-(sp)
34:          DOS     _EXIT2
35: *
36:          .data
37:          .even
38: *
39: cmd:     .dc.b   'attrib *.*',0          *起動コマンド
40: errmes: .dc.b   'コマンドが起動できません',0
41: *
42:          .stack
43:          .even
44: *
45: mystack:                         *スタック領域
46:          .ds.l   1024
47: mysp:
48:
49:          .end    ent             *実行開始アドレスはent
```

さて，サブルーチンchildはCOMMAND.Xを経由することなく，直接プログラムを子プロセスとして起動する。そのため，COMMAND.Xの内部コマンドや，リダイレクト，パイプなどの機能を利用することはできない。もし，これらの機能が使いたければ

```
COMMAND  DIR
COMMAND  DIR > $$$
COMMAND  DIR | MORE
```

のような文字列をchildに渡さなければならない。それが面倒であれば，

```
DIR | MORE
```

という文字列を渡すと，

COMMAND DIR | MORE

に変換してからchildを呼び出すようなサブルーチンを作っておけばよい。リスト7にその一例を示しておく。

リスト7
COMMAND.S

```
 1:          .include        doscall.mac
 2: *
 3:          .xdef   command         *外部定義
 4:          .xref   child           *外部参照
 5: *
 6:          .text
 7:          .even
 8: *
 9: *command(cmd)
10: *機能：  command.xをチャイルドプロセスとして起動する
11: *        d0.lにEXECの終了コードを持って戻る
12: *        d0.lが負の場合はエラー
13: *レジスタ破壊：d0.l,ccr
14: *
15: *        ex)
16: *                pea.l   cmd
17: *                bsr     command
18: *                addq.l  #4,sp
19: *                tst.l   d0
20: *                bmi     error
21: *                    :
22: *        cmd:    .dc.b   'dir | more',0
23: *
24: temp    =       -256
25: str     =       8
26: *
27: command:
28:          link    a6,#-256
29:          movem.l a0-a1,-(sp)
30:
31:          lea.l   temp(a6),a0     *一時領域に
32:          lea.l   comstr,a1       *  'command ' の文字列を
33:          move.w  #255-1,d0       *  コピーする
34: com0:    move.b  (a1)+,(a0)+     *
35:          dbeq    d0,com0         *
36:
37:          subq.l  #1,a0           *a0は行き過ぎている
38:
39:          movea.l str(a6),a1      *与えられた文字列を
40: com1:    move.b  (a1)+,(a0)+     *  それに連結する
41:          dbeq    d0,com1         * (合計で255バイトまで)
42:
43:          clr.b   (a0)            *念のための終端コード
44:
45:          pea.l   temp(a6)        *"command "+strを
46:          bsr     child           *  実行する
47:          addq.l  #4,sp           *
48:
49:          movem.l (sp)+,a0-a1
```

```
 50:          unlk    a6
 51:          rts
 52: *
 53:          .data
 54:          .even
 55: *
 56: comstr: .dc.b    'command ',0
 57:
 58:          .end
```

256バイトのローカルエリアに58行で用意した〝command〟という文字列＋メインから渡された文字列を作り上げ，childを呼び出すだけだ。プログラム上のテクニックとしては，2つのdbeqの微妙な使い方をチェックしておいてもらいたい。

プロセス操作の応用

最後に，childを使った実用プログラムを1つ示す。リスト8のRDERR.Xは，〝標準エラー出力をファイルにリダイレクトして子プロセスを実行する〟プログラムだ。エラーメッセージなど，標準エラー出力に出力されるメッセージをファイルに落としたいときに利用してもらいたい。

リスト8
RDERR.S

```
  1: *       標準エラー出力をリダイレクトする
  2: *
  3: *       作成法：as rderr
  4: *               lk rderr child memoff
  5: *
  6:         .include        doscall.mac
  7:         .include        const.h
  8: *
  9:         .xref   child           *外部参照
 10:         .xref   memoff          *
 11: *
 12:         .text
 13:         .even
 14: *
 15: ent:
 16:         lea.l   mysp,sp         *spの初期化
 17:
 18:         bsr     memoff          *余分なメモリを開放する
 19:
 20:         bsr     chkarg          *コマンドラインの解析
 21:
 22:         bsr     do              *メイン処理
 23:
 24:         DOS     _EXIT           *終了
 25:
 26: *
```

```
27: *       メイン処理
28: *
29: do:
30:         bsr     err_redirect    *標準エラー出力を
31:                                 *  リダイレクト
32:         move.l  a2,-(sp)        *チャイルドプロセス起動
33:         bsr     child           *
34:         addq.l  #4,sp           *
35:
36:         move.l  d0,-(sp)        *EXECのエラーコードを退避
37:
38:         move.w  #STDERR,-(sp)   *標準エラー出力をクローズ
39:         DOS     _CLOSE          *（割り当てはconに戻る）
40:         addq.l  #2,sp           *
41:
42:         move.l  (sp)+,d0        *d0.l=EXECの終了コード
43:         bmi     error2          *負ならエラー
44:
45:         rts
46:
47: *
48: *       標準エラー出力をfilnamにリダイレクトする
49: *
50: err_redirect:
51: wopen:
52:         move.w  #ARCHIVE,-(sp)  *指定されたファイルを
53:         pea.l   filnam          *  新規作成する
54:         DOS     _CREATE         *
55:         addq.l  #6,sp           *
56:         tst.l   d0              *エラー？
57:         bpl     wopen0          *  エラーがなければオープン完了
58:
59:         move.w  #WOPEN,-(sp)    *createでエラーが発生したときは
60:         pea.l   filnam          *  openを使って
61:         DOS     _OPEN           *  もう一度ライトオープンしてみる
62:         addq.l  #6,sp           *
63:         tst.l   d0              *エラー？
64:         bmi     error1          *  そうなら今度こそエラー終了
65:
66: wopen0: move.w  d0,d1           *d1.w=出力先ファイルハンドル
67:
68:         move.w  #STDERR,-(sp)   *オープンしたファイルハンドルを
69:         move.w  d1,-(sp)        *  標準エラー出力に
70:         DOS     _DUP2           *  強制コピー
71:         addq.l  #4,sp           *
72:         tst.l   d0              *エラー？
73:         bmi     error1          *  そうならエラー終了
74:
75:         move.w  d1,-(sp)        *いまオープンしたファイルハンドルは
76:         DOS     _CLOSE          *  もういらないから
77:         addq.l  #2,sp           *  クローズしてしまう
78:
79:         rts
80:
81: *
82: *       コマンドラインの解析
83: *
```

```
 84: chkarg:
 85:         addq.l  #1,a2           *a2=コマンドライン文字列先頭
 86:
 87:         bsr     nextarg         *スペースをスキップする
 88:         tst.b   (a2)            *コマンドライン引数があるか？
 89:         beq     usage           *  ないなら引数が足らない
 90:         lea.l   filnam,a0       *a0=ファイル名切り出し領域
 91:         bsr     getarg          *引数１つをa0以降に取り出す
 92:
 93:         lea.l   -100(sp),sp     *100バイトのローカルエリアに
 94:         move.l  sp,-(sp)        *  DOSコールを使って
 95:         move.l  a0,-(sp)        *  ファイル名を
 96:         DOS     _NAMECK         *  展開してみる
 97:         lea.l   100+8(sp),sp    *
 98:         tst.l   d0              *d0が０でなければ
 99:         bne     usage           *  ファイル名の指定に誤りがある
100:
101:         bsr     nextarg         *さらにスペースを飛ばす
102:         tst.b   (a2)            *まだあるか？
103:         beq     usage           *  実行すべきコマンドがない
104:
105:         rts
106:
107: *
108: *       スペースを飛ばしつぎの引数先頭までポインタを進める
109: *
110: nextarg:
111:         bsr     skipsp          *スペースをスキップ
112:
113:         cmpi.b  #'/',(a2)       *引数の先頭が
114:         beq     usage           *  /,-であれば
115:         cmpi.b  #'-',(a2)       *  使用法を表示
116:         beq     usage           *
117:
118:         rts
119:
120: *
121: *       a2の指す位置から引数１つ分を
122: *       a0の指す領域へコピーする
123: *
124: getarg:
125:         move.l  a0,-(sp)        * {レジスタ待避
126: gtarg0: tst.b   (a2)            *1)文字列の終端コードか
127:         beq     gtarg1          *
128:         cmpi.b  #SPACE,(a2)     *2)スペースか
129:         beq     gtarg1          *
130:         cmpi.b  #TAB,(a2)       *3)タブか
131:         beq     gtarg1          *
132:         cmpi.b  #'-',(a2)       *4)ハイフンか
133:         beq     gtarg1          *
134:         cmpi.b  #'/',(a2)       *5)スラッシュ
135:         beq     gtarg1          *
136:         move.b  (a2)+,(a0)+     *  が現れるまで転送を
137:         bra     gtarg0          *  繰り返す
138: gtarg1: clr.b   (a0)            *文字列終端コードを書き込む
139:         movea.l (sp)+,a0        *} レジスタ復帰
140:         rts
```

```
141:
142: *
143: *       コマンドライン先頭のスペースをスキップする
144: *
145: skpsp0: addq.l   #1,a2             *ポインタを進め
146:                                    *繰り返す
147: skipsp:                           *サブルーチンはここから始まる
148:         cmpi.b   #SPACE,(a2)       *スペースか？
149:         beq      skpsp0            *  そうなら飛ばす
150:         cmpi.b   #TAB,(a2)         *TABか？
151:         beq      skpsp0            *  そうなら飛ばす
152:         rts
153:
154:         DOS      _EXIT
155:
156: *
157: *       使用法の表示＆終了・エラー終了
158: *
159: usage:
160:         lea      usgmes,a0
161:         bra      err0
162: *
163: error1:
164:         lea      errms1,a0         *ファイルオープン時エラー
165:         bra      err0
166:
167: error2:
168:         lea      errms2,a0         *チャイルドプロセス生成時エラー
169:         bra      err0
170:         *
171: err0:   move.w   #STDERR,-(sp)     *標準エラー出力へ
172:         move.l   a0,-(sp)          *  メッセージを
173:         DOS      _FPUTS            *  出力する
174:         addq.l   #6,sp             *
175:
176:         move.w   #1,-(sp)          *終了コード１を持って
177:         DOS      _EXIT2            *  エラー終了
178:
179: *
180: *       データ
181: *
182:         .data
183:         .even
184: *
185: usgmes: .dc.b    '機　能：標準エラー出力をファイルに切り替えてから',CR,LF
186:         .dc.b    TAB,'指定のコマンドを実行します',CR,LF
187:         .dc.b    '使用法：RDERR　切り替え先ファイル　実行コマンド'
188: crlfms: .dc.b    CR,LF,0
189: errms1: .dc.b    'ファイルが作成できませんでした',0
190: errms2: .dc.b    'コマンドが起動できませんでした',0
191:
192: *
193: *       ワークエリア
194: *
195:         .bss
196:         .even
197: *
```

```
198: wfno:    .ds.w   1               *リダイレクト先ファイルハンドル
199: filnam:  .ds.b   256             *ファイル名切り出し用バッファ
200: *
201:          .stack
202:          .even
203: *
204: mystack:                         *スタック領域
205:          .ds.l   1024            *
206: mysp:
207:
208:          .end    ent             *実行開始アドレスはent
```

A>RDERR リダイレクト先ファイル コマンド

のようにして使用する。また,

A>RDERR /?

により，上記程度のかんたんな使用法を標準エラー出力に出力して終了する。このあたりのコマンドライン引数関係の処理は，前章のARG2.Sから流用し，微調整してある。

リダイレクトの処理にはDOSコールdup2を使っている。以前使ったdupは，すでにオープンしたファイルハンドルをコピーして新しいファイルハンドルを返すものだったが，dup2はコピー先のファイルハンドル番号を指定できる。つまり，ファイルハンドルを強制的に(上書きする形で)コピーするわけだ。ファイルハンドルAとBがあるときに，dup2でAをBに強制コピーすると，AとBは同じファイルを指すようになる。A=あるファイルをオープンしてできたファイルハンドル，B=標準エラー出力のファイルハンドルと考えれば，これは正しく標準エラー出力のリダイレクションである。心配なのは，このファイルハンドルが子プロセスに引き継がれるかどうかだが，標準入出力やエラー出力に使われる0～4のファイルハンドルはちゃんと引き継がれることになっている。

プログラム中では50行以下がリダイレクトの処理だ。まず，すでにコマンドライン解析ルーチンによってfilnam以下に切り出されているリダイレクト先ファイルを，createで新規作成する。以前のUPPER.Xを参考に，createでエラーが出ても，もう一度openで書き込みモードでオープンしている。これはconなどのキャラクタデバイスが指定された場合に備えてのことだった。

create，openともにエラーであれば，エラーメッセージを出して終了する。オープンできたら，そのファイルハンドルを標準エラー出力を表すファイルハンドルにdup2でコピーし，オープンしたファイルハンドルは不用になったのでクローズする。そして，メインルーチンに戻る。

メインルーチンではその後，childを呼び出して子プロセスを起動し，子プロセスから戻ったら，リダイレクトされていた標準エラー出力をクローズして元のconに戻す。childの戻り値(=execの戻り値)を調べてエラーの有無を判別する前に標準エラー出力を閉じているのは，エラーメッセージを標準エラー出力に出している関係だ。順序を間違えると，エラーメッセージがリダイレクト先のファイルに行ってしまい，エラーメッセージを標準エラー出力に出している意味がなくなる。

CHAPTER A

ファイル管理の方法

CHAPTER A

ファイル管理の方法

ASSEMBLER

この章ではHuman68kのファイル管理システムと絡めて，これまでの話から漏れたファイル，ディスク関連のDOSコールを紹介する。用語などに関しては，いつものようにコラムを参照してください。

COLUMN

ディスク関連用語

ここで，一般的なディスク関係の用語についてかんたんに説明しておく。

●サイド(side)

ディスクの面。表と裏があるわけで，サイド0，サイド1がある。

●トラック(track)

ディスクの面は同心円状に区切られていて(どこかでそんな図を見たことがあるだろう)，その一周分をトラックと呼ぶ。2HDのディスクの場合，片面あたり77トラックあり，外側から順にトラック0，1，…，75，76と数える。

●シリンダ(cylinder)

ディスクアクセスの際には，ディスクの半径上を読み書きヘッドが目的のトラック位置まで移動する。ヘッドはサイドごとに1つあるが，すべて連動して動く。で，ヘッドを移動せずにアクセスできる一周分をシリンダと呼ぶ。フロッピーディスクの場合は，サイド0とサイド1の同一番号のトラックをまとめて1シリンダとする。

●セクタ(sector)

トラックを扇型に区切ったもので，〝ディスクを読み書きする際の(物理的な)最小単位〟だ。それぞれのセクタ先頭には目印がつけられていて，ディスクアクセス時の指標となる。ディスクを(物理)フォーマットするという作業は，この目印を書き込むことだ。同じ2HDでも1トラックを何セクタに分割するかによって複数のフォーマットが存在するが，Human68kでは1セクタ＝1024バイト，8セクタ/トラックが採用されている。セクタは0からではなく，1から数える。

●(論理)セクタ

Human68kでは，〝DOSがディスクを読み書きする際の最小単位〟をやはりセクタと呼んでいる(レコードと呼ぶのが一般的だと思う)。本稿でもこれを踏襲するが，物理的なセクタと区別する必要があるときには〝論理セクタ〟という言葉で表現する。

Human68kの1つの論理セクタは1024バイトであり，0から数えはじめる。フロッピーディスクの場合はセクタの大きさが論理セクタと同じなので，セクタと論理セクタは1

対1で対応し，

サイド0・トラック0・セクタ1……0
サイド0・トラック0・セクタ2……1
サイド0・トラック0・セクタ3……2
：
サイド0・トラック0・セクタ8……7
サイド1・トラック0・セクタ1……8
サイド1・トラック0・セクタ2……9
：
サイド1・トラック0・セクタ8……15
サイド0・トラック1・セクタ1……16
：

のようにセクタ番号がふられている。また，ハードディスクの場合は，通常1セクタ＝256バイトのところを4セクタ＝1論理セクタとすることで，見かけ上，フロッピーディスクと同様に扱えるようになっている。

●クラスタ(cluster)

論理セクタをいくつか集めたもので，〝ファイルを管理するうえでの最小単位〟だ。どんなに小さなファイルを作っても(0バイトのファイルでも)，1クラスタ分のディスクスペースが消費される。Human68kの場合，1クラスタを何セクタとするかはデバイスドライバの作り方による。もっとも，Human68k Ver.1.0では1クラスタ＝1論理セクタ＝1024バイト固定であり，Ver.2.0で可変になったいまも，それ以外のものを見たことはまだない。

OSのファイル管理方法

Human68kは，FAT(File Allocation Table)とディレクトリによってファイルを管理している。このファイルシステムはMS-DOSからそっくりそのまま拝借したものだ[1]。FATとディレクトリのうち，とくにルートディレクトリはディスクの特定の位置に記録されている。また，サブディレクトリは必要に応じてディスク上のデータ領域に作成される。サブディレクトリは，ルートディレクトリと同じような構造をした〝ファイル(のようなもの)〟と考えてもよいだろう。固定領域にあるルートディレクトリだけではなく，この〝ファイル〟もディレクトリと見なして扱ってしまおうというのが階層ディレクトリの本質だ。

■ 1) このお陰で，2HDディスクを介せばHuman68k⟷MS-DOS間でファイルのやりとりができる。

FATは，各要素が1クラスタに対応した配列状のデータだ[2]。FATの要素の値が0ならそのクラスタは未使用，0以外なら使用中を表す。かりにFATの各要素をFAT[n]のように表すと

すると，第10クラスタが使用中かどうかは，FAT[10]が0か0以外かで判断できる。ディスクの使用状況を表すのが，FATの第1の役割だ。

■ 2) FATの実際の構造はここでは示さないが，興味のある人は『プログラマーズマニュアル』の付録を参照してもらいたい。

FATは，それと同時に複数クラスタにまたがるファイルのつながりぐあいをも表す。たとえば，第10クラスタから始まるファイルがあり，FATが，

FAT[10]＝11
FAT[11]＝12
FAT[12]＝14
FAT[13]＝0
FAT[14]＝－1

のようになっていたとする。この場合，ファイルは第10クラスタから第11クラスタ，さらに第12クラスタへと続き，そこから第14クラスタへ飛んでそこで終わる(便宜上－1で最後を示した)ことを表している。このように，ファイルが飛び飛びのクラスタに格納されていても，FATを追うことでつながり方がわかるようにできている。

ディレクトリは1つのファイルにつき32バイトが使われ，その内容は表1のようになっている。ファイル名やファイル長など，dirコマンドで表示される情報がひととおり揃っているのがわかるはずだ。ファイルをオープンしたりする際には，Human68kはこのディレクトリを検索し，該当ファイルを探し出す。

+00H	8 B	ファイル名
+08H	3 B	拡張子
+0BH	1 B	ファイル属性
+0CH	10 B	ファイル名2
+16H	1 W	ファイル変更最終時刻
+18H	1 W	ファイル変更最終年月日
+1AH	1 W	先頭クラスタ番号
+1CH	1 L	ファイル長

表1
ディレクトリ(ファイル1個あたり計32バイト)

COLUMN

ディスクマップ

ディスクにはファイルの中身だけでなく，各ファイルの個別情報や，ディスク全体の使用状況など，DOSがディスク・ファイルを管理するのに必要な情報があわせて記録されている。これらはふつうディスクの特定の場所を占める。

図にHuman68kの2HDディスク，ハードディスク(領域を20M確保した場合にHuman68

kの管理下に置かれる部分のみ),RAMディスク(RAMDISK.SYSで512Kバイト確保した場合)のディスクマップを示す。個々の領域の位置や大きさは異なるが,基本的にはどれも同じ構造をしている。

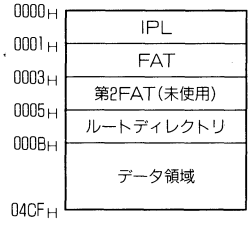
2HDディスク,ハードディスク先頭のIPL(Initial Program Loader)は,要するに〝HUMAN.SYSを読み込むプログラム〟だ。これはFORMAT.Xでディスクをフォーマットする際に(データディスクを作るときでも無条件に)書き込まれる。X68000の電源を入れたときには,最初にROMのIPLプログラムが走り,そのIPLによってディスク中のIPLが読み込まれ,そのIPLがHUMAN.SYSを読み込む。HUMAN.SYSは,さらにデバイスドライバをCONFIG.SYSにしたがって組み込み,続いてCOMMAND.Xなり,VS.Xなりが起動されることになる。

続くFATは,ディスクのどの部分を使用しているかをクラスタ単位で表している。同時に,複数クラスタにまたがるファイルのつながりぐあいもFATで示される。MS-DOSではなんらかの事故でFATが破壊されたときに備えてつねにFATを2組作成するのだが,Human68kではその痕跡が残っているだけで,実際には第2FATは使われていない。

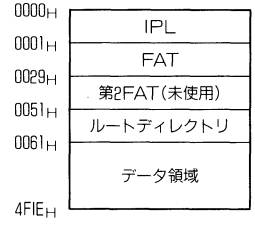
FATの後ろにはルートディレクトリ領域があり,ルートディレクトリに存在するファイルの情報が格納されている。1ファイルにつき32バイトが使われるので,1セクタに32個,2HDディスクでは最大192個,ハードディスクの場合は512個のファイル情報が格納できる。RAMディスクの場合は96個入りそうに見えるが,なぜか92個までということになっている。

そのさらに後ろが実際にファイルの中身を格納するデータ領域だ。FATで管理する都合上,クラスタ単位に分割されている。基本的にはファイルを作るにつれて,先頭から順に使用される。なお,Human68kでは,わけあって第0クラスタ,第1クラスタは存在せず,データ領域の先頭は第2クラスタになる。

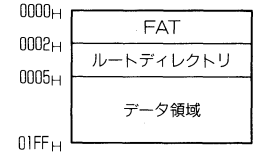
●2HDディスク



●ハードディスク(領域20Mバイト)



●RAMディスク(領域512Kバイト)



ディレクトリの構造は?

では,ディレクトリ上の個々の情報について順に説明する。図1に実際のディレクトリ領域をDB.Xで読み込みダンプしたものを示しておくから,あわせて見てもらいたい。なお,図1はHUMAN.SYSとCOMMAND.Xだけが存在するルートディレクトリの頭の部分だ。

```
+00  4855 4D41 4E20 2020 5359 5324 0000 0000          HUMAN   SYS$....
+10  0000 0000 0000 0060 8412 0200 9ED3 0000          .......  ...榠..
+20  434F 4D4D 414E 4420 5820 2020 0000 0000          COMMAND X   ....
+30  0000 0000 0000 5C64 8412 3700 766D 0000          ......¥d .7. vm..
+40  0000 0000 0000 0000 0000 0000 0000 0000          ................
```

図1
ディレクトリ領域の例

●ファイル名

ファイル名から拡張子を除いた部分の先頭8バイトが格納されている。ファイル名が8バイトに満たない場合は，残りはスペースのコード20_Hで埋められる。先頭の1バイトはときに特別な意味を持つ。先頭が00_Hであればディレクトリの終わりを意味し，以降の領域が使われていないことを表す。ファイル検索時に00_Hから始まるディレクトリ要素を見つけたら，それ以上の検索を行わなくてもよい(＝ファイルは見つからなかった)ことがわかるわけだ。

先頭が$E5_H$であれば消去されたファイルを意味し，ファイル検索時には$E5_H$から始まるディレクトリ要素はたんにスキップされる。また，新規にファイルを作成するときにディレクトリ中に$E5_H$から始まる部分があれば，その領域が新ファイルのために使われることになる。なお，Human68k(や日本語MS-DOS)では漢字のファイル名を許しており，このままでは$E5xx_H$の漢字コードで表される文字から始まるファイル名と消去されたファイルの区別がつかないので，ファイル名先頭の$E5_H$はディレクトリ上では便宜上05_Hで表すというつじつまあわせが行われる。

●拡張子

ファイルの拡張子が格納される。3バイトに満たない部分はやはりスペースで埋められる。

●ファイル属性

ファイルであるのか，サブディレクトリであるのか，また，書き込み禁止かどうかなどの属性を表す。図2に示すように1ビットごとに意味を持ち，該当するビットが1ならその属性を備えていることを表す。DOSコールcreateで指定するのと同じ形式だ。特殊なところではボリュ

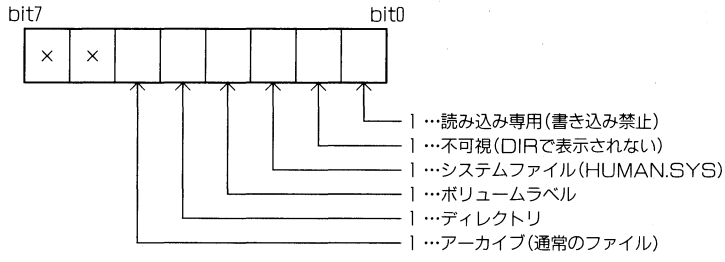


図2
ファイル属性

ーム名かどうかという属性がある。ボリューム名はディスク上の決まった領域に記録されるのではなく，〝ボリューム名の属性を持った実体のないファイル〟としてルートディレクトリ領域内に格納される。

●ファイル名2

ファイル名の後半部が格納されている。この領域はMS-DOSでは未使用になっており，Human68kで拡張された部分だ。

●変更最終時刻

ファイルを作成した(もしくは最後に更新した)時分秒が格納される。省スペース化のために，

HHHHHMMMMMMSSSSS

のように，時5ビット，分6ビット，秒5ビットのビットフィールドになっている[3]。ただ，あまり精度の要求されない秒だけは，本来6ビット必要なところを2で割って5ビットに押し込み，全体で16ビットに収まるようにしてある。なお，MS-DOSとの絡みで，80系プロセッサの癖を引きずっているので，実際には上位バイトと下位バイトが逆転して格納されている。

■ 3) 0～23の数なら5ビットで，0～59であれば6ビットで十分表現できる。

●変更最終年月日

変更最終時刻同様，

YYYYYYYMMMMDDDDD

のように年7ビット，月4ビット，日5ビットのビットフィールドになっている。年は西暦だが，実際の年から1980を引いた数で表現される。やはりディスク上では，上位バイト，下位バイトは逆転して格納されている。

●先頭クラスタ

ファイル本体が記録されている，先頭のクラスタ番号が格納されている。これまたバイト順は逆になっている。ボリューム名の場合は実体がないので，ここは0になる。

●ファイル長

ファイルの長さがバイト数で格納されている。これは，Human68kで扱えるファイルの理論上の最大長が4G－1バイト[4]であることを意味している。バイト順は，最下位バイトから最上位バイトの順で完全にひっくりかえった形で格納される。サブディレクトリやボリューム名の場合はファイル長は0になる。

■ 4) コンピュータ界では，1G＝1024M，1M＝1024K，1K＝1024である。

ディレクトリ操作用DOSコール

ファイルの読み書きを行う場合には,自動的にFATやディレクトリ領域が参照されたり書き換えられたりするわけだが, DOSコールの中には直接これらを参照・変更するためのものがある。ここでは, 比較的使い道がありそうなものをいくつか紹介する。

chmod:ファイル属性を変更する

```
move.w    属性, −(sp)
move.l    ファイル名, −(sp)
DOS       _CHMOD
addq.l    #6, sp
```

属性はディレクトリ内のフォーマットと同様の形で指定する。ただし,−1を指定すると変更するかわりに, 属性の読み込みを行う。その場合, 結果はd0.lで返される(意味を持つのは下位バイトのみ)。

filedate:ファイル最終更新日時を変更する

```
move.l    日時, −(sp)
move.w    ファイルハンドル, −(sp)
DOS       _FILEDATE
addq.l    #6, sp
```

chmodと異なり,ファイル名ではなく,ファイルハンドルでファイルを指定する。このファイルハンドルは, 書き込みモードか読み書き両用モードでオープンしたものでなければならない。

日時は上位ワードで年月日を, 下位ワードで時分秒をディレクトリの内部形式で指定する。日時に0を指定した場合は, 変更のかわりに日時の読み込みを行い, d0.lに返す。このとき, d0.lの上位ワードが$FFFF_H$であればエラーが発生したことを表す。Human68kのDOSコールは, リターン値の正負でエラーの有無を判別することが多いわけだが, filedateは例外的なケースといえる。

rename:ファイル名を変更する

```
move.l    新ファイル名, −(sp)
move.l    旧ファイル名, −(sp)
DOS       _RENAME
```

```
addq.l      #8, sp
```

このDOSコールは2つの機能をあわせ持っている。1つはディレクトリ領域に格納されたファイル名を書き換えることによってファイルのリネームを行う機能であり，もう1つはファイルを，あるディレクトリから他のディレクトリへ移動する機能だ。新ファイル名のパスと旧ファイル名のパスが同じであればリネームになり，異なれば移動となる。

ファイルの移動はディレクトリ上の操作であり，ファイル本体を動かすわけではない。該当ファイルのディレクトリ要素を移動先ディレクトリにコピーし，元のほうを消去することで，実質的なファイルの移動が行える。このため，renameでは異なるドライブ間の移動はできない。

delete:ファイルを消去する

```
move.l      ファイル名, −(sp)
DOS         _DELETE
addq.l      #4, sp
```

ファイルの消去といってもファイルの本体を消してしまうわけではなく，ディレクトリとFATの操作だけが行われる。ディレクトリの先頭は$E5_H$で置き換えられ(他の31バイトはいじらない)，同時にFATの該当部分が未使用に戻される。理屈では，消去した直後であれば，ディレクトリの先頭部分を$E5_H$以外に書き換え，FATを元に戻すことでファイルが復活することになる。しかし，先頭クラスタだけはディレクトリ中に残っているものの，どの空きクラスタをつなぎあわせればファイルが復活するのかという情報はすでに失われているわけであり，よほど単純なケースでないかぎり，FATを100%元に戻すことは困難だ。

files:ファイルを検索する

```
move.w      検索対象ファイル属性, −(sp)
move.l      ファイル名へのポインタ, −(sp)
move.l      ファイル情報格納アドレス, −(sp)
DOS         _FILES
lea.l       10(sp), sp
```

filesは，ディレクトリ領域中からファイルを検索して，そのファイルの情報を次ページの表2に示すような形式で指定バッファに返す。ファイル名にはワイルドカードが使用可能であり，その場合は最初に見つけたファイルに関する情報を返してくる。さらに連続して検索したい場合は次に述べるnfilesを使う。

ファイル属性はchmodやcreate同様，ディレクトリの内部形式で指定する。通常のファイルのみを検索したければ0020_H，サブディレクトリのみを検索したければ0010_Hで指定することに

+00H	21B	Human68kがファイル検索に使用する内部情報
+15H	1B	ファイル属性
+16H	1W	ファイル変更最終時刻
+18H	1W	ファイル変更最終年月日
+1AH	1L	ファイル長
+1CH	23B	ファイル名+'.'+拡張子+00H

表2
filesが返すファイル情報(計53バイト)

なる。複数の属性を同時に指定することもできるが，その場合は〝すべての条件を備えたファイル〟ではなく，〝どれか1つの条件を満たしたファイル〟を探す。もし，〝書き込み禁止属性のついた通常ファイル〟を検索したければ，属性に0021Hを指定したうえで，filesから返されるファイル属性を調べて，見つかったファイルが本当に両方の属性を持っているかどうかをチェックする必要がある。

filesは，検索ファイルが見つからなかった場合やファイル名に異常があったときには負の数のDOSエラーコードを，何かファイルを見つければ正の値をd0.lに返す。あらかじめnameckでファイル名の正当性を調べておけば，filesの戻り値が正か負かでファイルが見つかったかどうかを判断できる。

nfiles:次のファイルを検索する

```
move.l      ファイル情報格納アドレス,－(sp)
DOS         _NFILES
addq.l      #4,sp
```

ワイルドカードを指定してfilesを呼び出した後で，2番目以降のファイルを検索するのに使う。filesを呼び出したときに使ったファイル情報格納領域のアドレスをそのまま渡す。この領域にはさきほどfilesで検索したときの情報が残っているから，DOSはその情報から次はどこから検索を始めたらよいのかを知り，filesで指定したファイル名とマッチする次のファイルを探してfiles同様のデータ形式で返す。ファイルが見つからなければ，d0.lは負の値をとる。これにより，もう検索対象のファイルが存在しないことがわかる。1度filesを実行後，d0.lが負になるまでnfilesを繰り返し呼び出せば，順に該当ファイルの情報が返ってくる[5]。

■ 5) この順序はディレクトリ領域上での順序にしたがう。

dskfre:ディスクの残り容量を得る

```
move.l      結果格納領域アドレス,－(sp)
move.w      ドライブ,－(sp)
```

+00H	1W	空きクラスタ
+04H	1W	総クラスタ
+08H	1W	1クラスタあたりのセクタ数
+0CH	1W	1セクタあたりのバイト数

表3
dskfreが返す情報

```
DOS     _DSKFRE
addq.l  #6, sp
```

このDOSコールは，FATを調べて未使用クラスタの数を数えるものと考えられる。結果を格納する領域は8バイト用意する。dskfreの実行によって，この領域には表3のような情報が返される。使用可能クラスタ数×1クラスタあたりのセクタ数×1セクタのバイト数によってディスクの空き容量が求められる。もっとも，計算済みの結果がちゃんとd0.lに返ってくる。ドライブはAドライブなら1，Bなら2というふうに指定するが，とくに0の場合はカレントドライブが対象になる。

ファイルを作成する際にあらかじめファイルサイズが何バイトになるのかわかっているのであれば，dskfreで早めにチェックしておくことで，えんえんとディスクが回ったあげくエラーになるような事態を回避できるだろう。

余談ながら，ハードディスクのようにクラスタ数が多い(＝FAT領域が大きい)場合には，dskfreの実行にかなり時間がかかる。DIRコマンドでは内部的にこのDOSコールを発行するために，ハードディスクのディレクトリ表示(を始めるまで)が非常に遅くなっている。CONFIG.SYSのBUFFER＝～の行を極端に大きくすることでこの時間を短縮することはできるが，僕はハードディスクを購入したその日にCOMMAND.Xの一部を書き換えて，dskfreを行わないようにしてしまった。するべきことはかんたんで，X-BASICなりなんなりでCOMMAND.Xを読み込み，FF36H(DOS　_DSKFRE)を探して，7000H(moveq.l #0, d0)に書き換えてしまうだけだ。この弊害として，DIRコマンドではディスクの残り容量の表示が意味をなさなくなる(0と表示される)が，試してみたい人はどうぞ。

簡易DIRコマンド作成

DOSコールを紹介しただけで終わってはこの本のタイトルが泣くから，ここで，とくに問題になりそうなfiles，nfilesのサンプルプログラムを示す(リスト1)。このプログラムFILELIST.Xは，指定されたファイルを検索し(もちろん，ワイルドカード可)，たんに該当ファイル名をフルパスで表示する。ファイル名しか表示しないDIRコマンドのようなものだが，サブディレクトリ，ボリューム名は対象からはずしてある。なお，毎度のことながら，コマン

ドライン関係のサブルーチンの大部分は以前作った第７章のプログラムから流用している。

リスト1
FILELIST.S

```
 1: *       DOSコールfiles,nfilesのサンプル
 2:
 3:         .include        doscall.mac
 4:         .include        const.h
 5: *
 6:         .text
 7:         .even
 8: *
 9: ent:
10:         lea.l   mysp,sp         *spの初期化
11:
12:         bsr     chkarg          *コマンドラインの解析
13:
14:         bsr     do              *メイン処理
15:
16:         DOS     _EXIT           *正常終了
17:
18: *
19: *       メイン処理
20: *
21: do:
22:         bsr     chkname         *ファイル名に対する前処理
23:
24:         move.w  #ARCHIVE,-(sp)  *最初のファイルを検索する
25:         pea.l   arg             *
26:         pea.l   filbuf          *
27:         DOS     _FILES          *
28:         lea.l   10(sp),sp       *
29:
30: loop:   tst.l   d0              *ファイルは見つかったか？
31:         bmi     done            *  見つからなければ処理完了
32:
33:         bsr     setpath         *得られたファイル名を
34:                                 *  フルパスに再構成する
35:
36:         bsr     doit            *ファイル１個分を処理する
37:
38:         pea.l   filbuf          *つぎのファイルを検索する
39:         DOS     _NFILES         *
40:         addq.l  #4,sp           *
41:
42:         bra     loop            *繰り返す
43:
44: done:   rts
45:
46: *
47: *       ファイル１個分を処理する（ファイル名を表示するだけ）
48: *
49: doit:
50:         pea.l   arg             *setpathで構成された
51:         DOS     _PRINT          *  フルパスのファイル名を
52:         addq.l  #4,sp           *  表示する
53:
```

```
 54:          pea.l   crlfms          *改行する
 55:          DOS     _PRINT          *
 56:          addq.l  #4,sp           *
 57:
 58:          rts
 59:
 60: *
 61: *        files実行に先立ってファイル名に前処理を加える
 62: *
 63: chkname:
 64:          pea.l   nambuf          *ファイル名を展開する
 65:          pea.l   arg             *
 66:          DOS     _NAMECK         *
 67:          addq.l  #8,sp           *
 68:
 69:          tst.l   d0              *d0＜０なら
 70:          bmi     usage           *  ファイル名の指定に誤りがある
 71:
 72:          beq     nowild          *d0＝０ならワイルドカード指定なし
 73:
 74:          cmpi.w  #$00ff,d0       *d0≠FFHなら
 75:          bne     wild            *  ワイルドカード指定あり
 76:
 77: noname:                   *ファイル名が指定されていない場合
 78:          lea.l   arg,a0          *バッファargに
 79:          lea.l   nambuf,a1       *  nameckで展開したパス名+'*.*'
 80:          bsr     strcpy          *  を再構成する
 81:          lea.l   kome0,a1        *
 82:          bsr     strcpy          *
 83:
 84: wild:                    *ワイルドカードが指定された場合
 85:                         *何もしなくてよい
 86: cknam0: rts
 87:
 88: nowild:                  *ワイルドカードが指定されていない場合
 89:          move.w  #SUBDIR,-(sp)   *サブディレクトリであると仮定して
 90:          pea.l   arg             *  検索してみる
 91:          pea.l   filbuf          *
 92:          DOS     _FILES          *
 93:          lea.l   10(sp),sp       *
 94:
 95:          tst.l   d0              *見つかったか？
 96:          bmi     cknam0          *  見つからなければファイルだろう
 97:
 98:          lea.l   arg,a0          *バッファargに
 99:          lea.l   komekome,a1     *  もとのファイル名+'¥*.*'
100:          bsr     strcat          *  を再構成する
101:
102:          bra     chkname         *nameckでファイル名を展開するために
103:                                  *  サブルーチン先頭に戻る
104:
105: *
106: *        files,nfilesで見付けたファイル名をフルパスに構成し直し
107: *                arg以降に格納する
108: *
109: setpath:
110:          lea.l   arg,a0          *a0=コピー先
```

```
111:            lea.l   nambuf,a1       *a1=nameckで展開したパス名
112:            bsr     strcpy          *コピーする
113:            lea.l   filbuf+30,a1    *a1=files,nfilesで見付けたファイル名
114:            bsr     strcpy          *連結する
115:            rts
116:
117: *
118: *          コマンドラインの解析
119: *
120: chkarg:
121:            addq.l  #1,a2           *a2=コマンドライン文字列先頭
122:            bsr     skipsp          *スペースをスキップする
123: *          tst.b   (a2)            *引数があるか？
124: *          beq     usage           *  ないなら引数が足らない
125:                                    *好みによってこの２行を復活させよう
126:
127:            cmpi.b  #'/',(a2)       *引数の先頭が
128:            beq     usage           *  '/'か
129:            cmpi.b  #'-',(a2)       *  '-'であれば
130:            beq     usage           *  きっとヘルプが見たいのだろう
131:
132:            lea.l   arg,a0          *a0=引数切り出し領域
133:            bsr     getarg          *引数１つをa0以降に取り出す
134:
135:            bsr     skipsp          *さらにスペースをスキップ
136:            tst.b   (a2)            *引数があるか？
137:            bne     usage           *  あるなら引数が多い
138:
139:            rts
140:
141: *
142: *          a2の指す位置から引数１つ分をa0の指す領域へコピーする
143: *
144: getarg:
145:            move.l  a0,-(sp)        * {レジスタ待避
146: gtarg0:    tst.b   (a2)            *1)文字列の終端コードか
147:            beq     gtarg1          *
148:            cmpi.b  #SPACE,(a2)     *2)スペースか
149:            beq     gtarg1          *
150:            cmpi.b  #TAB,(a2)       *3)タブか
151:            beq     gtarg1          *
152:            cmpi.b  #'-',(a2)       *4)ハイフンか
153:            beq     gtarg1          *
154:            cmpi.b  #'/',(a2)       *5)スラッシュ
155:            beq     gtarg1          *
156:            move.b  (a2)+,(a0)+     *  が現れるまで転送を
157:            bra     gtarg0          *  繰り返す
158: gtarg1:    clr.b   (a0)            *文字列終端コードを書き込む
159:            movea.l (sp)+,a0        *} レジスタ復帰
160:            rts
161:
162: *
163: *          コマンドライン先頭のスペースをスキップする
164: *
165: skpsp0:    addq.l  #1,a2           *ポインタを進め
166:                                    *繰り返す
167: skipsp:                            *サブルーチンはここから始まる
```

```
168:            cmpi.b  #SPACE,(a2)     *スペースか？
169:            beq     skpsp0          *  そうなら飛ばす
170:            cmpi.b  #TAB,(a2)       *TABか？
171:            beq     skpsp0          *  そうなら飛ばす
172:            rts
173:
174: *
175: *          文字列の連結および複写
176: *          リターン時a0は文字列末の00Hを指す
177: *
178: strcat:
179:            tst.b   (a0)+           *(a0)は0か？
180:            bne     strcat          *そうでなければ繰り返す
181:            subq.l  #1,a0           *行きすぎたから１つ戻る
182: strcpy:
183:            move.b  (a1)+,(a0)+     *１文字ずつ
184:            bne     strcpy          *終了コードまでを転送する
185:            subq.l  #1,a0           *a0は進み過ぎている
186:                                    *a0は文字列末の00Hを指す
187:            rts
188:
189: *
190: *          使用法の表示＆終了
191: *
192: usage:
193:            move.w  #STDERR,-(sp)   *標準エラー出力へ
194:            pea.l   usgmes          *  ヘルプメッセージを
195:            DOS     _FPUTS          *  出力する
196:            addq.l  #6,sp           *スタック補正
197:
198:            move.w  #1,-(sp)        *終了コード１を持って
199:            DOS     _EXIT2          *  エラー終了
200:
201: *
202: *          メッセージデータ
203: *
204:            .data
205:            .even
206: *
207: usgmes:   .dc.b   '機　能：指定ファイル名をフルパスで表示します',CR,LF
208:            .dc.b   TAB,'ファイル名にはワイルドカードが使用できます',CR,LF
209:            .dc.b   '使用法：FILELIST [ファイル名]'
210: crlfms:   .dc.b   CR,LF,0
211:
212: komekome:
213:            .dc.b   '¥'
214: kome0:    .dc.b   '*.*',0
215:
216: *
217: *          ワークエリア
218: *
219:            .bss
220:            .even
221: *
222: arg:      .ds.b   256             *引数切り出し用バッファ
223:                                *filesで使うバッファは偶数アドレスに置く
224: filbuf:   .ds.b   53              *ファイル情報格納用バッファ
```

```
225:                            *nameckで使うバッファは奇数アドレスでもよい
226: nambuf: .ds.b   91                *ファイル名展開用バッファ
227: *
228:         .stack
229:         .even
230: *
231: mystack:
232:         .ds.l   256               *スタック領域
233: mysp:
234:         .end
```

引数取り込み処理を行った後，21行にきたときにはラベルarg以下にコマンドラインで指定された検索対象ファイル名が入っている。以下，ファイル名の指定に柔軟性を持たせるための前処理(22行)をしてからfilesで1個目のファイルを探し(24～28行)，見つかったらファイル名をフルパスに構成しなおして(33行)から表示する(36行)。こうして1個目のファイルの処理がすんだら，ファイルが見つからなくなるまでnfilesで検索しては表示するという処理を繰り返す(30～42行)。24～42行はfiles，nfilesの典型的な使い方になっている。

実質十数行のメイン処理に対して，22行で呼び出している前処理ルーチンchknameは多少複雑な構成をしている。そのわりにはたいしたことをしているわけではなく，〝A:¥BIN〟や〝A:¥BIN¥〟のようなディレクトリ名のみの指定を〝A:¥BIN¥*.*〟に変換するにすぎない[6]。

■ 6) この処理の必要性は，
A>DIR A:¥BIN
と，
A>ATTRIB A:¥BIN
の結果を見比べてみるとわかると思う。ちなみに，DIRはchkname相当の処理を行っているが，ATTRIB, Xは行っていない。

サブルーチンchknameは，まずDOSコールnameckを実行し，その戻り値であるd0.lを調べることで，どのような形式でファイル名が指定されたのかを知り，処理を振り分ける。nameckの結果起こりうるケースとしては，

1) d0.l＜0：エラー(ファイル名の指定がおかしい)
2) d0.l=FF_H)：ファイル名が指定されていない
   ex)A:¥BIN¥
      A:..
      A:
3) d0.l=0：ワイルドカード指定なし
   ex)A:¥BIN¥ED.X
      A:¥BIN
4) それ以外：ワイルドカード指定あり
   ex)A:¥BIN¥????.X
      A:*.*

があるから，それぞれ個別に対応する。

かんたんなのは，1)のエラーと4)のワイルドカードが使われている場合だ。エラーはすぐにはじけばよいし，ワイルドカードが指定された場合はなにもする必要がない。また，ファイル名の指定がなかった2)の場合も楽で，nameckで展開したファイル情報を寄せ集めてフルパスのファイル名の形にし，末尾に〝*.*〟を付け足せばよい。リスト1では78～82行がこれにあたる。

最後にnameckの戻り値が0の場合，つまりファイル名が存在し，かつ，ワイルドカードが使用されていない場合が残った。一見，このままなにもしないでもかまわないように見えるが，じつはこのケースがいちばん複雑なのだ。上の例を見てもらえばわかるように，nameckの結果だけではふつうのファイルが指定されたのか，サブディレクトリ名が指定されたのかがわからない。そこで少々技を使う。

まず，nameckで展開する前の形のままでfilesにかける(89～93行)。ただし，通常のファイルを検索するのではなく，サブディレクトリのみを検索対象とする。その結果，一致するサブディレクトリが存在すれば，「ああ，ディレクトリだったんだな」ということがわかるので，末尾に〝¥*.*〟を補う(98～100行)。その後ふたたびchknameの先頭に飛んでいるのは，後でnameckで展開した結果が必要になるためだ。102行以下にnameckの呼び出しを入れてもかまわなかったのだが，chknameの先頭に飛べば，nameck実行後75行のチェックに引っかかることがわかっているので，こういう手抜きをしてみた。

また，一致するディレクトリがなければ，「たぶんファイルだろう」というわけで，何もせずに戻る。

ファイル操作応用編

さて，リスト1のFILELIST.Xは，メイン処理ルーチンdoit(と使用法のメッセージ)を差し替えるだけで，かんたんに他のプログラムに変身する。ファイル名だけでなく，ファイルサイズや更新日時も表示するようにすれば，好みのフォーマットのDIRコマンドができるし，ファイル名を表示するかわりにオープンして内容を表示すれば，TYPEにもDUMPにもなる。最後に例をいくつか挙げておこう。

●ファイル属性を表示する(リスト2：ATR.X)

リスト2
ATR.S

```
 1: *       ファイル属性を表示する

46: *
47: *       ファイル1個分を処理する
48: *
49: doit:
50:         bsr     prtatr          *ファイル属性を表示する
51:
```

```
 52:          pea.l   filbuf+30       *ファイル名を表示する
 53:          DOS     _PRINT          *
 54:          addq.l  #4,sp           *
 55:
 56:          pea.l   crlfms          *改行する
 57:          DOS     _PRINT          *
 58:          addq.l  #4,sp           *
 59:
 60:          rts
 61:
 62: *
 63: *        ファイル属性を表示する
 64: *
 65: prtatr:
 66:          lea.l   atrtbl,a0       *a0=ファイル属性表示用データ
 67:
 68:          move.b  filbuf+21,d1    *d1.b=00ADVSHR
 69:          lsl.b   #2,d1           *d1.b=ADVSHR00
 70:
 71:          moveq.l #6-1,d2         *属性は６種類
 72:
 73: pratr0: moveq.l #0,d0           *
 74:          move.b  (a0)+,d0        *d0=表示用ファイル属性（文字）
 75:          move.w  d0,-(sp)        *それをスタックに積んでおく
 76:
 77:          lsl.b   #1,d1           *ファイル属性を１ビットシフト
 78:                                  *  1) d1.b=DVSHR000, C=A
 79:                                  *  2) d1.b=VSHR0000, C=D
 80:                                  *       :
 81:          bcs     pratr1          *その属性がセットされていれば
 82:                                  *  既にスタックに積んである属性を
 83:                                  *  そのまま表示する
 84:          move.w  #'-',(sp)       *そうでなければ
 85:                                  *  スタックトップを'-'に置き換える
 86: pratr1: DOS     _PUTCHAR        *属性１つを表示する
 87:          addq.l  #2,sp           *
 88:
 89:          dbra    d2,pratr0       .*繰り返す
 90:
 91:          move.w  #TAB,-(sp)      *タブをひとつ出力する
 92:          DOS     _PUTCHAR        *
 93:          addq.l  #2,sp           *
 94:
 95:          rts
 96:
 97: *
 98: *        files実行に先立ってファイル名に前処理を加える
 99: *
100: chkname:～

238: *
239: *        メッセージデータ
240: *
241:          .data
242:          .even
243: *
244: usgmes:  .dc.b    '機　能：ファイル属性を表示します',CR,LF
```

```
245:          .dc.b   TAB,'ファイル名にはワイルドカードが使用できます',CR,LF
246:          .dc.b   '使用法：ATR [ファイル名]'
247: crlfms:  .dc.b   CR,LF,0
248:
249: komekome:
250:          .dc.b   '¥'
251: kome0:   .dc.b   '*.*',0
252:
253: atrtbl:  .dc.b   'ADVSHR'          *属性表示用
254:
255: *
256: *        ワークエリア
257: *
258:          .bss
259:          .even
260: *
261: arg:     .ds.b   256              *引数切り出し用バッファ
262:                                 *filesで使うバッファは偶数アドレスに置く
263: filbuf:  .ds.b   53               *ファイル情報格納用バッファ
264:                                 *nameckで使うバッファは奇数アドレスでもよい
265: nambuf:  .ds.b   91               *ファイル名展開用バッファ
266: *
267:          .stack
268:          .even
269: *
270: mystack:
271:          .ds.l   256              *スタック領域
272: mysp:
273:          .end
```

ATTRIB.Xと同じ形式で，ファイル属性とファイル名を表示する。リスト１に対する追加・変更部分だけを示してある。あくまでサンプルではあるが，ファイル名の指定に融通がきかないATTRIB.Xよりも便利といえば便利だろう。好みによっては，検索対象を通常ファイルだけでなく，サブディレクトリなどにも広げてもよい。その場合はリスト１の24行で指定しているファイル属性を適当に変更する。$003F_H$にすれば，すべての属性が検索・表示の対象になる。

COLUMN

ccr

ccr(Condition Code Register)は各種フラグをひとまとめにしたものだ(図)。ccr中で意味を持つのは５ビットだけで，残る３ビットはccr全体を〝コンピュータにとってきりのよい大きさ〟である８ビットに揃えるためだけに存在する。

C(Carry)，Z(Zero)，N(Negative)は条件分岐でおなじみだろう。V(oVerflow)は以前divu命令のところで少し出てきたように，演算のオーバーフローの有無を表す。divuの場合は，商か余りが16ビットで収まらなかった場合にセットされたが，加減算の場合もやはりVビットでオーバーフローを表す。VビットもCやZ同様に条件分岐に使うことができ，V＝１のとき分岐したければbvs，V＝０のとき分岐したければbvcを使う。

ここで，Cビットで表される繰り上がりや桁借りと，Vビットで表されるオーバーフローとの意味の違いに注意したい。Cビットは演算の結果がオペレーションサイズに収まらなかったことを表すが，Vビットは符号付き演算時の算術的な意味でのオーバーフローを表

す。たとえば，d0.b＝7F$_H$(＝127)のとき，

```
addq.b    #1, d0
```

を実行したとする。結果のd0.bは当然80$_H$であり，Cは0だ。この演算が無符号で行われたのなら，127＋1＝128という正しい結果が得られたことになるが，符号付き演算と見なせば，127＋1＝－128という意味のない結果となる。このことを表すために，CとはべつにVビットが用意されている。

残るX(eXtend)ビットは，その他のビットとは少し性格が異なる。演算の結果を表すというよりも，演算結果からはみ出した1ビット分のデータを保持するものと考えたほうがよいだろう。加減算時は繰り上がりや桁借りが生じたときに1になる。これはCビットの変化とまったく同じだが，Xビットは繰り上がりが生じたことを表すためにセットされるのではなく，〝繰り上がった桁自体〟を意味しているのだと考えてもらいたい(2進数の世界では結局どちらも同じことではあるが)。Xビットはその性格上，条件分岐には使えず，おもに多倍長演算時に特別な命令とともに用いられる。

```
addx      B, A
```

はBをAに足し，さらにXビットを加える。この命令を利用すると，

```
add.l     d2, d0
addx.l    d3, d1
```

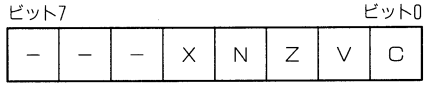
によって，d1d0，d3d2のようにレジスタ2つを連結した形で表される64ビット数どうしを加えることができる。下位ロングワードどうしの加算の繰り上がり分を上位ロングワードにaddxで足し込むわけだ。また，

```
subx      B, A
```

はBをAから引き，さらにXビットを引く。addx同様に，

```
sub.l     d2, d0
subx.l    d3, d1
```

によって64ビット数どうしの減算を行ったりするのに利用される。



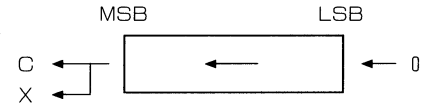
COLUMN

シフト・ローテート命令

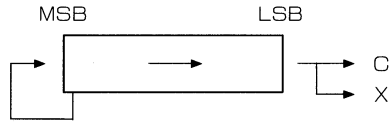
シフト(shift)命令，ローテート(rotate)命令は，データレジスタなり，メモリなりに格納されたデータを数値としてではなく，たんなるビット列として扱う命令群だ。べつに知らなくてもプログラムが書けないというわけではないが，知っていればなかなか便利な場面も多い。シフトはビット列を左右にずらす動作で，ローテートはビット列の両端がつながっているものとして回転させる動作だ。言葉で説明してもわからないだろうから，以下の説明は図を見ながら読んでもらおう。なお，図中，MSB(Most Significant Bit：直訳すれば〝もっとも重要なビット〟)とLSB(Least Significant Bit：同〝もっとも重要でないビット〟)は，それぞれ最上位ビット，最下位ビットの意味で使われる略語だ。

●asl(Arithmetic Shift Left)，asr(……Right)

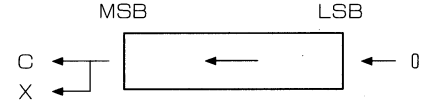
●ASL



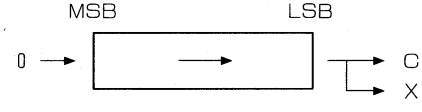
●ASR



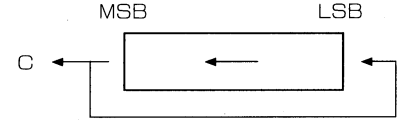
●LSL



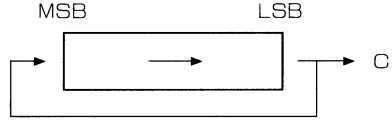
●LSR



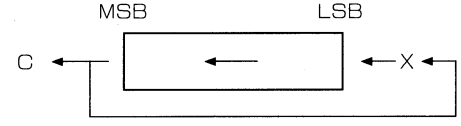
●ROL



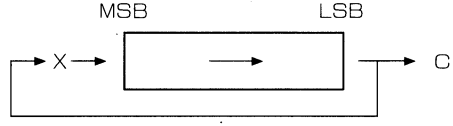
●ROR



●ROXL



●ROXR



ビット列を〝数値としても考慮しつつ〟シフトするので「算術シフト命令」と呼ばれる。しかし，〝算術的〟の意味合いはシフト方向によってちょっと違う。asrの場合は，符号ビットである最上位ビットを変化させないという意味で算術的(シフトしても負の数は負の数，正の数は正の数であり続ける)であり，aslの場合は〝符号ビットが変化したらオーバーフローを表すためにVビットをセットする〟という意味で算術的だ(だまされたような気もするが)。

10進数では桁を1桁左にずらし，末尾に0を加えると10倍したことになるように，2進数は桁を1桁左にシフトし末尾に0を付け足すと2倍したことになる。これはaslが数を2のn乗倍するのに利用できることを意味する。逆に，asrは2のn乗で割る(ただし，端数は切り捨て)のに使える。

●lsl(Logical Shift Left)，lsr(……Right)

論理シフト命令と呼ばれる。ビット列を単純に左右にシフトする命令だ。lslとaslはシフト動作自体はまったく同じで，Vビットの変化のしかただけが異なる。lslはビット列を〝数値とは考えない〟のでオーバーフローはありえず，Vビットはつねにリセットされる。

●rol(ROtate Left)，ror(…Right)

シフトし，押し出されたビットがCフラグに入るのと同時に反対側から入ってくるという動作をする。シフト命令と異なり，Xビットが変化しない点に注意。

●roxl(ROtate with eXtend Left)，roxr(…Right)

Xビットも含めてローテートする命令で，Xビットをレジスタに取り込むときなどに利用する。

以上8つの命令では，68000にとっては不本意なことに使えるアドレッシングモードなどにかなりの制限がある。許される形式は次の3つだけだ。

●asl　#即値，データレジスタ

例）asl.b　#3，d0

指定されたデータレジスタを即値で示されるビット数だけシフトする。即値の範囲は1～8でなければならない。

●asl　データレジスタ1，データレジスタ2

例）asl.l　d1，d0

データレジスタ2をデータレジスタ1で指定されるビット数だけシフトする。データレジスタ1は下位6ビットのみが有効であり，結局0～63までになる。とはいっても，0の場合はシフトしないし，33以上も意味がない。

●asl.w　実効アドレス

例）asl.w　(a0)

　　asl.w　mem

指定アドレスに格納されたワードデータを1ビットシフトする。サイズはワードに固定されている。

●書き込み禁止属性を反転する(リスト3：REVATR.X)

リスト3
REVATR.S

```
 1: *       ファイルの読み込み専用属性を反転する

46: *
47: *       ファイル1個分を処理する
48: *
49: doit:
50:         moveq.l #0,d0           *一応上位バイトをクリア
51:         move.b  filbuf+21,d0    *d0.w＝ファイル属性
52:         eor.b   #READONLY,d0    *読み込み専用属性を反転する
53:
54:         move.w  d0,-(sp)        *属性
55:         pea.l   arg             *ファイル名
56:         DOS     _CHMOD          *属性変更
57:                 *本当はここでエラーチェックをすべきだが省略する
58:         DOS     _PRINT          *ついでにファイル名表示
59:         addq.l  #6,sp           *
60:
61:         pea.l   crlfms          *改行する
62:         DOS     _PRINT          *
63:         addq.l  #4,sp           *
64:
65:         rts
66:
67: *
68: *       files実行に先立ってファイル名に前処理を加える
69: *
```

```
 70: chkname:～

208: *
209: *       メッセージデータ
210: *
211:         .data
212:         .even
213: *
214: usgmes: .dc.b   '機　能：ファイルの読み込み専用属性を反転します',CR,LF
215:         .dc.b   TAB,'ファイル名にはワイルドカードが使用できます',CR,LF
216:         .dc.b   '使用法：REVATR [ファイル名]'
217: crlfms: .dc.b   CR,LF,0
218:
219: komekome:
220:         .dc.b   '¥'
221: kome0:  .dc.b   '*.*',0
222:
223: *
224: *       ワークエリア
225: *
226:         .bss
227:         .even
228: *
229: arg:    .ds.b   256             *引数切り出し用バッファ
230:                                 *filesで使うバッファは偶数アドレスに置く
231: filbuf: .ds.b   53              *ファイル情報格納用バッファ
232:                                 *nameckで使うバッファは奇数アドレスでもよい
233: nambuf: .ds.b   91              *ファイル名展開用バッファ
234: *
235:         .stack
236:         .even
237: *
238: mystack:
239:         .ds.l   256             *スタック領域
240: mysp:
241:         .end
```

役には立ちそうもないが，いちおうchmodの使用例として作ってみた。すでに書き込み属性がセットされていればリセットし，リセットされていればセットする。書き込み禁止属性の反転は，files，nfilesで得られるファイル属性と01_Hの排他的論理和をとることで行っている。

●ファイルの更新時刻を12:00:00に揃える(リスト4：FNOON.X)

リスト4
FNOON.S

```
  1: *       ファイルの最終変更時刻を12:00:00に揃える

 46: *
 47: *       ファイル1個分を処理する
 48: *
 49: doit:
 50:         move.w  #WOPEN,-(sp)    *更新モードで
 51:         pea.l   arg             *  ファイルをオープンする
 52:         DOS     _OPEN           *
 53:         move.w  d0,d1           *d1=ファイルハンドル
```

```
54:         bmi     werror          *d1が負ならエラー
55:                                 *（多分、書き込み禁止）
56:
57:         DOS     _PRINT          *ついでにファイル名を表示
58:         addq.l  #6,sp
59:
60:         pea.l   crlfms          *改行する
61:         DOS     _PRINT          *
62:         addq.l  #4,sp           *
63:
64:         move.w  filbuf+24,d0    *d0の下位ワード＝変更最終年月日
65:         swap.w  d0              *d0の上位ワード＝変更最終年月日
66:
67:         move.w  #$6000,d0       *d0の下位ワード=12:00:00
68:                                 *$6000 = %01100_000000_00000
69:                                 *         HHHHH MMMMMM SSSSS
70:
71:         move.l  d0,-(sp)        *日時
72:         move.w  d1,-(sp)        *ファイルハンドル
73:         DOS     _FILEDATE       *日時変更
74:         DOS     _CLOSE          *すかさずクローズ
75:         addq.l  #6,sp           *
76:
77:         rts
78:
79: *
80: werror:
81:         move.w  #STDERR,-(sp)   *標準エラー出力へ
82:         pea.l   errmes          *  エラーメッセージを
83:         DOS     _FPUTS          *  出力する
84:         addq.l  #6,sp           *
85:
86:         move.w  #1,-(sp)        *終了コード１を持って
87:         DOS     _EXIT2          *  エラー終了
88:
89: *
90: *       files実行に先立ってファイル名に前処理を加える
91: *
92: chkname:～

230: *
231: *       メッセージデータ
232: *
233:         .data
234:         .even
235: *
236: usgmes: .dc.b   '機　能：ファイルの更新時刻を12:00:00に変更します',CR,LF
237:         .dc.b   TAB,'ファイル名にはワイルドカードが使用できます',CR,LF
238:         .dc.b   '使用法：FNOON [ファイル名]'
239: crlfms: .dc.b   CR,LF,0
240:
241: komekome:
242:         .dc.b   '¥'
243: kome0:  .dc.b   '*.*',0
244:
245: errmes: .dc.b   'FNOON：書き込み禁止ファイルです（たぶん）',CR,LF,0
246:
```

```
247: *
248: *       ワークエリア
249: *
250:         .bss
251:         .even
252: *
253: arg:    .ds.b   256             *引数切り出し用バッファ
254:                                 *filesで使うバッファは偶数アドレスに置く
255: filbuf: .ds.b   53              *ファイル情報格納用バッファ
256:                                 *nameckで使うバッファは奇数アドレスでもよい
257: nambuf: .ds.b   91              *ファイル名展開用バッファ
258: *
259:         .stack
260:         .even
261: *
262: mystack:
263:         .ds.l   256             *スタック領域
264: mysp:
265:         .end
```

これはfiledateのサンプルだ。ファイル最終更新日時のうち，日付は変えずに時刻だけを12:00:00に変更する。

この章はこのあたりで切り上げることにする。ゆとりがあるようだったらFILELIST.Xをベースにしたファイル処理プログラムを思いつくままに作ってみるとよいだろう。DIRもどき，TYPEもどき，DUMPもどきも1度は作ろうとしてみてほしい。ひょっとすると完成させることができないかもしれないし，ひどく使いにくいプログラムができ上がるかもしれないが，それなりに得るものもあるだろう。また，ATR.XとREVATR.Xを合体して多少手を加えると，ATTRIB.Xもどきができあがる。うまくできればファイル名指定に柔軟性がある分，オリジナルのATTRIB.Xよりも使いやすいものができるはずだ。

CHAPTER 8

デバイスドライバを作る

CHAPTER 8

デバイスドライバを作る

ASSEMBLER

この章では，Human68kのデバイスドライバを取り上げる。最終的には実用を目指したデバイスドライバプログラムの作成までいくが，その前段階として，まずは軽く周辺を押さえておこう。

デバイスドライバとは?

デバイスドライバ(Device Driver)とは〝装置を駆動するもの〟であり，広義では〝特定のハードを制御する専用プログラム〟全般を指す。いわゆる〝低レベルI/Oルーチン〟と同義だと思って差し支えないが，デバイスドライバという言葉には，単独の〝ルーチン〟ではなく，ある程度のまとまりを持った〝モジュール〟といったニュアンスが込められているようだ。

さまざまな周辺装置を制御しなければならないOSを設計するうえで，各装置の扱いをどうするかは頭の悩ませどころだ。やみくもに制御ルーチンを組み込んでいったのでは将来の拡張が困難になるし，そもそも信頼性の高いシステムを作り上げることができるかどうかも疑わしい。となると，ごく自然に各装置の制御ルーチンをひとまとめのモジュールとして抜き出し，ついでに各モジュールの仕様を統一しておこうという発想が生まれてくる。

〝ついでに〟なんていってしまったが，各デバイスドライバの仕様が統一されているというのは重要なポイントだ。仕様が同じであれば，OSは入出力先のハードが何であれ，ある決まった方法でデバイスドライバからデータを受け取ったり，デバイスドライバへデータを渡したりするだけで入出力が行える。デバイスドライバも入出力装置の一部だと見なせば，論理的には〝まったく同じ手順で入出力が行える装置〟がいくつもあるのと同じことになるわけだ。

また，新しいデバイスをサポートするときにもOS本体に手を入れる必要はなく，デバイスドライバを追加するだけですむというのも大きな利点といえる。こうして見ると，OSにおけるデバイスドライバは，〝周辺装置を操作する下請けルーチン群〟というより，むしろ〝OSと周辺装置の橋渡しをするもの〟と位置づけられるだろう。

ところで，〝デバイスドライバも装置の一部とみなす〟という考え方は，各装置をOSから見えなくして，抽象化することにほかならない。OSとの間でつじつまがあっていれば，デバイスド

ライバ内部ではかなり好き勝手なことがやれるということだ。たとえば，OSから渡されたデータをそのままではなく加工してから装置に出力してもいいし，RAMディスクのようにハードウェアをともなわない仮想的な装置があるふりをしてもいい。なんなら，渡されたデータをどこにも送らずに捨ててしまってもOSにはバレない。アイデアしだいでどうにでもなるだけの自由度があるのだ。

というところで，ぼちぼちHuman68kにおけるデバイスドライバに話を持っていく。

Human68kでは……

ご存じのように，Human68k(やMS-DOS)ではCONFIG.SYSの〝DEVICE=〜〟行に記述するというスマートな方法で，デバイスドライバの組み込みが行えるようにできている。このおかげでユーザーはメーカーその他から提供される(もしくは自作の)新しいデバイスドライバを追加したり，不用なデバイスドライバを外してメモリを節約したりして，自分の環境を自由に構築することができる。

これらCONFIG.SYSで組み込むデバイスドライバ以外にも，Human68kには最低限の入出力を行うための数種のデバイスドライバがあらかじめ組み込まれている。とくにデバイスドライバを組み込んでいないのにフロッピーディスクが使えたり，キーボードからの文字入力が行えるのは，このHuman68k内蔵のデバイスドライバがあるためだ。内蔵のデバイスドライバとしては，以下のものが用意されている。

1)キャラクタデバイス

NUL(何もしないダミーのデバイス)
CON(画面とキーボード)
PRN(プリンタ：漢字コードの変換あり)
LPT(プリンタ：漢字コードの変換なし)
AUX(RS-232Cポート)
CLOCK(カレンダーと時計)

2)ブロックデバイス

2HDフロッピーディスク
ハードディスク

ここで，キャラクタデバイスとは文字単位で順番に(シリアルな)入出力を行うデバイスのことを，ブロックデバイスとはあるまとまり単位で入出力を行い，ランダムアクセス可能なデバイスのことをいう。Human68kでは，キャラクタデバイスとブロックデバイスは完全に区別して扱われ，デバイスドライバの作りも若干異なる。

CONFIG.SYSで組み込むデバイスドライバとHuman68k内蔵のデバイスドライバは同じ構造をしており，いったん組み込んでしまえば両者は区別なく扱われる。キャラクタデバイス

は固有の名前を持っており，場合によっては同名のデバイスが複数組み込まれることもありうるが，そのときには最後に組み込んだものだけが有効となる。これにより，内蔵のデバイスドライバを他のものに置き換えることが可能になっている。現にPRNとLPTは，PRNDRV.SYSとしてより高機能なものが用意されているし，日本語FEPであるASK68Kの実体はCONデバイスドライバであり，内蔵のCONデバイスと置き換わる形で組み込まれる[1]。なお，ブロックデバイスは登録順にA:，B:，……という名前で追加されていくだけで，置き換えはできない。

■ 1) 実際にはCONデバイスは普通のデバイスドライバより多くの機能をサポートしなければならず，組み込み手順や作り方も若干異なる。これについては『プログラマーズマニュアル』に詳し……くはないが載っている。

デバイスドライバの構造

Human68kのデバイスドライバは，デバイスヘッダ，ストラテジルーチン，割り込みルーチンの3つの部分から構成される。とりあえず，デバイスヘッダについてのみ解説しておこう。デバイスヘッダはデバイスドライバの先頭に必ず置かれる表1のような構造をしたデータで，ここにはデバイスドライバの種類などを示す情報が格納されている。

●リンクポインタ

OSがメモリ上でデバイスドライバを管理するのに必要な情報で，続くデバイスドライバの先頭を指している(次がなければ$FFFFFFFF_H$=−1で表される)。Human68k内蔵のデバイスドライバは，NULから始まってNUL→CON→AUX→PRN→LPT→CLOCK→2HDディスク→ハードディスクの順序で鎖状にリンクポインタでつながっており[2]，Human68kは鎖を順にたどって目的のデバイスドライバを探す。CONFIG.SYSで組み込むデバイスドライバはこの後ろにリンクされる[3]。

■ 2)ただし，ハードディスクから起動すると，2HDディスクのドライバの前にハードディスクのドライバがリンクされる。
■ 3)キャラクタデバイスは置き換えが可能であるが，デバイスドライバの登録自体はあくまで追加する形で行われる。

$+00_H$	1L	リンクポインタ
$+04_H$	1W	デバイス属性
$+06_H$	1L	ストラテジルーチン先頭アドレス
$+0A_H$	1L	割り込みルーチン先頭アドレス
$+0E_H$	8B	デバイス名

表1 デバイスヘッダ(計22バイト)の構造

●デバイス属性

デバイスのもろもろの性質・特性を表している。図1に示すようにビット単位で意味を持ち，該当ビットが1であれば，その属性を持っていることを表す。第0～5ビットはキャラクタデバイス特有の属性であり，ブロックデバイス(第15ビットが0)では意味を持たない。

第0ビットを1にしておくと，起動時に(デバイスドライバ組み込み時に)標準入力に割り付けられる。同様に，第1ビットが1なら標準出力(と同時に標準エラー出力)に割り付けられる。通常CONが標準入力，標準出力，標準エラー出力を独占しているのは，CONのデバイス属性の第0，1ビットがともにセットされているためだ。

第2ビットはNULデバイスにかぎってセットされる。ちょっと実験してみたところ，このビットがセットされているデバイスドライバは(どうせ入出力しても無駄だと判断され)，呼び出しそのものが省略されるようだ。どんなキャラクタデバイスも，このビットを立てるだけで機能しなくなってしまう。また，第3ビットはCLOCKデバイスかどうかを表す。CLOCKデバイスは，Human68k内部で日付や時刻の設定・取得に使用される特殊なデバイスで，ユーザーの目にふれることはない。

第5ビットはおもにDOSコールのreadとwriteを使って入出力を行う際に参照される。このビットが1であれば，readやwriteのパラメータで指定されたサイズ分まとめてHuman68k←→デバイスドライバ間でデータのやりとりが行われ(RAWモード)，0であれば1バイトずつデータの受け渡しが行われる(COOKEDモード)。たとえば，writeで1024バイトのデータを出力する際には，RAWモードであれば1回のデバイスドライバ呼び出しで1024バイトがまとめて送り出され，COOKEDモードなら1バイトずつ1024回に分けて出力される。大量のデータを扱うときには，RAWモードであったほうがデバイスドライバ呼び出しにかかる時間が少なくてすむぶんだけ高速に入出力を行うことができる。そのかわり，RAWモードには以下のような癖がある。

1) ^C，^Sの扱い

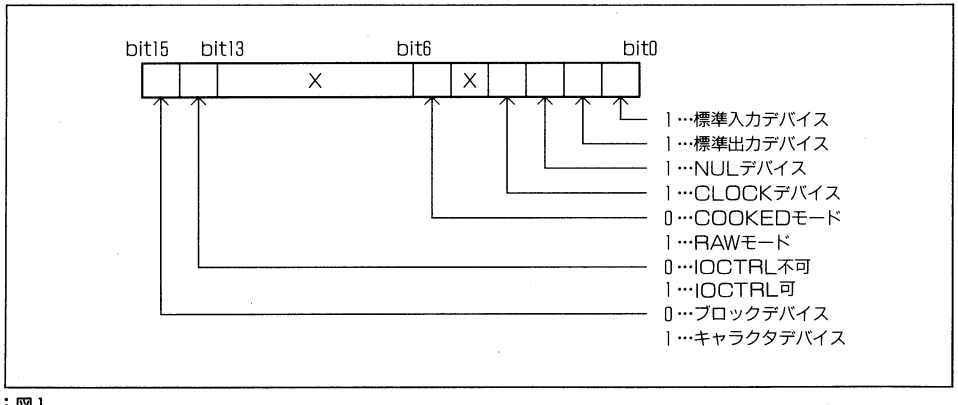


図1
デバイス属性

COOKEDモードであれば，1バイト入出力するごとに(OSに戻るから)ブレイクチェックが効くが，RAWモードでは指定されたバイト数分のデータを処理するまでデバイスドライバから戻ってこないので，^Cで止めることも，^Sで一時停止することもできない。

2)^Zの扱い

デバイスドライバからの入力データ中に^Zがあった場合，COOKEDモードでは^Zをデータの最後と見なしてそれ以上の入力を行わないのに対し，RAWモードの場合は^Z以降もとにかく指定されたバイト数だけのデータを入力する。DOSコールreadを使って入力するときには，指定した入力要求バイト数と戻り値のd0.lを比べ，d0.lのほうが小さければ入力がもうないと判断するのが常套手段である。しかし，RAWモードのデバイスから入力するときには，d0.lと入力要求バイト数はいつまでたっても等しいままで，readの戻り値にだけ頼っているとプログラムがいつまでも止まらないことになる。

こういった事情もあって，たいていのキャラクタデバイスは速度が低下するのを承知でCOOKEDモードにしておく。しかし，〝モード〟という表現が暗示しているように，RAWモードとCOOKEDモードはアプリケーション側からDOSコールを利用して切り替えることが可能になっており，大量のデータを送るときに一時的にRAWモードに切り替えるという技が使える。これについては後で実例を示す。

さて，デバイス属性中，空きになっている第6～13ビットはつねに0にしておき，第14ビットはDOSコール$FF44のioctrlによる入出力機能が使用できるかどうかを表している。1なら使用可能で，0なら不可だ。ioctrlによる入出力は(デバイスに対してではなく)デバイスドライバに対してコマンドを送ったりするためにあり，デバイスドライバのモード切り替えなどに利用する。

最後の第15ビットは，見てのとおり，キャラクタデバイスかブロックデバイスかを表している。1ならキャラクタデバイス，0ならブロックデバイスだ。

●ストラテジルーチン先頭アドレス

●割り込みルーチン先頭アドレス

デバイスドライバの2つのエントリアドレスが並んでいる[4]。Human68kはここを見て，デバイスドライバの処理ルーチンがどこにあるのかを知る。

■ 4) エントリ(entry)は，ルーチンの入口の意味。

●デバイス名

キャラクタデバイスの場合にのみ意味を持つ。8バイトまでのデバイス名を入れ，残りは半角スペースで埋めておく。デバイス名には，通常のファイル名として許される文字のみが使用できる[5]。なお，ブロックデバイスの場合は先頭の1バイトは20_H以下のコードにしておく約束になっている。残りの7バイトは参照されることはないが，適当なデバイス名を入れておいてもよい。

■ 5) 漢字も使えてしまう。

サンプルプログラム

デバイスヘッダ関連のサンプルとして，デバイスドライバの組み込み状況を表示するプログラムDEVICE.Xをリスト1に示す。サンプルとはいっても，デバイスドライバを作ったり，システムの解析を行う際にはなかなか重宝するはずだ。リスト2に用意したデバイスドライバ関連定数定義ファイルDRIVER.Hをインクルードしていることと，以前作った数値→16進文字列変換サブルーチンitohをリンクしている点に注意して実行ファイルを作成してもらいたい。なお，DRIVER.Hは，また後で使うことを考えてDEVICE.S中では使わない定数も数多く定義されている。DEVICE.Xを作るためだけであれば，38行以下はなくてもかまわない。

リスト1
DEVICE.S

```
*       デバイスドライバの組み込み状況を表示する
*
        .include        doscall.mac
        .include        const.h
        .include        driver.h
*
        .xref   itoh
*
PUTC    macro   chr             *1文字出力マクロ
        move.w  chr,-(sp)
        DOS     _PUTCHAR
        addq.l  #2,sp
        endm
*
PUTSP   macro                   *スペース1個出力マクロ
        PUTC    #SPACE
        endm
*
PUTS    macro   strptr          *文字列出力マクロ
        pea.l   strptr
        DOS     _PRINT
        addq.l  #4,sp
        endm
*
NEWLIN  macro                   *改行マクロ
        PUTS    crlfms
        endm
*
SELMES  macro   bit,str1,str2   *属性ビットに応じて
        local   skip,done       *  2種類の文字列の
        btst.l  #bit,d7         *  どちらかを表示するマクロ
        beq     skip
        PUTS    str1
        bra     done
skip:   PUTS    str2
done:
        endm
*
        .text
```

```
40:             .even
41: *
42: ent:
43:             lea.l   mysp,sp         *spの初期化
44:
45:             clr.l   -(sp)           *スーパーバイザモードへ移行
46:             DOS     _SUPER          *
47:             move.l  d0,(sp)         *ssp待避
48:
49:             bsr     chkarg          *コマンドラインの解析
50:
51:             bsr     do              *メイン処理
52:
53:             DOS     _SUPER          *ユーザーモードへ復帰
54:             addq.l  #4,sp           *
55:
56:             DOS     _EXIT           *正常終了
57:
58: *
59: *       メイン処理
60: *
61: do:
62:             bsr     seanul          *デバイスドライバリンクの
63:                                     *  先頭を探す
64:             tst.l   d1              *もし見つけられなければ
65:             bmi     error           *  エラー
66:
67:             PUTS    title           *見出し表示
68:
69: loop:       movea.l d1,a0           *a0=デバイスドライバ先頭
70:             move.w  DEVATR(a0),d7   *d7.w=デバイス属性
71:
72:             bsr     prtadr          *先頭アドレスを表示する
73:             bsr     prtnam          *デバイス名を表示する
74:             bsr     prtatr          *デバイス属性を表示する
75:             SELMES  IOCTRL_BIT,okmes,notmes
76:                                     *IOCTRL可/不可を表示する
77:             NEWLIN                  *改行する
78:
79:             move.l  DEVLNK(a0),d1   *d1=次のデバイスドライバ先頭
80:             cmpi.l  #-1,d1          *d1=-1であれば終了
81:             bne     loop            *そうでなければ繰り返す
82:
83:             rts                     *処理完了
84: *
85: HUMANST  equ     $6800               *Human68k先頭アドレス
86: NULATR   equ     $8024               *NULデバイスのデバイス属性
87: *
88: seanul:
89:             lea.l   nulnam,a0       *a0=検索デバイス名
90:             move.w  (a0)+,d0        *d0='NU'
91:             move.l  (a0)+,d1        *d1='L   '
92:             move.w  (a0)+,d2        *d2='  '
93:
94:             lea.l   HUMANST,a0      *a0=Human先頭アドレス
95:
96: seanl0:      cmp.w   (a0)+,d0        *先頭２文字を比較
```

```
 97:          bne      seanl0              *一致するまで繰り返す
 98:                                       *（デバイスヘッダは必ず
 99:                                       *　偶数番地から始まる）
100:          cmp.l    (a0),d1             *真ん中４文字を比較
101:          bne      seanl0              *一致しなければやり直し
102:
103:          cmp.w    4(a0),d2            *後半２文字を比較
104:          bne      seanl0              *一致しなければやり直し
105:
106:          cmpa.l   nulnam+2,a0         *本当にNULデバイスかどうかの
107:          beq      notfound            *　チェック１
108:
109:          cmp.w    #NULATR,DEVATR-DEVNAM-2(a0)      *チェック２
110:          bne      seanl0                           *
111:
112:          lea.l    DEVLNK-DEVNAM-2(a0),a0  *a0=デバイスドライバ先頭
113:          move.l   a0,d1                   *d1=同上
114:          rts
115: notfound:
116:          moveq.l  #-1,d1              *NULデバイスが見つからなかった！？
117:          rts
118:
119: *
120: *         デバイスドライバの先頭アドレスを表示する
121: *
122: prtadr:
123:          pea.l    temp                *アドレスを16進８桁に変換する
124:          move.l   a0,-(sp)            *
125:          bsr      itoh                *
126:          addq.l   #8,sp               *
127:
128:          PUTS     temp+2              *下６桁のみを表示する
129:
130:          PUTSP                        *スペース出力
131:          rts
132:
133: *
134: *         デバイス名を表示する
135: *
136: prtnam:
137:          lea.l    DEVNAM(a0),a1       *a1=デバイス名先頭
138:          moveq.l  #0,d1               *上位バイトをクリアしておく
139:          moveq.l  #8-1,d2             *デバイス名は８文字
140:
141: prtnm0:  move.b   (a1)+,d1            *１文字取り出し
142:          cmpi.b   #SPACE,d1           *コントロールコードか？
143:          bcc      prtnm1              *そうでなければそのまま表示
144:          moveq.l  #'.',d1             *'.'に変換する
145: prtnm1:  PUTC     d1                  *１文字出力
146:          dbra     d2,prtnm0           *繰り返す
147:          rts
148:
149: *
150: *         デバイス属性を表示する
151: *
152: prtatr:
153:          btst.l   #ISCHRDEV_BIT,d7
```

```
154:            beq     prtat2
155:
156:            PUTS    chrmes          *以下キャラクタデバイス用
157:
158:            SELMES  ISRAW_BIT,rawmes,cokmes
159:
160:            move.w  d7,d1
161:
162:            lea.l   atrdat,a1
163:            moveq.l #0,d2
164:            moveq.l #4-1,d3
165: prtat0:    move.b  (a1)+,d2
166:            lsr.w   #1,d1
167:            bcs     prtat1
168:            moveq.l #'-',d2
169: prtat1:    PUTC    d2
170:            dbra    d3,prtat0
171:            rts
172:
173: prtat2:    PUTS    blkmes          *ブロックデバイスのとき
174:            rts
175:
176: *
177: *       コマンドラインの解析
178: *
179: chkarg:
180:            addq.l  #1,a2           *a2=コマンドライン文字列先頭
181:            bsr     skipsp          *先頭のスペースを飛ばす
182:            tst.b   (a2)
183:            bne     usage
184:            rts
185:
186: *
187: *       コマンドライン先頭のスペースをスキップする
188: *
189: skpsp0:    addq.l  #1,a2           *ポインタを進め
190:                                    *繰り返す
191: skipsp:                           *サブルーチンはここから始まる
192:            cmpi.b  #SPACE,(a2)     *スペースか？
193:            beq     skpsp0          *  そうなら飛ばす
194:            cmpi.b  #TAB,(a2)       *TABか？
195:            beq     skpsp0          *  そうなら飛ばす
196:            rts
197:
198: *
199: *       エラー終了/使用法の表示
200: *
201: usage:
202:            lea.l   usgmes,a0       *使用法メッセージ
203:            bra     errout
204: *
205: error:     lea.l   errmes,a0       *NULドライバがない
206:            *
207: errout:    move.w  #STDERR,-(sp)   *標準エラー出力へ
208:            move.l  a0,-(sp)        *  メッセージを
209:            DOS     _FPUTS          *  出力する
210:            addq.l  #6,sp           *
```

```
211:
212:            move.w  #1,-(sp)        *終了コード１を持って
213:            DOS     _EXIT2          *  エラー終了
214:
215: *
216: *          データ
217: *
218:            .data
219:            .even
220: *
221: *                   12345678
222: nulnam:    .dc.b   'NUL     '      *検索デバイス名
223: *
224: title:     .dc.b   '  開始  デバイス名       属性         IOCTRL',CR,LF
225:            .dc.b   '------ ---------- ---------------- ------',CR,LF,0
226: *                   1234567890123456789O
227: chrmes:    .dc.b   '    CHR ',0
228: blkmes:    .dc.b   '    BLOCK           ',0
229: rawmes:    .dc.b   '(RAW)    ',0
230: cokmes:    .dc.b   '(COOKED) ',0
231: okmes:     .dc.b   '    可',0
232: notmes:    .dc.b   '   不可',0
233: *                   123456789
234: *
235: atrdat:    .dc.b   'IONC'          *属性表示用データ
236: *
237: usgmes:    .dc.b   '機　能：デバイスドライバの組み込み状況を'
238:            .dc.b           '表示します',CR,LF
239:            .dc.b   '使用法：DEVICE',CR,LF,0
240: *
241: errmes:    .dc.b   'DEVICE：ありえないことですが…',CR,LF
242:            .dc.b           'NULデバイスが見つかりません'
243: crlfms:    .dc.b   CR,LF,0
244: *
245:            .bss
246:            .even
247: *
248: temp:      .ds.b   8+1             *16進変換用ワーク
249: *
250:            .stack
251:            .even
252: *
253: mystack:
254:            .ds.l   256             *スタック領域
255: mysp:
256:            .end
```

リスト2
DRIVER.H

```
1: *          デバイスドライバ作成用定数定義
2: *                  driver.h
3: *
4:
5: *
6: *          デバイスヘッダ内オフセット
7: *
```

```
 8:          .offset 0
 9: *
10: DEVLNK: .ds.l   1       *リンクポインタ
11: DEVATR: .ds.w   1       *デバイス属性
12: DEVSTR: .ds.l   1       *ストラテジルーチンエントリ
13: DEVINT: .ds.l   1       *割り込みルーチンエントリ
14: DEVNAM: .ds.b   8       *デバイス名
15: *
16:         .text
17: *
18: *       デバイス属性
19: *                        54321098 76543210
20: CHAR_DEVICE     equ     %10000000_00000000
21: BLOCK_DEVICE    equ     %00000000_00000000
22: ENABLE_IOCTRL   equ     %01000000_00000000
23: DISABLE_IOCTRL  equ     %00000000_00000000
24: RAW_MODE        equ     %00000000_00100000
25: COOKED_MODE     equ     %00000000_00000000
26: CLOCK_DEVICE    equ     %00000000_00001000
27: NUL_DEVICE      equ     %00000000_00000100
28: STDOUT_DEVICE   equ     %00000000_00000010
29: STDIN_DEVICE    equ     %00000000_00000001
30: *
31: ISCHRDEV_BIT    equ     15
32: IOCTRL_BIT      equ     14
33: ISRAW_BIT       equ     5
34: ISCLOCK_BIT     equ     3
35: ISNUL_BIT       equ     2
36: ISSTDOUT_BIT    equ     1
37: ISSTDIN_BIT     equ     0
38:
39: *
40: *       エラーコード
41: *
42: _ABORT          equ     $1000
43: _RETRY          equ     $2000
44: _IGNORE         equ     $4000
45: *
46: ILLEGAL_CMD     equ     $0003.or._ABORT.or._IGNORE
47:
48: *
49: *       リクエストヘッダ内オフセット
50: *
51: CMD_CODE        equ     2
52: ERR_LOW         equ     3
53: ERR_HIGH        equ     4
54: *
55: *init
56: DEV_END_ADR     equ     14
57: PAR_PTR         equ     18
58: *
59: *sns
60: SNS_DATA        equ     13
61: *
62: *i/o
63: DMA_ADR         equ     14
64: DMA_LEN         equ     18
```

プログラムのアルゴリズム自体は単純で，デバイスドライバのリンクをNULデバイスから順にたどりながら，その先頭アドレスとデバイス名，およびデバイス属性を表示しているだけだ。ただ，NULデバイスがどこにあるのかを知るまっとうな方法がないので，かなり汚い方法を使わざるをえなかった[6]。Human68k内部を覗いて，NULデバイスのデバイスヘッダを検索している。具体的には，デバイス名の部分に相当する〝NUL〟＋5文字のスペースという文字列を探し，見つかったらさらにデバイス属性がある〝はず〟の部分が，あらかじめ調べておいたNULデバイスのデバイス属性に一致するかどうかを調べている。両者とも一致すれば，〝かなり高い確率〟で，そこはNULデバイスドライバのデバイスヘッダと考えられるわけだ。100%そうだとはいい切れないのがつらいが，この手法はOPMDRV.Xでも使われており[7]，その意味では〝OSの供給元によって保証された正当な方法〟だ。

■ 6) Human68kのVer.2.0で拡張されたDOSコールを使えば，もっときれいな方法でNULデバイスを探すことができる。
■ 7) OPMDRV.Xはコマンドレベルから起動して組み込むことができるが，その際デバイスドライバのチェインを追って終端を探し，そこに自分自身をつなぐ。キャラクタデバイスの場合は，デバイスドライバのチェインリンクにつなげさえすれば，デバイスとして使えるようになる。

プログラムの細部については解説の必要もないと思うが，デバイス名を表示するときにブロックデバイスに備えて(画面が乱れないように)20_H未満のコントロールコードをピリオドに置き換えて出力している点(142～144行)と，属性表示サブルーチン中のlsr命令の使い方(166行)あたりはいちおうチェックしておいてもらおう。

また，リスト中ではまだろくに説明していないマクロの定義，オフセットの定義といったAS.Xの機能を使っているし，任意のビットが0か1かを調べるbtst命令が初登場している。それぞれコラムを参照してほしい。

COLUMN

マクロ

アセンブラのマクロ機能とは，いくつかの命令をひとまとめにして名前をつける機能だ。AS.Xでは，

```
マクロ名   macro [仮引数名[,……]]
           定義内容
           endm
```

のようにしてマクロの定義を行う。たとえば，

```
PUSH       macro     reglist
           movem.l   reglist, －(sp)
           endm
```

のようにマクロPUSHを定義しておくと，

```
movem.l   d0/a4-a5, －(sp)
```

と書くかわりに，

```
    PUSH    d0/a4-a5
```

と記述できるようになる。定義内容は数行にわたってもよく，リスト1にもあるように，

```
    PUTC    macro   chr
            move.w  chr, -(sp)
            DOS     _PUTCHAR
            addq.l  #2, sp
            endm
```

と定義しておけば，3行にわたるDOSコールの呼び出しも，

```
    PUTC    #'a'
```

とか，

```
    PUTC    d0
```

の1行ですむ。そういえば，日ごろDOSコールの呼び出しに使っていたDOS ～もDOS-CALL.MAC中で定義されたマクロだった。期せずして示せたように，マクロは入れ子にもできる。

マクロはタイプ量を減らし，ときにプログラムの見通しをよくするのにも有効な手段であり，その意味ではサブルーチンと性格が似ていなくもない。が，要は単純な文字列の置き換えにすぎず，サブルーチンの実体はプログラム中ただ1つなのに対して，マクロはアセンブル時にその場その場で展開され，定義同様のコードが埋め込まれる。簡便さから多用しすぎると，プログラムをいたずらに大きくすることにもなりかねない。逆にサブルーチンの呼び出しには若干の時間がかかるが，マクロにすればその呼び出し時間をなくすことができるので，速度的にはマクロにしたほうが有利だ。

アセンブラのマクロ機能は，アセンブリ言語を多少なりとも高級言語仕立てにして，開発の手間を押さえるために生まれた(だろう)もので，汎用のマクロをたくさん作っておけばおくほど開発効率を上げることができる。しかし，多くの場合，汎用に作ろうとすればするほど，マクロ定義自体が冗長になっていく。どこで妥協するかは人それぞれだろう。

ちなみに，僕は冗長さがいっさいない場面を除いてあまりマクロを使わないほうだが，使いはじめるとそれなりに楽なものだからついつい多用してしまう。もっとも，そういう〝楽したい〟という精神状態のときは，すぐに〝もっと楽したい〟と考えるようになり，いきなりC言語を使い出したりする。で，しばらくCでプログラムを書いているうちに，むくむくと最適化の虫が頭をもたげてきて，Cのプログラムを(あくまでCのレベルで)最適化したりなんかしているうちに，ついついインラインアセンブラを使い出す。インラインアセンブラは使い出すとそれなりに楽しいものだからますます多用してしまう。しかし，そういう〝最適化したい〟という精神状態のときは，すぐに〝もっと最適化したい〟と考えるようになり，マシン語に戻ってきては，1ワード，2クロックの単位での最適化に精を出すのだ。

おっと，肝心なことを言い忘れるところだった。リスト1の29行以下で定義しているマクロSELMESを見てもらいたい。30行でlocalという疑似命令を使っている。この命令は〝以下のラベルはマクロ内でのみ有効である〟ことを宣言するのに使う。ここで宣言したラベルはソースの他の部分からは〝見えなく〟なり，同名のラベルがあってもかちあうことがない。マクロ内でラベルを定義するときにはかならずlocalで宣言しておくものだと思っていればまちがいないだろう。

COLUMN

.offset疑似命令

デバイスヘッダのような〝構造を持ったデータ〟を扱うときには，その先頭アドレスをアドレスレジスタに入れておき，ディスプレースメント付きアドレスレジスタ間接アドレッシングでアクセスするのがわかりやすい方法だ。a0がデバイスヘッダの先頭を指しているとすると，デバイス属性は先頭から4バイト目のワードデータだから，

```
move.w    4(a0), d0
```

によってd0.wに取り出すことができ，デバイス名は先頭から14バイト目だから，

```
lea.l     14(a0), a1
```

によって，そのアドレスをa1に入れることができる。さらにわかりやすく表記したければ，各要素のオフセットアドレス(先頭からのアドレスの差)を，

```
DEVATR  equ     4
DEVNAM  equ     14
```

のようにラベル定義しておき，

```
move.w    DEVATR(a0), d0
lea.l     DEVNAM(a0), a1
```

とかやればよい。

ところが，デバイスヘッダのような単純な構造ならまだしも，各要素の大きさがまちまちで数も多い場合，オフセットを手計算で求めるのは面倒だし，計算違いが入り込む恐れも出てくる。ここで登場するのが.offset疑似命令だ。.offsetは，

```
          .offset  0
DATA1:    .ds.l    1
DATA2:    .ds.w    1
DATA3:    .ds.l    1
          .text
```

のように使う。いわば疑似的なセクションを指定する命令であり，後ろに数値をともなうことを除けば，.textや.dataなどの命令と同様の使われ方をする。ただし，.offsetで指定される疑似セクションには.ds命令しか置くことができない。しかも，あくまで疑似的なセクションであり，.offset以下の.dsによって実際にメモリが確保されることはなく，結果としてラベルの値だけが残る。.offsetの後ろに.dsでずらずらとデータ構造を記述しておくだけで，各要素のオフセットアドレスが得られるわけだ。上の例ではラベルDATA1に0を，DATA2に4を，DATA3に6を，equで個別に定義したのと同じ効果を得ている。なお，.offsetの後ろの数値は，この疑似セクションの先頭アドレス(＝最初のデータのオフセットアドレス)となる。上の例を，

```
    .offset  4
```

に変更すると，DATA1～3はそれぞれ4ずつずれて，4，8，10になる。

.offset疑似命令は，データ構造を定義する以外にスタックフレーム上の位置を定義するのにも便利だし，多少姑息だが，

```
          .offset  0
SUN:      .ds.b    1
MON:      .ds.b    1
```

```
TUE:    .ds.b   1
WED:    .ds.b   1
                :
```

のようにPascalやCでいう列挙型定数を定義するのにも使える。この例ではSUNが0，MONが1……と，順に1ずつ増える定数群を定義しているわけだ。

COLUMN

ビット操作命令

ビット単位の操作を行う命令として68000には，

```
btst
bclr
bset
bchg
```

の4命令が用意されている。順にビットのテスト(0か1かを調べる)，テストしてからクリア(0にする)，テストしてからセット(1にする)，テストしてから反転(0なら1，1なら0にする)する命令だ。テスト結果はccrのZビットに反映され，指定ビットが0であればZ＝1になる。セットしたりクリアしたりするときにも，さきにテストが行われるのを忘れないことだ。これらの命令は，たとえば，

```
btst.l      #7,d0
```

のようにして使い，ソースオペランドでビット位置を，デスティネーションオペランドで対象を指定する。この例の場合，d0.lの第7ビットをテストしている。

ビット操作命令は，いかにもマシン語らしい命令であり，使う場面をまちがえなければそれなりに役に立つ。しかし，68000の命令にしては，適用できるアドレッシングモードやサイズに制限が多く，

```
btst.b      #即値，  実効アドレス
btst.l      #即値，  データレジスタ
btst.b      データレジスタ，実効アドレス
btst.l      データレジスタ，データレジスタ
```

の形式しか許されない。ソースオペランドは即値かデータレジスタで，デスティネーションオペランドがデータレジスタのときはサイズはロングワード固定，それ以外はバイト固定だ(サイズがバイトのときはソースオペランドの下位3ビット，ロングのときは下位5ビットのみが有効になる)。

図2にDEVICE.Xの実行結果を示す。Human68k Ver.2.0でメーカー純正のデバイスドライバをひととおり組み込んだ状態だ。20_H以下のコードは〝.〟に置き換えて表示するようにしてあるので，ブロックデバイス名は〝.〟から始まるかのように表示されている。CONやPRN，LPTが2つずつある理由はさきほど話したとおりで，上がHuman68k内蔵，下がCONFIG.SYSで組み込んだものだ。

見慣れない〝FLOAT*/-〟，〝HIST/**/〟というヤツはダミーのデバイス名で，実際に

```
 開始  デバイス名        属性                IOCTRL
-------------------------------------------------------------
00ECE6  NUL          CHR (RAW)       --N-   不可
00ED7E  CON          CHR (COOKED)    IO--   不可
00F91A  AUX          CHR (COOKED)    ----   不可
00F9F4  PRN          CHR (COOKED)    ----   不可
00FA5E  LPT          CHR (COOKED)    ----   不可
00FB14  CLOCK        CHR (COOKED)    ---C   不可
00FE22  .HARDDSK     BLOCK                  不可
00FE22  .DISK2HD     BLOCK                  不可
030DEE  CON          CHR (COOKED)    IO--   不可
0606CE  PRN          CHR (COOKED)    ----   不可
06075E  LPT          CHR (COOKED)    ----   不可
0608D0  PCM          CHR (RAW)       ----    可
06099E  OPM          CHR (RAW)       ----   不可
075CC4  FLOAT*/-     CHR (COOKED)    ----   不可
078C44  HIST/**/     CHR (COOKED)    ----   不可
081024  .RAMDISK     BLOCK                  不可
08126A  .S_RAM       BLOCK                  不可
```

図2

はデバイスドライバとしては機能していない。このことは〝/〟や〝*〟などのファイル名・デバイス名には使えないはずの文字が含まれていることでもわかると思う。FLOATn.XやHISTORY.Xは，本当はデバイスドライバとは別の形でシステムを拡張するように作られたプログラムであり，CONFIG.SYSで組み込めるようにするためだけに体裁を整えてデバイスドライバのふりをしているのだ。

デバイス操作のDOSコール

ここで，デバイスドライバと直接かかわるDOSコールioctrlを紹介する。このDOSコールは複数の機能を持ち，モード番号によって引数の数や意味が異なる。

●モード0:装置情報の取得

```
move.w    ファイルハンドル, −(sp)
clr.w     −(sp)          * モード0
DOS       _IOCTRL
addq.l    #4, sp
```

この機能は，以前フィルタを作ったときに利用したことがある。装置情報は，d0.wに図3のような形式で返される。指定したファイルハンドルがキャラクタデバイスに割り付けられていた場合は(多少形式の違いはあるが)，デバイス属性が得られることになる。

●モード1:装置情報の設定

```
move.w    装置情報, −(sp)
```

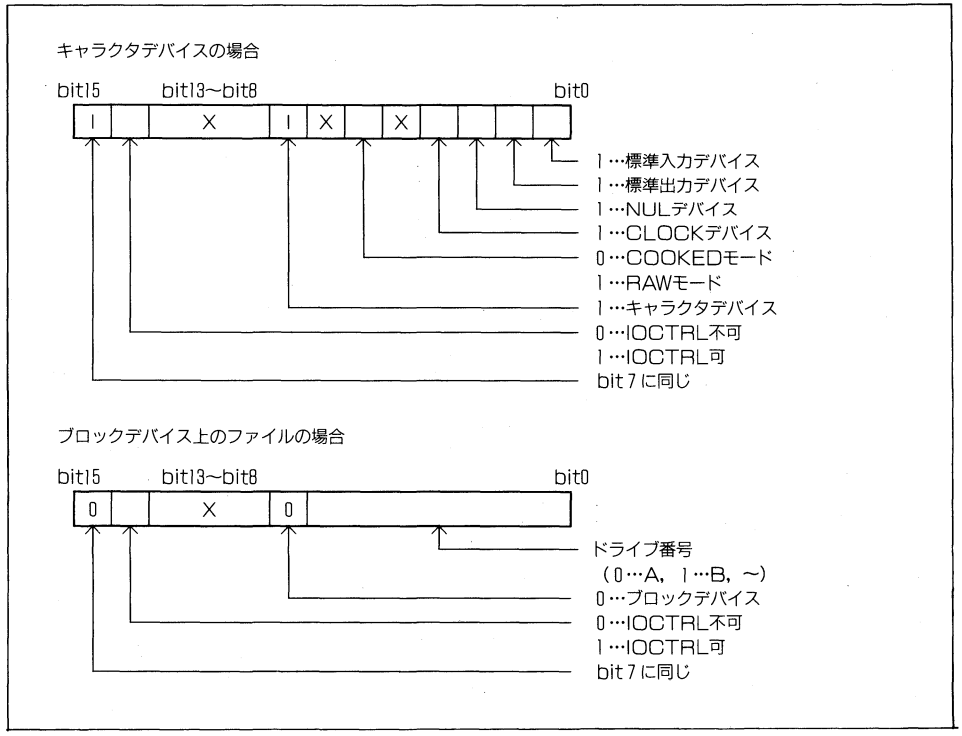


図3
装置属性

```
move.w    ファイルハンドル, −(sp)
move.w    #1, −(sp)        * モード1
DOS       _IOCTRL
addq.l    #6, sp
```

設定する装置情報は第5ビットだけが意味を持ち，また，ファイルハンドルはキャラクタデバイスに割り付けられたものでなければならない。つまり，この機能はRAWモードとCOOKEDモードを切り替えるためのみに存在する。装置情報に0020_Hを指定すればRAWモードになり，0を指定すればCOOKEDモードになる。ここで，モードを切り替えてもデバイスドライバのデバイス属性自体が書き換えられるわけではない点に注意してもらいたい。CONをオープンしRAWモードに切り替えたとしても，後でまたオープンしたときには(本来のデバイス属性のままの)COOKEDモードに戻っている。

なお，エラーが発生した場合は戻り値のd0.lに負のエラーコードが返ってくるが，正常終了した場合は(『プログラマーズマニュアル』の記述と異なり)意味のない正の値が返ってくるようだ。

リスト3のRAWTEST.Xは適当な文字列をCONに，まずはCOOKEDモードで，続いてRAWモードで出力する。実行経過を見ると，表示速度が違うのがわかるだろう。69行以下の出

力データを数倍に増やすともっとはっきりするはずだ。速度の違いが納得できたら，表示の途中で^Sを押してみて，RAWモードとCOOKEDモードでのブレイクチェックの違いも確認しておいてもらいたい。

リスト3
RAWTEST.S

```
*       RAWモードとCOOKEDモードの違いを見る
*
        .include        doscall.mac
        .include        const.h
        .include        driver.h
*
        .text
        .even
*
ent:
        lea.l   mysp,sp         *spの初期化

        move.w  #WOPEN,-(sp)    *CONを書き込みモードで
        pea.l   connam          *  オープンする
        DOS     _OPEN           *
        addq.l  #6,sp           *

        move.w  d0,d1           *d1=出力先

        move.w  #COOKED_MODE,d0 *COOKEDモードにして
        bsr     setmode         *
        bsr     out             *テストデータを出力

        move.w  #RAW_MODE,d0    *RAWモードにして
        bsr     setmode         *
        bsr     out             *テストデータを出力

        move.w  d1,-(sp)        *クローズ
        DOS     _CLOSE          *
        addq.l  #2,sp           *

        DOS     _EXIT

*
*       ファイルハンドルd1.wに
*       装置情報d0.wをセットする
*
setmode:
        move.w  d0,-(sp)        *装置情報
        move.w  d1,-(sp)        *ファイルハンドル
        move.w  #1,-(sp)
        DOS     _IOCTRL
        addq.l  #6,sp
        rts

*
*       ファイルハンドルd1.wに
*       mes以下のデータを出力する
*
```

```
51: meslen  =       mesend-mes
52:
53: out:
54:         move.l  #meslen,-(sp)
55:         pea.l   mes
56:         move.w  d1,-(sp)
57:         DOS     _WRITE
58:         lea.l   10(sp),sp
59:         rts
60:
61: *
62: *       データ
63: *
64:         .data
65:         .even
66: *
67: connam: .dc.b   'CON',0
68: *
69: mes:    .dc.b   '12345678901234567890'
70:         .dc.b   '12345678901234567890'
71: *                       :
72:         .dc.b   '12345678901234567890'
73:         .dc.b   '12345678901234567890'
74:         .dc.b   CR,LF
75: mesend:
76: *
77:         .stack
78:         .even
79: *
80: mystack:
81:         .ds.l   256             *スタック領域
82: mysp:
83:         .end
```

●モード2：デバイスドライバからの直接入力

●モード3：デバイスドライバへの直接出力

```
    move.l    入出力バイト数,－(sp)
    move.l    入出力バッファアドレス,－(sp)
    move.w    ファイルハンドル,－(sp)
    move.w    #2,－(sp)          *モード2
*   move.w    #3,-(sp)           *または3
    DOS       _IOCTRL
    lea.l     12(sp),sp
```

デバイスドライバに対して制御コマンドを送ったりするのに利用する。デバイス属性の第14ビットがセットされているデバイスドライバに対してのみ使用できる。戻り値のd0.lは，エラー時は負のエラーコード，正常終了時は(またまたマニュアルと異なり)0だ。

サンプルとして，Human68k標準のデバイスドライバのうち唯一ioctrlによる入出力をサポートしている(図2参照)，PCMドライバのモードを切り替えるプログラムPCMMODE.Xを

リスト4に示す。パラメータとして/0～/4をとり，数字が小さいほどPCMドライバのサンプリング周波数が低くなる。PCMドライバにはioctrlによって2バイトのデータを送ることでADPCMのサンプリング周波数を切り替える機能がついていたのだ。

```
A>pcmmode /0
```

のようにモードを切り替えてから

```
A>copy beep.sys pcm
```

といったぐあいに適当なPCMデータファイルをPCMデバイスにコピーして，音の高さが違っていることを確認してもらいたい。

リスト4
PCMMODE.S

```
*       IOCTRLによる出力を利用して
*       ADPCMドライバのサンプリング周波数を設定する
*
        .include        doscall.mac
        .include        const.h
*
        .text
        .even
*
ent:
        lea.l   mysp,sp         *spの初期化

        bsr     chksw           *コマンドラインの解析

        bsr     do              *メイン処理

        DOS     _EXIT           *正常終了

*
*       メイン処理
*
do:
        move.w  #RWOPEN,-(sp)   *'PCM'を読み書き両用モードで
        pea.l   pcmnam          *オープンしてみる
        DOS     _OPEN           *
        addq.l  #6,sp           *

        move.w  d0,d1           *d1=ファイルハンドル
        bmi     error1          *負であればオープンできなかった
                                *（PCMドライバが組み込まれていない）

        move.w  d1,-(sp)        *オープンしたファイルハンドルの
        clr.w   -(sp)           *  装置情報を取得する
        DOS     _IOCTRL         *
        addq.l  #4,sp           *

        tst.b   d0              *装置情報の第7ビットが0ならば
        bpl     error1          *  デバイスではない
                                *  （PCMドライバが存在せず、
                                *    偶然同名のファイルがあった）

        add.w   d0,d0           *装置情報の第14ビットが0ならば
```

```
43:             bpl     error2          *  IOCTRLが許可されていない
44:
45:             move.l  #2,-(sp)        *出力データは２バイト
46:             pea.l   mode            *出力データアドレス
47:             move.w  d1,-(sp)        *ファイルハンドル
48:             move.w  #3,-(sp)        *出力モード
49:             DOS     _IOCTRL         *
50:             lea.l   12(sp),sp       *
51:
52:             tst.l   d0              *戻り値が負ならエラー
53:             bmi     error2          *
54:
55:             move.w  d1,-(sp)        *ファイルハンドルをクローズ
56:             DOS     _CLOSE          *
57:             addq.l  #2,sp           *
58:
59:             rts                     *処理完了
60:
61: *
62: *       オプションスイッチの解析
63: *
64: chksw:
65:             tst.b   (a2)+           *コマンドライン引数はあるか？
66:             beq     usage           *  なければエラー
67:                                     *a2=コマンドライン文字列先頭
68: chksw0:     bsr     skipsp          *先頭のスペースを飛ばす
69:             cmpi.b  #'/',(a2)       *先頭が'/'か'-'でなければエラー
70:             beq     chksw1          *
71:             cmpi.b  #'-',(a2)       *
72:             bne     usage           *
73: chksw1:     addq.l  #1,a2           *
74:             move.b  (a2)+,d0        *スイッチを取り出す
75:
76:             subi.b  #'0',d0         *文字→数値変換
77:             bcs     usage           *
78:             cmpi.b  #4+1,d0         *上限のチェック
79:             bcc     usage           *
80:
81:             move.b  d0,mode         *ワークに格納する
82:
83:             bsr     skipsp          *スペースを飛ばす
84:             tst.b   (a2)            *まだ文字列があれば
85:             bne     usage           *  引数が多過ぎる
86:
87:             rts
88:
89: *
90: *       コマンドライン先頭のスペースをスキップする
91: *
92: skpsp0:    addq.l  #1,a2           *ポインタを進め
93:                                     *繰り返す
94: skipsp:                            *サブルーチンはここから始まる
95:             cmpi.b  #SPACE,(a2)     *スペースか？
96:             beq     skpsp0          *  そうなら飛ばす
97:             cmpi.b  #TAB,(a2)       *TABか？
98:             beq     skpsp0          *  そうなら飛ばす
99:             rts
```

```
100:
101: *
102: *       エラー終了/使用法の表示
103: *
104: usage:
105:         lea.l   usgmes,a0       *使用法メッセージ
106:         bra     errout
107: *
108: error1: lea.l   errms1,a0       *PCMドライバがない
109:         bra     errout
110: error2: lea.l   errms2,a0       *IOCTRLを受け付けない
111:         *
112: errout: move.w  #STDERR,-(sp)   *標準エラー出力へ
113:         move.l  a0,-(sp)        *  メッセージを
114:         DOS     _FPUTS          *  出力する
115:         addq.l  #6,sp           *
116:
117:         move.w  #1,-(sp)        *終了コード１を持って
118:         DOS     _EXIT2          *  エラー終了
119:
120: *
121: *       データ
122: *
123:         .data
124:         .even
125: *
126: pcmnam: .dc.b   'PCM',0          *出力先デバイス名
127: mode:   .dc.b   $00,$03          *出力データ
128: *
129: usgmes: .dc.b   '機　能：PCMドライバのサンプリング周波数を'
130:         .dc.b           '設定します',CR,LF
131:         .dc.b   '使用法：PCMMODE [スイッチ]',CR,LF
132:         .dc.b   TAB,'/0 3.9kHz',CR,LF
133:         .dc.b   TAB,'/1 5.2kHz',CR,LF
134:         .dc.b   TAB,'/2 7.8kHz',CR,LF
135:         .dc.b   TAB,'/3 10.4kHz',CR,LF
136:         .dc.b   TAB,'/4 15.6kHz（標準状態）',CR,LF
137:         .dc.b   0
138: *
139: errms1: .dc.b   'PCMMODE：PCMドライバが組み込まれていません',CR,LF,0
140: errms2: .dc.b   'PCMMODE：うまく設定できませんでした',CR,LF,0
141: *
142:         .stack
143:         .even
144: *
145: mystack:
146:         .ds.l   256             *スタック領域
147: mysp:
148:         .end
```

リスト中23～38行は，特定のデバイスが組み込まれているかどうかをチェックする際の決まりきった方法だ。とにかくオープンしてみてから，装置情報を取得してキャラクタデバイスかどうかを確認するわけだ。読み書き両用モードでオープンしているので，もし偶然同名のファイルがあった場合にもすぐクローズすればファイル内容が失われることはない。

参考までに付け加えておくと，出力している2バイトデータはPCMドライバが内部でIOCSコールを呼び出す際のパラメータだ。詳しくは『プログラマーズマニュアル』中IOCSコール$60, $61のADPCMOUT, ADPCMINPを参照してもらいたい。また，PCMドライバからioctrlによって入力すると1バイトのステータスコードが返ってくるが，これはIOCSコール$66のADPCMSNSの戻り値そのものとなっている。

●モード4：ブロックデバイスドライバからの直接入力

●モード5：ブロックデバイスドライバへの直接出力

```
    move.l    入出力バイト数,－(sp)
    move.l    入出力バッファアドレス,－(sp)
    move.w    装置番号,－(sp)
    move.w    #4,－(sp)          *モード4
*   move.w    #5,－(sp)          *または5
    DOS       _IOCTRL
    lea.l     12(sp),sp
```

ファイルハンドルのかわりに，装置番号で直接任意のブロックデバイスドライバを指定する以外はモード2，3と変わらない。装置番号はA:なら1，B:なら2のように指定し，とくにカレントドライブは0で指定することが許される。

●モード6：入力ステータス取得

●モード7：出力ステータス取得

```
    move.w    ファイルハンドル,－(sp)
    move.w    #6,－(sp)          *モード6
*   move.w    #7,－(sp)          *または7
    DOS       _IOCTRL
    addq.l    #4,s
```

指定したファイルハンドルと対応づけられたデバイスドライバが現在入出力可能かどうかを調べる。リターン時のd0.lがFF_Hなら可能で，0なら不可だ。要するにデバイスドライバにおうかがいを立てて，データを送ったり受け取ったりできる状態かどうかを確認する機能である。プリンタのように低速な装置にデータを出力する場合などに利用することが考えられる。しかし，わざわざ調べなくてもデバイスドライバにデータを送りつけさえすれば，後はデバイスドライバがうまくタイミングを見はからってくれるはずなので，この機能を使う場面はほとんどないだろう。

デバイスドライバ呼び出しの手順

次に，Human68kがデバイスドライバをどういった手順で呼び出すかという話に移ろう。

Human68k←→デバイスドライバ間のデータの受け渡しは，リクエストヘッダ(Request header)と呼ばれるメモリブロックを介して行われる[8]。デバイスドライバを呼び出す際に，Human68kは処理内容を格納したリクエストヘッダを作成し，その先頭アドレスをa5レジスタに入れてデバイスドライバに渡す。リクエストヘッダの構造や大きさは処理内容ごとに異なるが，先頭の5バイトは表2のように決まっている。入力を行うのか出力なのかといった処理の種類は0～12のコマンドコード(表3)で表され，リクエストヘッダの2バイト目に格納される。

■ 8) Human68kがなにかにつけ参考にしているMS-DOSでは，Human68kでいうリクエストヘッダのことを，たんに「コマンドパケット」(command packet)と呼び，とくにその先頭の部分のことを「リクエストヘッダ」と呼んでいる。

デバイスドライバ側は，渡されたリクエストヘッダからコマンドコードや付随するパラメータを取り出し，必要な処理を行ってから処理がうまくいったかどうかというステータスをリクエストヘッダに格納して返す。終了ステータスは正常終了時は0000_H，エラーのときは次ページの図4および表4に示す2バイトのエラーコードで，リクエストヘッダの3，4バイト目に下

0	26という定数
1	ブロックデバイスではユニット番号 キャラクタデバイスでは未使用
2	コマンドコード
3	終了ステータス下位バイト
4	終了ステータス上位バイト

表2
リクエストヘッダの
先頭5バイト

コード	キャラクタデバイス	ブロックデバイス
0	初期化	初期化
1		ディスク交換チェック
2		
3	IOCTRLによる入力	IOCTRLによる入力
4	入力	入力
5	先読み入力	ドライブのコントロールおよびセンス
6	入力ステータスチェック	
7	入力バッファクリア	
8	出力（VERIFY OFF時）	出力（VERIFY OFF時）
9	出力（VERIFY ON時）	出力（VERIFY ON時）
10	出力ステータスチェック	
11		
12	IOCTRLによる出力	IOCTRLによる出力

表3
コマンドコード

位バイト，上位バイトの順でセットする[9]。デバイスドライバがエラーを返した場合は，Human68kは即座にエラー処理に入り，画面中央にエラーメッセージを表示し，中止/再実行/無視の選択をユーザーに求める。中止/再実行/無視の選択肢のうちどれを有効とするかは，デバイスドライバの返すエラーコードで指定することができる。

■ 9)バイト順がひっくり返っているのはMS-DOSにあわせたためらしい。こんなことまで真似しなくてもよかったように思うが。

上の説明では，Human68kがリクエストヘッダをデバイスドライバに渡すといっきに処理をすませてしまうような書き方をしたが，現実にはデバイスドライバの呼び出しは以下のように2度に分けて行われる。

1) Human68kはリクエストヘッダを作成し，その先頭アドレスをa5レジスタに入れてデバイスドライバのストラテジエントリをサブルーチンコールする。
2) デバイスドライバのストラテジルーチンでは，渡されたリクエストヘッダへのポインタを内部に待避し，いったんHuman68kに戻る。

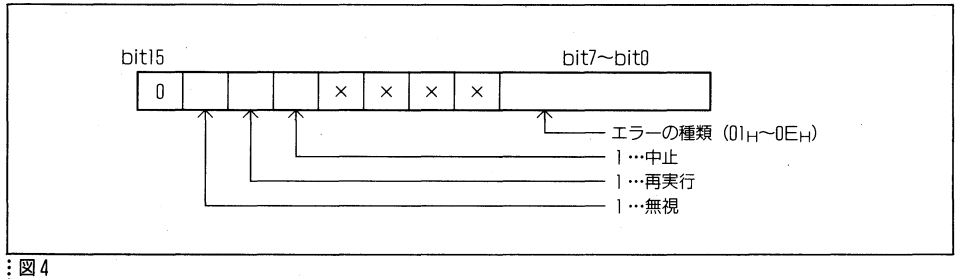


図4
エラーコード

01H	ユニット番号が不正
02H	ドライブの準備ができていない（ディスクが入っていない）
03H	コマンドコードが無効
04H	CRCエラー
05H	ディスクの管理領域が破壊されている
06H	シークエラー
07H	メディアが無効
08H	指定セクタが見つからない
09H	プリンタオフライン
0AH	書き込みエラー
0BH	読み込みエラー
0CH	その他のエラー
0DH	ライトプロテクト（ディスクの取り換え不可）
0EH	ライトプロテクト（ディスクの取り換え可）

表4
エラーの種類

3) Human68kはすかさずデバイスドライバの割り込みルーチンを呼び出す。

4) インタラプトルーチンでは，2)で待避しておいたリクエストヘッダをあらためて取り出し，実際の入出力処理を行う。

このような奇妙な手順になっているのは，将来Human68kがマルチタスク化されるときに対応しやすくするため，とのことだ[10]。マルチタスクのシステムでは，平行して走っている[11]複数の処理単位(呼び方はシステムによってさまざまだが，ここでは「タスク」と呼ぼう)がデバイスドライバの都合を考えずに入出力を要求してくる。当然，デバイスドライバが処理を行っている最中に別のタスクが同一のデバイスドライバに入出力を要求することもある。さすがに，複数の入出力要求を同時に満たすことはできないし，といって要求があるたびにその場その場で入出力処理を行ってしまおうなんて考えるとシステム全体の効率が悲惨なことになる。結局，入出力要求をどこかに溜め込んでおき，そこから1つずつ取り出しては処理するような細工が必要になってくる。その管理をOSに任せることも考えられないではないが，かなり負担が大きいのでここはデバイスドライバ側で対応する。入出力要求を受け取るだけのルーチンと，入出力処理そのものを行うルーチンを分離しておくのだ。前者は入出力要求があるたびに呼び出され，要求された仕事をキュー(queue：待ち行列)に加えて，すぐOSに戻る。後者はタスク管理の一環として定期的に呼び出され，キューから入出力要求を取り出しては実際の入出力処理を行う。

■10) もっとも，マルチタスク化されたときにいままでのデバイスドライバがそのまま使えるというわけではないようだ。

■11) "平行して走っている" とはいっても，CPUが1つしかないかぎり，ある瞬間には1つのプログラムしか動かしようがない。細かく時間を分割し，各タスクを少しずつ実行することで見かけ上平行動作しているように見せるわけだ。

将来のそういう使い方を想定して，いまのうちからHuman68kのデバイスドライバには2つのエントリポイントが設けられているわけだ。ただし，Human68kが本当にマルチタスク化されるかどうかはまだわからない。

サンプルで試してみよう

Human68kのデバイスドライバを構成するデバイスヘッダ，ストラテジルーチン，割り込みルーチンの3つのモジュールのうち，デバイスヘッダに関してはすでに話してある。また，ストラテジルーチンは前述のようにa5で渡されるリクエストヘッダを内部ワークに待避して戻るだけのルーチンであり，

```
move.l    a5, 待避用ワーク
rts
```

の2行で書けてしまう。後はデバイスドライバの実質的な本体である割り込みルーチンが残っ

ているわけだが，さきにこれまでの話を確認する意味で，ごくかんたんなサンプルプログラムを見てもらおう。

リスト5のCTESTDRV.Sは，非常に単純なキャラクタデバイスドライバのサンプルだ。DRIVER.Hをインクルードしていることに注意して，

```
AS CTESTDRV
LK CTESTDRV
REN CTESTDRV.X CTESTDRV.SYS
```

のようにアセンブル・リンクしたうえで〝.SYS〟にリネームするか，そうでなければ，

```
LK CTESTDRV /OCTESTDRV.SYS
```

とリンク時にOスイッチでファイル名を指定してCTESTDRV.SYSを作成する[12]。その後，CONFIG.SYSに，

```
DEVICE = [パス名]CTESTDRV.SYS
```

の1行を追加してからシステムを立ち上げ直せば，CHRTESTというデバイスが使えるようになる[13]。このデバイスCHRTESTは，入力時には無条件にEOFコード($1A_H$)を返し，出力時には出力データをどこにも送らずに捨ててしまう〝何もしないデバイス〟だ。

■12) デバイスドライバの拡張子はべつに〝.SYS〟である必要はなく，CONFIG.SYSに登録する名前とつじつまがあっていればなんでもかまわない。しかし，あやまって実行してしまわないように〝.X〟のままにしておくのだけはやめておいたほうがよいだろう。

■13) キャラクタデバイスドライバを作るときに，ソースファイル，実行ファイルのファイル名はデバイス名と同じにしてしまってはいけない。開発中はいいが，いざプログラムが完成し組み込んだ時点で，デバイス名と同名のファイルは読み書きできなくなってしまうからだ。

リスト5
CTESTDRV.S

```
 1: *       実験用デバイスドライバ
 2:
 3:         .include        doscall.mac
 4:         .include        const.h
 5:         .include        driver.h
 6: *
 7:         .text
 8:         .even
 9: *
10: *       デバイスヘッダ
11: *
12: device_header:
13:         .dc.l   -1
14:         .dc.w   CHAR_DEVICE+DISABLE_IOCTRL+COOKED_MODE
15:         .dc.l   strategy_entry
16:         .dc.l   interrupt_entry
17:         .dc.b   'CHRTEST '
18: *                12345678
19: *
20: request_header:                 *リクエストヘッダ待避領域
21:         .dc.l   0
```

```
22:
23: *
24: *       ストラテジルーチン
25: *
26: strategy_entry:
27:         move.l  a5,request_header       *リクエストヘッダへのポインタを
28:                                         *  待避して
29:         rts                             *速やかに戻る
30:
31: *
32: *       割り込みルーチン
33: *
34: interrupt_entry:
35:         movem.l d0/a4-a5,-(sp)          *レジスタ待避
36:
37:         movea.l request_header,a5       *a5=リクエストヘッダ
38:
39:         moveq.l #0,d0                   *d0.l=コマンドコード
40:         move.b  CMD_CODE(a5),d0         *
41:         add.w   d0,d0                   *2倍する
42:         add.w   d0,d0                   *2倍の2倍で4倍
43:         lea.l   jump_table,a4           *a4=ジャンプテーブル先頭
44:         adda.l  d0,a4                   *a4=コマンド処理ルーチンへの
45:                                         *  ポインタへのポインタ
46:         movea.l (a4),a4                 *a4=コマンド処理ルーチンへの
47:                                         *  ポインタ
48:         jsr     (a4)                    *a4の指すアドレスを
49:                                         *  サブルーチンコール
50:
51:         move.b  d0,ERR_LOW(a5)          *終了ステータスをセット
52:         lsr.w   #8,d0                   *
53:         move.b  d0,ERR_HIGH(a5)         *
54:
55:         movem.l (sp)+,d0/a4-a5          *レジスタ復帰
56:         rts                             *Humanへ戻る
57:
58: *
59: *       コマンド処理ジャンプテーブル
60: *
61: jump_table:
62:         .dc.l   init            *0      初期化
63:         .dc.l   notcmd          *1      (無効)
64:         .dc.l   notcmd          *2      (無効)
65:         .dc.l   ioctrl_in       *3      IOCTRLによる入力
66:         .dc.l   input           *4      入力
67:         .dc.l   sense           *5      1バイト先読み入力
68:         .dc.l   inpstat         *6      入力ステータスをチェック
69:         .dc.l   flush           *7      入力バッファをクリア
70:         .dc.l   output          *8      出力(VERIFY OFF)
71:         .dc.l   voutput         *9      出力(VERIFY ON)
72:         .dc.l   outstat         *10     出力ステータスをチェック
73:         .dc.l   notcmd          *11     (無効)
74:         .dc.l   ioctrl_out      *12     IOCTRLによる出力
75:
76: *
77: *       各コマンドの処理
78: *
```

```
 79:
 80: *
 81: *       無効（コマンドコード1,2,11）
 82: *       IOCTRLによる入力（コマンドコード3）
 83: *       IOCTRLによる出力（コマンドコード12）
 84: *
 85: notcmd:
 86: ioctrl_in:
 87: ioctrl_out:
 88:         move.w  #ILLEGAL_CMD,d0         *エラーコードを持って
 89:         rts                             *  戻る
 90:
 91: *
 92: *       入力（コマンドコード4）
 93: *
 94: input:
 95:         tst.l   DMA_LEN(a5)             *入力要求が0バイトであれば
 96:         beq     done                    *  何もせずに戻る
 97:                                     *そうでなければ
 98:         movea.l DMA_ADR(a5),a4          *a4=データ読み込み領域
 99:         move.b  #EOF,(a4)               *入力データをセット
100:
101: done:   moveq.l #0,d0                   *正常終了
102:         rts
103:
104: *
105: *       1バイト先読み入力（コマンドコード5）
106: *
107: sense:
108:         move.b  #EOF,SNS_DATA(a5)       *先読みデータをセット
109:         bra     done                    *正常終了
110:
111: *
112: inpstat:                *入力ステータスチェック（コマンドコード6）
113: flush:                  *入力バッファクリア（コマンドコード7）
114: output:                 *VERIFY OFF時の出力（コマンドコード8）
115: voutput:                *VERIFY ON時の出力（コマンドコード9）
116: outstat:                *出力ステータスチェック（コマンドコード10）
117:         bra     done                    *正常終了
118: *
119: *       ↑ここまでがメモリに居坐るデバイスドライバ本体
120:
121: *
122: *       初期化（コマンドコード0）
123: *
124: init:
125:         pea.l   title                   *タイトルを表示
126:         DOS     _PRINT                  *
127:         addq.l  #4,sp                   *
128:
129:         move.l  #init,DEV_END_ADR(a5)   *デバイスドライバ本体の
130:                                         *  最終アドレスをセット
131:
132:         bra     done                    *正常終了
133: *
134:         .data
135:         .even
```

```
136: *
137: title:                                    *タイトルメッセージ
138:         .dc.b   CR,LF
139:         .dc.b   '実験用キャラクタデバイス',CR,LF
140:         .dc.b   'CHRTESTの名前で入出力試験が行えます',CR,LF
141:         .dc.b   0
142:
143:         .end
```

では，リスト５の頭の部分から見ていこう。プログラムの先頭，12行からデバイスヘッダが始まっている。リンクポインタはデバイスドライバリンクの終わりを示す－１になっているが，組み込み時にHuman68kによって次のデバイスドライバを指すように勝手に書き換えられる。デバイス属性は，キャラクタデバイスで，ioctrlは不可で，COOKEDモード，という属性を与えてある。記号定数を使って書いてあるから一目瞭然だろう[14]。それから，ストラテジルーチンへのポインタ，割り込みルーチンへのポインタ，デバイス名が順に並んでいる。デバイス名は８バイト必要なので，CHRTESTの７文字の後ろにスペースが１個加えてある。

■14) ここで使っている定数はDRIVER.Hの中で定義されている。

26行からのストラテジルーチンでは，20行で用意したワークにリクエストヘッダへのポインタであるa5を格納している。そして，その後ろが肝心の割り込みルーチンだ。デバイスドライバを作る話もようやく佳境に入る。34行以下では割り込みルーチンの基本形が示されている。最初に，デバイスドライバ内で使用するレジスタをスタックに待避する(35行)。デバイスドライバ中ではレジスタの値を保存しておかなければならないのだ[15]。リスト５では使用するレジスタであるd0とa4,a5だけを待避しているが，いちいちレジスタの使用状況を調べるのが面倒であれば，d0～d7とa0～a6の全レジスタを待避してしまえばよい。ただし，movemは複数のレジスタを一命令で転送できるとはいえ，転送するレジスタの数が多いと実行に時間がかかるので，速度を気にするのであれば無駄なレジスタ待避はしないほうがいいだろう。

■15)ccrまでは気にしなくてよい。

続いて37行で，ストラテジルーチンでしまっておいたリクエストヘッダへのポインタをa5に取り出している。それからコマンドコードに応じて処理を振り分ける。各コマンドの処理は80行以下にサブルーチンの形で用意してある。ここではジャンプテーブルの手法によって，多方向への分岐をすっきりとまとめている。61行以下が各コマンドコードに対応した処理ルーチンの先頭アドレスを順に並べたジャンプテーブルだ。68000のアドレスは４バイトを占めるから，コマンドコードを４倍し，ジャンプテーブルの先頭アドレスに加え，〝コマンドの処理ルーチンの先頭アドレスを格納しているジャンプテーブル上の位置〟を求める(39～44行)。そこから処理ルーチンの先頭アドレスをレジスタに取り出し(46行)，そのアドレスをサブルーチンコールしている(48行)。bsrでは分岐先にラベルしか指定できないので，豊富なアドレッシングモード

が利用できるjsr命令を使った。48行の,

```
jsr     (a4)
```

は,〝a4の指すアドレスをサブルーチンコールする〟という意味だ。

処理がすんでサブルーチンから戻ってきたら,終了ステータスをリクエストヘッダ内に格納する。各コマンドの処理ルーチンは,終了ステータスをd0.wに入れて戻るように作ってあるので,51行でd0.wの下位バイトを,続いて8ビット左シフトして53行で上位バイトをセットしている。後は待避してあったレジスタを復帰してrtsでHuman68kに戻る。

割り込みルーチンの話

では,割り込みルーチンがサポートする各コマンドについて順に解説していこう。なお,本来ならキャラクタデバイスとブロックデバイスの両方を網羅すべきなのだろうが,ここではキャラクタデバイスの場合のみを取り上げる。また,CTESTDRV.Sはあまりに簡略化しすぎたため,あまりよい例にはなっていない。それでも,ないよりはましなので,適当にリストを参照しながら読み進めてもらいたい。

初期化

入力：	2(a5)	1B	コマンドコード(0)
	18(a5)	1L	引数へのポインタ
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)
	14(a5)	1L	デバイスドライバで使用する メモリの最終アドレス＋1

初期化ルーチンは,デバイスドライバ組み込み時にただ一度だけ呼び出される。CONFIG.SYSに記述された引数を受け取り,必要に応じて装置などの初期化を行った後,デバイスドライバで使用するメモリの最終アドレス+1をリクエストヘッダ内にセットして戻る。引数は〝DEVICE = ～〟行の〝=〟の直後を指すポインタの形で渡される。ただし,〝.X〟ファイルのコマンドライン文字列とは異なり,引数はあらかじめHuman68kによって語単位で区切られている。CONFIG.SYSに,

```
DEVICE = A:¥SYS¥RAMDISK.SYS #G
```

と記述してあったとすると,引数は,

```
.dc.b   'A:¥SYS¥RAMDISK.SYS', 0
.dc.b   '#G', 0
.dc.b   0
```

のように分解される。空白を0に置き換え，最後にもう1つ0を付け加えた形だ[16]。

■16) 空白はいくつあっても1個の0に置き換えられる。

通常，初期化ルーチンはデバイスドライバプログラムの最後に置き，初期化後は切り離す。つまり，OSに戻るときに初期化ルーチンの先頭アドレスを14(a5)にセットするわけだ。このとき，以後のデバイスドライバの動作に必要なデータまであやまって切り離さないように十分注意しなければならない。メモリ上のプログラムはつねにテキストセクション，データセクション，ブロックストレージセクションの順に並んでいるから，テキストセクションにある初期化ルーチン以降を切り離すと，データセクションやブロックストレージセクションもいっしょになくなってしまう。デバイスドライバを作成する際には，データ類も初期化ルーチンより前のテキストセクションに置いておかなければならないことになる。

入力(コマンドコード4)

入力：	2(a5)	1B	コマンドコード(4)
	14(a5)	1L	転送先バッファへのポインタ
	18(a5)	1L	入力要求バイト数
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)

装置から指定されただけのバイト数を読み込み，指定バッファに順に書き込む。データが用意できていない場合は揃うまで待つ[17]。なお，COOKEDモードでは入力要求バイト数はつねに1になるはずだが，前章でふれたように，COOKEDモードとRAWモードはioctrlで切り替えることができるため，デバイス属性がCOOKEDでもRAWモードでアクセスされる可能性がある。また，入力要求バイト数が0ということもありうるようだ。よって，デバイスドライバの入力処理ルーチンは，入力要求バイト数が何バイトであろうとも過不足なく処理が行えるように作る必要がある。

■17)たとえば入力先デバイスがキーボードであるならキーが押されるまで，RS-232Cポートならデータが送信されてくるまで待つわけだ。

先読み入力(コマンドコード5)

入力：	2(a5)	1B	コマンドコード(8, 9)
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)
	13(a5)	1B	先読みしたデータ

装置から1バイト先読みし，リクエストヘッダ内にセットして戻る。先読みだから，読み込んだデータは後でコマンドコード4で入力される場合に備えて残しておく。もしデータがない

場合はデータが揃うのを待ったりせず，即座に0を返す。

入出力ステータスチェック(コマンドコード8, 10)

入力：	2(a5)	1B	コマンドコード(8, 9)
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)

装置が現在入力可能かどうか(コマンドコード6)，出力可能かどうか(コマンドコード10)を調べ，可能であれば0000_Hを，不可能であれば0001_Hを終了ステータスとして返す。正常終了時に0000_H以外の終了ステータスを返す唯一の例外だ。

入力バッファクリア(コマンドコード7)

入力：	2(a5)	1B	コマンドコード(8, 9)
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)

入力データを一度内部のバッファに溜め込み，入力要求があった場合はそのバッファからデータを取り出して返すような構成のデバイスドライバにおいて，そのバッファに溜まっていたデータを破棄するコマンドだ。コマンドコード5で先読みしていたデータが保存されているときにはいっしょに捨てる。

出力(コマンドコード8，9)

入力：	2(a5)	1B	コマンドコード(8, 9)
	14(a5)	1L	出力データへのポインタ
	18(a5)	1L	出力要求バイト数
出力：	3(a5)	1B	終了ステータス(下位)
	4(a5)	1B	終了ステータス(上位)

指定アドレス以降のデータを指定されただけのバイト数分装置に出力する。コマンドコード9の場合は，出力したデータをすかさず読み込み，出力前のデータと比較して正しく出力できているかどうかを確認する(万が一，一致しない場合は書き込みエラーを返す)。入力コマンドのところで行ったCOOKEDモードとRAWモードに関する注意は，そのまま当てはまる。

ioctrlによる入出力(コマンドコード3,12)

リクエストヘッダの構成はそれぞれ，コマンドコード4，コマンドコード8と同様なので省略する。この機能をサポートしない場合は無効なコマンドコードとしてきちんとはじかなければならない。

入出力コマンドはともかく，その他のコマンドがどのような状況で呼び出されるかというのはなかなか興味深い問題だ。そこで，ものは試し，CTESTDRV.Sに与えられたコマンドを表示する試験用ルーチンを追加してみよう。リスト5の38行に，

```
bsr     test
```

の1行を付け加え，リスト6を初期化ルーチンより前，118行あたりにでも挿入し，再アセンブルしてもらいたい。これによって，CHRTESTが呼び出されるたびに画面に青でコマンドの種類が表示されるようになる。ついでに入出力系のコマンドの場合は，入出力要求されたバイト数もいっしょに表示するようにしてある。本来文字表示を行うはずのないタイミングで強引に表示を行っていることもあり，画面がかなりうるさくなるし，ときには文字化けを起こすこともあるが，試験用ということで勘弁してもらいたい。

リスト6
CTESTDRV.S
追加部

```
 1: *
 2: *       試験用ルーチン
 3: *
 4: test:
 5:         move.l  d1,-(sp)
 6:
 7:         bsr     showcmd             *コマンドの種類を表示
 8:         bsr     showlen             *入出力系コマンドであれば
 9:                                     *  データ長を表示
10:         move.l  (sp)+,d1
11:         rts
12:
13: *
14: *       コマンドの種類を表示する
15: *
16: showcmd:
17:         moveq.l #0,d0               *コマンドの種類を表す文字列の
18:         move.b  CMD_CODE(a5),d0     *  先頭アドレスをa4に得る
19:         add.w   d0,d0               *
20:         add.w   d0,d0               *
21:         lea.l   cmd_table,a4        *
22:         add.l   d0,a4               *
23:         movea.l (a4),a4             *
24:
25:         move.l  a4,-(sp)            *コマンドの種類を表示する
26:         move.w  #1,-(sp)            *
27:         DOS     _CONCTRL            *
28:
29:         move.l  #crlfms,2(sp)       *改行する
30:         DOS     _CONCTRL            *
31:         addq.l  #6,sp               *
```

```
32:          rts
33:
34: *
35: *        メッセージへのポインタのテーブル
36: *
37: cmd_table:
38:          .dc.l    mes00,mes01,mes02,mes03
39:          .dc.l    mes04,mes05,mes06,mes07
40:          .dc.l    mes08,mes09,mes10,mes11
41:          .dc.l    mes12
42:
43: *
44: *        入出力系コマンドであればデータ長を16進表示する
45: *
46: showlen:
47:          moveq.l  #0,d0                    *d0.l=コマンド番号
48:          move.b   CMD_CODE(a5),d0          *
49:          move.l   #%00010011_00011000,d1   *入出力系コマンドかどうかを
50:          btst.l   d0,d1                    *  調べる
51:          beq      slen0                    *そうでなければ何もしない
52:
53:          pea.l    temp                     *データ長を16進8桁に変換する
54:          move.l   DMA_LEN(a5),-(sp)        *
55:          bsr      itoh                     *
56:          addq.l   #8,sp                    *
57:          pea.l    temp                     *表示する
58:          move.w   #1,-(sp)                 *
59:          DOS      _CONCTRL                 *
60:
61:          move.l   #crlfms,2(sp)            *改行する
62:          DOS      _CONCTRL                 *
63:          addq.l   #6,sp                    *
64: slen0:   rts
65:
66: *
67: *        数値→16進文字列変換
68: *
69: value    =        8
70: buff     =        12
71: *
72: itoh:
73:          link     a6,#0
74:          movem.l  d0-d2/a0,-(sp)
75:
76:          move.l   value(a6),d0     *値
77:          movea.l  buff(a6),a0      *文字列格納アドレス
78:
79:          moveq.l  #8-1,d2          *以下を８回繰り返す
80:
81: itoh0:   rol.l    #4,d0            *d0.lを左に４ビット回転する
82:          move.b   d0,d1            *d0の下位バイトをd1に取り出し
83:          andi.b   #$0f,d1          *  下位４ビットを残してマスクする
84:          addi.b   #'0',d1          *ここで数値から16進を表す文字へ
85:          cmpi.b   #'9'+1,d1        *  変換する
86:          bcs      itoh1            *  0～9の場合は'0'を足すだけだが
87:          addq.b   #'A'-'0'-10,d1   *  A～Fの場合はさらに補正が必要
88:
```

```
 89: itoh1:  move.b  d1,(a0)+        *変換した文字をしまう
 90:
 91:         dbra    d2,itoh0        *繰り返す
 92:
 93:         clr.b   (a0)            *文字列終端コードを書き込む
 94:
 95:         movem.l (sp)+,d0-d2/a0
 96:         unlk    a6
 97:         rts
 98:
 99: *
100: *       コマンドの種類表示用文字列
101: *
102: mes00:  .dc.b   '初期化',0
103: mes01:  .dc.b   'コマンド１（無効）',0
104: mes02:  .dc.b   'コマンド２（無効）',0
105: mes03:  .dc.b   'IOCTRLによる入力',0
106: mes04:  .dc.b   '入力',0
107: mes05:  .dc.b   '先読み入力',0
108: mes06:  .dc.b   '入力ステータスをチェック',0
109: mes07:  .dc.b   '入力バッファをクリア',0
110: mes08:  .dc.b   '出力（VERIFY OFF）',0
111: mes09:  .dc.b   '出力（VERIFY ON）',0
112: mes10:  .dc.b   '出力ステータスをチェック',0
113: mes11:  .dc.b   'コマンド11（無効）',0
114: mes12:  .dc.b   'IOCTRLによる出力',0
115: *
116: crlfms: .dc.b   CR,LF,0                    *改行用文字列
117: *
118: temp:   .ds.b   8+1                       *16進変換用バッファ
119: *
120:         .even
```

組み込んで，

```
TYPE CONFIG.SYS >CHRTEST
```

などとやってみると，１文字ごとにデバイスドライバが呼び出される様子を見ることができる。

```
COPY CHRTEST TEMPFILE
```

とか，

```
UPPER CHRTEST
```

とか，いろいろ試してみるとデバイスドライバの動作がよりよくつかめるだろう。デバイス属性をRAWモードに変更してアセンブルし直し，COOKEDモードとRAWモードの違いを確認するのもおもしろい。

プログラムについてもいちおうふれておこう。コマンド名の表示は４行以下のサブルーチンshowcmdで行っている。コマンド名を表す文字列へのポインタをテーブルにしておき，ジャンプテーブルのときとほとんど同じ手順で文字列へのポインタを取り出し，DOSコールconctrlで表示している。printではなく，conctrlを使っているのは，

```
TYPE CONFIG.SYS >CHRTEST
```

のようにリダイレクトされた場合に備えてのことだ。リダイレクトすることで標準出力にCHRTESTが割り付けられ，そのCHRTESTの処理ルーチンの中からさらに標準出力に出力すると，自分自身を呼び出すわけだから無限ループに陥ってしまう。

プログラム上のテクニックとしては47～51行をチェックしておきたい。ここではコマンドコードが入出力系のコマンド(3, 4, 8, 9, 12)であるかどうかをbtst命令を使って調べている。d1に第3, 4, 8, 9, 12ビットだけが1であるようなデータを，d0にコマンドコードを入れておき，

```
btst.l   d0,d1
```

を実行すれば，結果のZビットだけでd0が上記5つの値のどれかと一致しているかどうかがわかるという寸法だ。cmpを5個並べるよりはずーっとスマートだと思うが，どうだろうか。

実用的サンプルプログラム

CTESTDRV.Sがあまり参考にならなかったので，もう1つ，今度はもっと実用になるデバイスドライバを作ってみよう。次のような動作をするキャラクタデバイスを考える。

1) 出力時には出力データを画面に表示しながら，同時に内部のバッファにも溜め込む。

2) 入力時には，1)で溜め込んだデータを返す。

知っている人は知っているように，これは電脳倶楽部のVol.1を飾ったプログラムと同機能だ。なかなか便利だし，プログラムの題材としても手ごろなのでアイデアを拝借してきた。オリジナルに敬意を表してデバイス名も同名のTAPにしてしまおう。さきほどのCTESTDRV.Sにコマンド処理ルーチンを加えていったら，リスト7のように仕上がった。なお，オリジナルではバッファの大きさをCONFIG.SYS中で指定できるようになっているのだが，リスト7では固定されている。サイズを変更したいときは201行の定数をかげんしてアセンブルし直すことになる。

リスト7
TAPDRV.S

```
  1: *       TAPドライバ（バッファサイズ固定版）
  2:
  3:         .include        doscall.mac
  4:         .include        const.h
  5:         .include        driver.h
  6: *
  7:         .text
  8:         .even
  9: *
 10: *       デバイスヘッダ
 11: *
 12: device_header:
 13:         .dc.l   -1
 14:         .dc.w   CHAR_DEVICE+DISABLE_IOCTRL+COOKED_MODE
 15:         .dc.l   strategy_entry
```

```
16:         .dc.l   interrupt_entry
17:         .dc.b   'TAP'
18: *               12345678
19: *
20: request_header:                 *リクエストヘッダ待避領域
21:         .dc.l   0
22:
23: *
24: *       ストラテジルーチン
25: *
26: strategy_entry:
27:         move.l  a5,request_header       *リクエストヘッダへのポインタを
28:                                         *  待避して
29:         rts                             *速やかに戻る
30:
31: *
32: *       割り込みルーチン
33: *
34: interrupt_entry:
35:         movem.l d0-d2/a0-a1/a4-a5,-(sp) *レジスタ待避
36:
37:         movea.l request_header,a5       *a5=リクエストヘッダ
38:
39:         moveq.l #0,d0                   *d0.l=コマンドコード
40:         move.b  CMD_CODE(a5),d0         *
41:         add.w   d0,d0                   *2倍する
42:         add.w   d0,d0                   *2倍の2倍で4倍
43:         lea.l   jump_table,a4           *a4=ジャンプテーブル先頭
44:         adda.l  d0,a4                   *a4=コマンド処理ルーチンへの
45:                                         *  ポインタへのポインタ
46:         movea.l (a4),a4                 *a4=コマンド処理ルーチンへの
47:                                         *  ポインタ
48:         jsr     (a4)                    *a4の指すアドレスを
49:                                         *  サブルーチンコール
50:
51:         move.b  d0,ERR_LOW(a5)          *終了ステータスをセット
52:         lsr.w   #8,d0                   *
53:         move.b  d0,ERR_HIGH(a5)         *
54:
55:         movem.l (sp)+,d0-d2/a0-a1/a4-a5 *レジスタ復帰
56:         rts                             *Humanへ戻る
57:
58: *
59: *       コマンド処理ジャンプテーブル
60: *
61: jump_table:
62:         .dc.l   init            *0      初期化
63:         .dc.l   notcmd          *1      (無効)
64:         .dc.l   notcmd          *2      (無効)
65:         .dc.l   ioctrl_in       *3      IOCTRLによる入力
66:         .dc.l   input           *4      入力
67:         .dc.l   sense           *5      1バイト先読み入力
68:         .dc.l   inpstat         *6      入力ステータスをチェック
69:         .dc.l   flush           *7      入力バッファをクリア
70:         .dc.l   output          *8      出力 (VERIFY OFF)
71:         .dc.l   voutput         *9      出力 (VERIFY ON)
72:         .dc.l   outstat         *10     出力ステータスをチェック
```

```
 73:         .dc.l   notcmd          *11     (無効)
 74:         .dc.l   ioctrl_out      *12     IOCTRLによる出力
 75:
 76: *
 77: *       各コマンドの処理
 78: *
 79:
 80: *
 81: *       無効（コマンドコード1,2,11）
 82: *       IOCTRLによる入力（コマンドコード3）
 83: *       IOCTRLによる出力（コマンドコード12）
 84: *
 85: notcmd:
 86: ioctrl_in:
 87: ioctrl_out:
 88:         move.w  #ILLEGAL_CMD,d0         *エラーコードを持って
 89:         rts                             *  戻る
 90:
 91: *
 92: *       入力（コマンドコード4）
 93: *
 94: input:
 95:         move.l  DMA_LEN(a5),d2          *入力要求が0バイトであれば
 96:         beq     done                    *  何もせずに戻る
 97:                                 *そうでなければ
 98:         movea.l readptr,a0              *a0=読み出し位置
 99:         movea.l DMA_ADR(a5),a4          *a4=データ読み込み領域
100:                                         *d2.l=入力要求バイト数
101:
102: inp0:   cmpa.l  writeptr,a0              *データがもうなければ
103:         beq     empty                   *  ループを抜ける
104:         move.b  (a0)+,(a4)+             *1バイト転送
105:         cmpa.l  buffend,a0              *ポインタがバッファ最後を
106:         bcs     inp1                    *  越えたら
107:         lea.l   bufftop,a0              *  先頭を指すように修正する
108: inp1:   subq.l  #1,d2                   *ループカウンタd2.lが0になるまで
109:         bne     inp0                    *  繰り返す
110:
111:         move.l  a0,readptr              *読み出し用ポインタ更新
112:
113: done:   moveq.l #0,d0                   *正常終了
114:         rts
115:
116: empty:  move.b  #EOF,(a4)               *バッファが空の場合は
117:                                         *  EOFを返す
118:         bra     done                    *正常終了
119:
120: *
121: *       VERIFY OFF時の出力（コマンドコード8）
122: *       VERIFY ON時の出力（コマンドコード9）
123: output:
124: voutput:
125:         move.l  DMA_LEN(a5),d2          *入力要求が0バイトであれば
126:         beq     done                    *  何もせずに戻る
127:                                 *そうでなければ
128:         movea.l writeptr,a0             *a0=次に書き込む位置
129:         movea.l readptr,a1              *a1=次に読み出す位置
```

```
130:            movea.l DMA_ADR(a5),a4          *a4=出力データ
131:                                            *d2.l=出力要求バイト数
132:
133:            moveq.l #0,d1                   *作業用レジスタをクリア
134:
135: out0:      move.b  (a4)+,d1                *１バイト取り出す
136:            move.b  d1,(a0)+                *バッファに追加
137:
138:            cmpi.b  #EOF,d1                 *EOFコードは画面クリアの
139:                                            *  コントロールコードなので
140:            beq     out1                    *  表示はしない
141:
142:            move.l  d1,-(sp)                *１バイト画面に出力
143:            DOS     _CONCTRL                *
144:            addq.l  #4,sp                   *
145:
146: out1:      cmpa.l  buffend,a0              *ポインタがバッファ最後を
147:            bcs     out2                    *  越えたら
148:            lea.l   bufftop,a0              *  先頭を指すように修正する
149:
150: out2:      cmpa.l  a1,a0                   *書き込み位置が読みだし位置に
151:            bne     out3                    *  追いついてしまった場合は
152:            addq.l  #1,a1                   *  読みだし位置を強制的にずらす
153:
154:            cmpa.l  buffend,a1              *その結果読み出し位置が
155:            bcs     out3                    *  バッファ最後を越えたら
156:            lea.l   bufftop,a1              *  先頭を指すように修正する
157:
158: out3:      subq.l  #1,d2                   *ループカウンタd2.lが0になるまで
159:            bne     out0                    *  繰り返す
160:
161:            move.l  a0,writeptr             *書き込み用ポインタ更新
162:            move.l  a1,readptr              *読み出し用ポインタ更新
163:
164:            bra     done                    *正常終了
165:
166: *
167: *       １バイト先読み入力（コマンドコード5）
168: *
169: sense:
170:            moveq.l #EOF,d0                 *仮にEOFコードを入れておく
171:            movea.l readptr,a0              *読み出しポインタと
172:            cmpa.l  writeptr,a0             *  書き込みポインタが
173:            beq     sense0                  *  等しければバッファは空
174:            move.b  (a0),d0                 *そうでなければ何かあるから
175:                                            *  ポインタは固定のまま取り出す
176: sense0:    move.b  d0,SNS_DATA(a5)         *先読みデータをセット
177:            bra     done                    *正常終了
178:
179: *
180: *       入力バッファクリア（コマンドコード7）
181: flush:
182:            move.l  writeptr,readptr        *書き込み位置と
183:                                            *  読み出し位置を一致させる
184:            bra     done                    *正常終了
185:
186: *
```

```
187: inpstat:                   *入力ステータスチェック（コマンドコード6）
188: outstat:                   *出力ステータスチェック（コマンドコード10）
189:        bra     done                        *正常終了（常に入出力可）
190: *
191: readptr:
192:        .dc.l   bufftop                     *次に読み出す位置を指すポインタ
193: writeptr:
194:        .dc.l   bufftop                     *次に書き込む位置を指すポインタ
195: buffend:
196:        .dc.l   0                           *バッファ最終アドレス+1
197: *
198: *       ↓以下をバッファとして使用
199: bufftop:
200:
201: BUFFSIZE       =       16*1024             *バッファのバイト数
202:
203: *
204: *       初期化部（コマンドコード0）
205: *
206: init:
207:        pea.l   title                       *タイトルを表示
208:        DOS     _PRINT                      *
209:        addq.l  #4,sp                       *
210:
211:        lea.l   bufftop+BUFFSIZE,a4         *a4 = バッファ最終アドレス+1
212:        move.l  a4,buffend                  *
213:        move.l  a4,DEV_END_ADR(a5)          *デバイスドライバで使用する
214:                                            *  メモリの最終アドレスをセット
215:
216:        bra     done                        *正常終了
217:
218: *
219: ent:                          *安全のため
220:        DOS     _EXIT
221: *
222:        .data
223:        .even
224: *
225: title:                                     *タイトルメッセージ
226:        .dc.b   CR,LF,'TAP DRIVER for X68000',CR,LF
227:        .dc.b   'TAPのデバイス名で入出力が行えます',CR,LF,0
228: *
229:        .end    ent
```

さて，このプログラムではデバイスドライバとしての構造云々よりも，データを(後でちゃんと順序よく取り出せるような形で)溜め込む処理そのものが重要なポイントだ。こういった場合は，リングバッファというデータ構造を使うのが定石になっている。リングバッファはFIFO(First In First Out：最初に入れたものが最初に出てくる)を実現するデータ構造で，キューを形成するのによく用いられる。あらたに格納されるデータはバッファ内のデータ列末尾に付け加えられ，取り出すときは列の先頭のデータが取り出される。ちなみに，スタックは最初に入れたデータが最後に出てくるLIFO(Last In First Out)のデータ構造だ。

スタックはただ1個のポインタで管理されるが，リングバッファの場合はデータを書き込む

位置を示すポインタと，データを読み出す位置を表すポインタの2つを使う。データがなにもない状態では，図5-aにあるように，書き込み位置を表すポインタ(図ではWで表した)と読み出し位置を表すポインタ(同じくRで表した)は同じ位置を指している。ここから書き込み位置を進めつつデータを書き込んでいくと，図5-bのようになる。WとRの差がバッファ内のデータの個数を表している。読み出すときには，読み出し位置のポインタのほうを進めながらデータを取り出していく。RがWに追いついたときにはバッファは空になっている(図5-c)。こうやって読み書きを行っていくと，いつかはポインタがバッファの最後を越えてしまうことになるが，そうなったらすかさずバッファの先頭を指すように修正してやる。バッファの最後と最

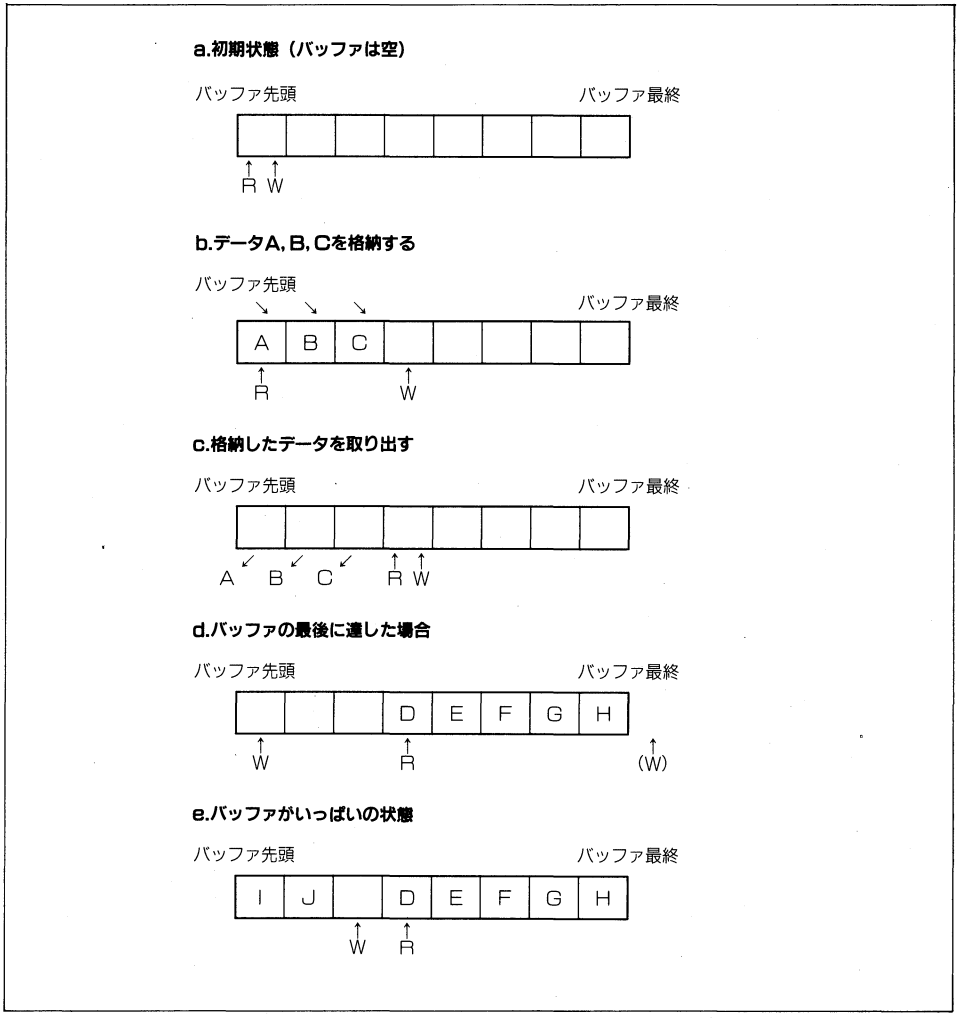


図5
リングバッファ

初が論理的にはつながって輪になっていると考えるわけだ(図5-d)。

また,読み出しを行わずに書き込みだけを続けると,Rよりさきを指していたはずのWがぐるっと回ってRに追いつくことになる。完全にRとWが一致してしまうと,バッファが空の状態と区別できなくなるので,その直前の状態(図5-e)をもってバッファがいっぱいであることを表すと考える。この場合,バッファに格納できるデータの最大数はバッファの大きさよりデータ1個分少なくなる。それがもったいないと思うなら多少プログラムが複雑になるが,バッファ内のデータの個数を数えるカウンタを設ければバッファの大きさいっぱいまで使えるようにできる。

ここまでが一般的なリングバッファの考え方だが,これをTAPドライバに採用するにあたっては2点ほど考えておかなければならないことがある。1つはバッファが空のときに入力が要求されたときの処理だ。デバイスドライバの仕様では,データがない場合はデータが入力されるのを待つことになっているが,TAPでは入力コマンドの処理中はどんなに待ってもデータは入ってくるはずがない。そこで,バッファが空の場合はEOFコードを返すことにする。

もう1つはバッファがいっぱいのときに出力が要求された場合だ。対策としては,

1) 即座にエラーを返す。

2) 新しいデータはバッファに追加せずに戻る。

3) 古いデータを消して新しいデータを格納する。

といった手段が考えられる。しかし,どれも完全とはいいがたい。1)はある意味で正しいデバイスドライバの作り方だと思うが,実行中のプログラムを止めなければならなくなる。2),3)は,プログラムは止まらないかわりにバッファ内のデータが失われる。迷ったのだが,結局,リスト7では3)の方法を採用した。

では,かんたんにプログラムの解説をしていこう。

まず,206行以下の初期化部分。ここではタイトルメッセージの表示とデバイスドライバで使用するメモリの最後をリクエストヘッダに格納しているだけで,CTESTDRV.Sと実質的な違いはない。ただし,初期化部をたんに切り離すのではなく,ついでにリングバッファの分のメモリを確保している。199行のラベルbufftopがバッファの先頭を表しており,ここから定数BUFFSIZEを足した位置までをバッファとして使用する。バッファを.dsで確保せずに,初期化ルーチンと重ねてあるのがポイントだ。いつものようにブロックストレージセクションにバッファを.dsで確保したとすると,デバイスドライバ本体とバッファで初期化ルーチンを挟む形になり,初期化ルーチンを切り離すことができなくなる。といって,初期化ルーチンの前にバッファを確保しようとすると,textセクションに.dsを書かなければならず,実行ファイルがバッファの大きさ分膨れ上がってしまう。

```
            .text
            デバイスドライバ本体
            .bss
buff:   .ds.b   16 * 1024
```

```
        .text
    初期化ルーチン
```

のようにソース上で順序を変えてみたところで，アセンブル・リンク時に各セクションは1つにまとめられてしまうので無駄だ。

94行以下が入力処理ルーチンだ。a0をバッファ内の読み出し位置を指すポインタ，a4を転送先へのポインタ，d2.lをループカウンタとして使用している。基本的には，

```
    move.b   (a0)+,(a4)+
```

を指定された入力バイト数回繰り返しているだけだが，バッファが空の場合にループを抜けてEOFコードを返す処理，およびポインタがバッファの最後を越えたら，バッファ先頭を指すように修正する処理が挟まっている。

また，123行からが出力ルーチンだ。ちょっと手を抜いてVERIFY ONのときの処理とOFFのときの処理を共通にし，ベリファイを省略してしまったが，それほど大きな問題にはならないだろう。バッファがいっぱいになったときに古いデータから消していく処理は，書き込み用のポインタが読み出し用のポインタに追いついたら，強制的に読み出し用ポインタを1バイト進めることで対処している。後，142行以下のCONCTRLの呼び出し方は多少姑息かもしれない。

```
    move.w    d1,-(sp)
    clr.w     -(sp)
```

とすべきところを，d1.lの上位ワードをあらかじめ0クリアしておくことで，

```
    move.l    d1,-(sp)
```

の一命令に置き換えている。

169行からの先読み入力処理では，バッファが空でなければ読み出し位置から1バイト取り出して返し，空であれば0ではなくEOFを返している。これはTAPの仕様である。また，181行の入力バッファクリア処理はリングバッファならではの単純さだ。リングバッファでは読み出し位置と書き込み位置を一致させるだけで，バッファが空になるわけだ。

TAPDRV改良版

では，最後におまけとして，TAPのバッファサイズをCONFIG.SYSで指定できるように拡張する方法を示す。リスト7の199行以下をリスト8に置き換えると，起動時のパラメータを解釈してバッファサイズを決める処理が追加される。数値の表示に以前作ったサブルーチンput-decを使っているので忘れずにリンクしてもらいたい。バッファサイズは1Kバイト単位で，

```
    DEVICE = TAPDRV.SYS #/B64
```

のように指定するようになっている。〝#/B〟というのが冗長だが，これはPRNDRV.SYSやオリジナル版TAPにあわせたためだ。

プログラム中では，10進文字列から数値へ変換する処理を行う85行以下のサブルーチンatoiと，1Kバイトの単位からバイトに変換するためにビットシフトを利用して1024倍にしている18～19行はきちんと理解しておいてもらいたい。両者ともに，コラム「乗算」が参考になるだろう。

リスト8
TAPDRV.S追加部

```
  1: *       TAPドライバ（バッファサイズ可変版）

199: bufftop:
200:
201:        .xref   putdec                  *外部参照
202: *
203: *       初期化部（コマンドコード0）
204: *
205: init:
206:
207:        pea.l   title                   *タイトルを表示
208:        DOS     _PRINT                  *
209:        addq.l  #4,sp                   *
210:
211:        bsr     getbufsiz               *バッファサイズ取得
212:
213:        move.w  d0,-(sp)                *(sp)=バッファサイズ（単位K）
214:
215:        lea.l   bufftop,a4              *a4=バッファ先頭
216:        moveq.l #10,d1                  *1024倍
217:        lsl.l   d1,d0                   *
218:                                        *d0.l=バッファサイズ
219:        add.l   d0,a4                   *a4=バッファ最終+1
220:        move.l  a4,buffend              *
221:        move.l  a4,DEV_END_ADR(a5)      *デバイスドライバで使用する
222:                                        *  メモリの最終アドレスをセット
223:
224:        bsr     putdec                  *バッファサイズを表示
225:        addq.l  #2,sp                   *
226:
227:        pea.l   mes2                    *初期化完了メッセージを表示
228:        DOS     _PRINT                  *
229:        addq.l  #4,sp                   *
230:
231:        bra     done                    *正常終了
232:
233: *
234: *       CONFIG.SYSで指定されたバッファサイズを取得する
235: *
236: getbufsiz:
237:        movea.l PAR_PTR(a5),a4          *a4=パラメータ先頭
238:
239: *       'TAPDRV.SYS',0,'#/B64',0,0
240: *         'a4
241:
242: skip:   tst.b   (a4)+                   *ファイル名を飛ばす
243:        bne     skip                    *
244:
245: *       'TAPDRV.SYS',0,'#/B64',0,0
```

```
246: *                          a4
247:
248:          tst.b    (a4)                    *パラメータがなければ
249:          beq      default                 *  デフォルト値を使う
250:
251:          cmpi.b   #'#',(a4)+              *'#/B'の並びを順にチェック
252:          bne      inierr                  *
253:          cmpi.b   #'/',(a4)+              *
254:          bne      inierr                  *
255:          move.b   (a4)+,d0                *
256:          andi.b   #%1101_1111,d0          *大文字化
257:          cmpi.b   #'B',d0                 *
258:          bne      inierr                  *
259:
260: *        'TAPDRV.SYS',0,'#/B64',0,0
261: *                              a4
262:
263:          bsr      atoi                    *文字列→数値変換
264:          tst.w    d0                      *0なら
265:          beq      inierr                  *  正しくない
266:          cmpi.l   #1024+1,d0              *上限のチェック
267:          bcc      inierr                  *
268:
269:          rts                         *d0.w=バッファサイズ（単位K）
270: *
271: inierr:
272:          pea.l    mes1                    *エラーメッセージを表示
273:          DOS      _PRINT                  *
274:          addq.l   #4,sp                   *
275: *        bra      default
276: default:
277:          moveq.l  #16,d0                  *仮に16Kバイト確保
278:          rts
279:
280: *
281: *        文字列→数値変換
282: *
283: atoi:
284:          moveq.l  #0,d0                   *結果を入れるd0.lをクリア
285:          moveq.l  #0,d1                   *
286: atoi0:   move.b   (a4)+,d1                *１文字取り出す
287:          subi.b   #'0',d1                 *文字→数値変換
288:          bcs      atoi1                   *
289:          cmpi.b   #'9'+1,d1               *
290:          bcc      atoi1                   *
291:          mulu.w   #10,d0                  *10倍して
292:          add.w    d1,d0                   *  １桁追加
293:          bra      atoi0                   *繰り返す
294: atoi1:   rts
295:
296: *
297: ent:                                     *安全のため
298:          DOS      _EXIT
299: *
300:          .data
301:          .even
302: *
```

```
303: title:                                    *タイトルメッセージ
304:          .dc.b   CR,LF,'TAP DRIVER for X68000',CR,LF,0
305: mes1:    .dc.b   'パラメータの指定に誤りがあります',CR,LF
306:          .dc.b   'バッファサイズは以下の形式で指定します',CR,LF
307:          .dc.b   TAB,'DEVICE = TAPDRV.SYS #/Bn',CR,LF
308:          .dc.b   TAB,'（nは1Kバイト単位）',CR,LF
309:          .dc.b   '仮に',0
310: mes2:    .dc.b   'Kバイトのバッファを確保しました',CR,LF
311:          .dc.b   'TAPのデバイス名で入出力が行えます',CR,LF,0
312: *
313:          .end    ent
```

COLUMN

乗算

除算命令divu, divsは以前紹介したし，すでに何度か使ってもいる。ここでは，これらと対になる乗算命令mulu，mulsを紹介する。

mulu，mulsは，

```
mulu.w    #10,d0
mulu.w    d1,d0
```

のようにして使い，データレジスタの下位ワードに16ビット数をかけ，結果を32ビットで求める命令だ(サイズはワード固定)。上の例ではd0.w×10，d0.w×d1.wをd0.lに求めている。muluが無符号演算で，mulsが符号付きなのはdivu，divsの関係と同様だ。オーバーフローはありえない($FFFF_H$を自乗しても32ビットで収まる)ので，演算の結果，ccrのCビット，Vビットはつねにリセットされる。

乗算を行う専用命令が用意されているのはたしかに便利なのだが，68000の乗算命令は内部ではもっと原始的な処理に展開されているらしく，addなどの単純な命令よりもかなり実行時間がかかる。最悪の場合20倍近い。このため，乗算命令を使うまでもないようなかんたんなケースでは加減算やシフトを利用して積を計算することがよくある。表Aにd0.wを定数倍するパターンをいくつか挙げてみた。複数の実現法が考えられる場合にはもっとも実行時間が短くなるものを選んである(はず)。ただし，ここでは符号とオーバーフローは考慮していない。また，作業用にd1を使っていることがある。

表A

```
2倍     add.w   d0,d0
3倍     move.w  d0,d1
        add.w   d0,d0
        add.w   d1,d0
4倍     add.w   d0,d0
        add.w   d0,d0
5倍     move.w  d0,d1
        add.w   d0,d0
        add.w   d0,d0
        add.w   d1,d0
6倍     move.w  d0,d1
        add.w   d0,d0

        add.w   d1,d0
        add.w   d0,d0
7倍     move.w  d0,d1
        lsl.w   #3,d0
        sub.w   d1,d0
8倍     lsl.w   #3,d0
16倍    lsl.w   #4,d0
32倍    lsl.w   #5,d0
256倍   lsl.w   #8,d0
1024倍  moveq.l #10,d1
        lsl.w   d1,d0
```

CHAPTER 1

脱〝入門編〟のための身辺整理

CHAPTER

脱"入門編"のための身辺整理

ASSEMBLER

本章では，これまでの章で十分ふれることができなかった部分をフォローしながら明日へつなぐという主旨で，補足的なこまごまとした話題を集めてみたい。〝知らないよりは知っているほうがいい〟という程度の話題ばかりだから，頭と時間に余裕があるときに拾い読みしてもらうのがいちばんいいかもしれない。

まずは68000のアドレッシングモードの総ざらいから始める。

アドレッシングモードの総まとめ

68000は本当に豊富で強力なアドレッシングモードを備えている。この本ではできるだけ複雑なアドレッシングモードを使うのを避けてきたが，それでも以下の9種類のアドレッシングモードが登場している([ ]内は例)。

・データレジスタ直接形式[move.l d1, d0]
・アドレスレジスタ直接形式[move.l a0, d0]
・アドレスレジスタ間接形式[move.l (a0), d0]
・ポストインクリメントアドレスレジスタ間接形式[move.l (a0)+, d0]
・プリデクリメントアドレスレジスタ間接形式[move.l -(a0), d0]
・ディスプレースメント付きアドレスレジスタ間接形式[move.l 4(a0), d0]
・イミディエイトデータ形式[move.l #1234, d0]
・クイック・イミディエイトデータ形式[moveq.l #10, d0]
・絶対ロングアドレス形式[move.l LABEL, d0]

どれももうおなじみのものばかりだと思う。いまさら解説することはなにもない。ここでは，確認の意味で，比較的引っかかりやすいアドレスレジスタ間接形式関係の要点を挙げるにとどめよう。

●レジスタの直接と間接の違い

アドレスレジスタ直接形式はアドレスレジスタの中身自体を対象にする。対して，アドレスレジスタ間接形式は〝アドレスレジスタが指す(ポイントする)メモリ〟を操作対象にする。

COLUMN

アドレスレジスタ直接形式の落とし穴

68000はデータとアドレスを明確に分離して扱っており,同じレジスタ直接アドレッシングでもデータレジスタ直接形式とアドレスレジスタ直接形式の扱いはかなり違う。第1に,アドレスレジスタ直接形式では,バイトサイズのオペレーションが許されていない。たとえば,

```
move.b  a0,d0
adda.b  #4,a0
```

などはないわけだ。これはまだいい。問題なのはデスティネーションオペランドにアドレスレジスタ直接形式,サイズにワードを指定した

```
movea.w d0,a0
adda.w  #123,a0
```

のような場合だ。この場合は,データ転送や演算に先立ってソースオペランドは32ビットへ符号拡張され,結果のアドレスレジスタは全32ビットが影響を受けることになっている。d0.l, a0.lにそれぞれ0008FFFF$_H$が,d1.lに1が入っているときに,

```
add.w   d1,d0
adda.w  d1,a0
```

を実行すると,d0＝80000$_H$,a0＝90000$_H$になるし,同じ条件で,

```
move.w  d1,d0
movea.w d1,a0
```

を実行すれば,d0＝80001$_H$,a0＝1$_H$という結果になる。

●ポストインクリメント/プリデクリメントアドレスレジスタ間接形式とオペレーションサイズの関係

ポストインクリメントアドレスレジスタ間接形式はアドレスレジスタが指すメモリを操作した後でアドレスレジスタの値を増やし,プリデクリメント～のほうはメモリアクセスに先立ってアドレスレジスタの値が減じられ,減らした後のアドレスレジスタによって指されるメモリが操作対象となる。ここで,アドレスレジスタが増減される量は演算のサイズに等しく,バイトのときは1,ワードのときは2,ロングワードのときは4バイトだった。

●ディスプレースメント付きの意味

ディスプレースメント付きアドレスレジスタ間接形式では,〝アドレスレジスタにディスプレースメントを加えたアドレスで指定されるメモリ〟が操作対象になる。ディスプレースメントは16ビットの符号付き数として扱われるから,アドレスレジスタが指す位置の前後－32768～＋32767のメモリを指定できることになる。ディスプレースメントをアドレスレジスタに加算して実効アドレスを求めるときには,自動的に符号拡張が行われるのを忘れてはならない。a0にE0000$_H$が入っているときに,

```
move.w  $8000(a0),d0
```

で参照されるのは,a0に00008000$_H$を加えたE8000$_H$番地ではなく,FFFF8000$_H$を加えた(＝

00008000Hを引いた)D8000H番地となる。

●どんなときに，どのアドレッシングモードを使うか

ポストインクリメント/プリデクリメントアドレスレジスタ間接形式は，スタックの操作や，配列状に並べられたデータを次々に操作していくのに重宝する。また，ディスプレースメント付き～は，スタックフレームのアクセスや構造を持ったデータを扱うのに便利だ。プログラミングのうえでは，ディスプレースメントは数値で直接指定するよりもequや.offsetなどにより記号定数に定義して使うほうがわかりやすいという話もした。

どのみち，基本であるアドレスレジスタ間接形式だけを知っていれば，他の長ったらしい名前のアドレッシングモードを使わずとも同等の処理を行うことはできる。たとえば，

```
move.l  (a0)+,d0
```

に相当する処理は，

```
move.l  (a0),d0
addq.l  #4,a0
```

の2命令で，また，

```
move.l  8(a0),d0
```

は，

```
addq.l  #8,a0
move.l  (a0),d0
(subq.l #8,a0     ＊必要に応じてa0の値を元に戻す)
```

で実現される。こういう理屈さえわかっていれば，後はどうにでもなるものだ。

と，あらかじめ予防線を張ったところで，アドレスレジスタ間接アドレッシングの最後のバリエーション，〝インデックス付きアドレスレジスタ間接形式〟を紹介してしまおう。

このアドレッシングモードは，

```
move.b  2(a0,d0),d1
```

のようにして使用し，一般形は，

```
disp8(an,index)
```

となる。〝アドレスレジスタにインデックスと8ビットディスプレースメントを加えたアドレス〟を指定するという，なかなか凝ったアドレッシングモードだ。

インデックスには任意のデータレジスタおよびアドレスレジスタが利用でき，しかもインデックスのロングワードすべてを使用するか，下位ワードだけを使用するかを指定することができる。インデックスのサイズがワードかロングワードかの指定は，

```
move.b  2(a0,d0.w)
move.b  2(a0,d0.l)
```

のように，後ろに〝.w〟か〝.l〟をつけて表す。

```
move.b  2(a0,d0)
```

のように省略するとワードと見なされる。また，ワードサイズのインデックスはアドレス計算

時に32ビットに符号拡張される。

ディスプレースメントは8ビットの符号付き数として扱われる。そのため，範囲は−128〜＋127とちょっと狭めだ。これもアドレス計算時には32ビットに符号拡張してから加算される。

インデックス付きアドレスレジスタ間接形式は，多少複雑なデータ構造(Cでいうところの構造体の配列とか)を扱うときにはなかなかの威力を発揮する。たとえば，

```
        .offset 0
NEXT:   .ds.l   1
PREV:   .ds.l   1
DATA1:  .ds.l   1
DATA2:  .ds.l   1
```

のような16バイトでひとまとまりのブロックが，あるアドレス(かりにARRAYTOPとする)以降のメモリ領域にいくつも並んでいる場合を考えよう。n番目(0から数えて)のブロックの先頭アドレスは，

ARRAYTOP＋16×n

で得られるから，そのブロック内の各項目のアドレスは，

ARRAYTOP＋16×n＋NEXT
ARRAYTOP＋16×n＋PREV
ARRAYTOP＋16×n＋DATA1
ARRAYTOP＋16×n＋DATA2

という，〝基準となるアドレス〟＋〝基準からブロック先頭までのずれ〟＋〝ブロック先頭から各項目までの小さなずれ〟の形で表現できるだろう。「インデックス付きアドレスレジスタ間接形式を使え」といわんばかりの御膳立てが整っている。

a0 ← ARRAYTOP

のように下ごしらえしておいて，後は必要なときに，

d0 ← 16×n
d1 ← 16×m

というぐあいに適当なレジスタをインデックスにして，

```
move.l  DATA1(a0,d0), DATA2(a0,d1)
```

によってデータを参照すればよいわけだ。

アドレッシングの基準としてのプログラムカウンタ

2章でアドレスの概念を説明した際，任意のメモリは80000$_H$番地というような絶対的なアドレスで指定する以外に，ある基準からの相対的な差でも表現できるという話をした。ディスプレースメント付きアドレスレジスタ間接形式などはアドレスレジスタを基準にして相対的にア

ドレスを指定するアドレッシングモードといえる。

アドレスレジスタ以外にアドレスを保持するレジスタとしては，プログラムカウンタ(pc)がある。pcはプログラムの実行にともなってどんどん勝手に更新されていくとはいえ，ある命令を実行しようとした瞬間には，まちがいなくその命令の置かれた位置自体をポイントしている。それならば，〝実行しようとしている命令が置かれたアドレスから4バイト先〟というような形で特定のアドレスを指定するのにpcを利用できるのではないかという発想が沸き上がってくる。「pc相対アドレッシング」という考え方だ。

68000のpc相対アドレッシングには〝ディスプレースメント付きpc相対形式〟と，〝インデックス付きpc相対形式〟の2種類がある。それぞれ，

```
move.w  10(pc),d0
move.w  8(pc,d1),d0
```

のようにして使い，一般形は，

```
disp16(pc)
disp8(pc,index)
```

となる。見てのとおり，ディスプレースメント付き/インデックス付きアドレスレジスタ間接形式と同じような格好をしている。実際，ディスプレースメントの範囲なども同じだ。

しかし，アドレスレジスタ間接形式が主として動的なデータのアクセスに用いられるのに対して，pc相対形式はその性格上，静的な(プログラムの実行中，つねに同じアドレスに存在しつづける)データ/プログラムの参照にのみ使われる。用途の点では，むしろ絶対アドレス形式に近い。このあたりの事情はアセンブラもよくわかっていて，pc相対アドレッシングでは，

```
        move.w  LABEL(pc),d0
        move.w  LABEL(pc,d1),d0
          :
LABEL:  .dc.w   123
```

のように，ディスプレースメントの部分に〝アクセスしたいアドレス自体を表すラベル〟を使うことが許されている。こう書いておくと，命令の置かれたアドレスとアクセスするアドレスの差をアセンブラが勝手に計算してくれるのだ[1)]。

■ 1)細かな話になるが，68000のpc相対アドレッシングでアドレス計算の基準となるpcは，命令が置かれた先頭アドレスから2バイトだけずれている。これは68000の命令実行サイクルが，
1) pcの位置から1ワード読み込む
2) pcを1ワード分＝2バイト進める
3) 命令を解釈し，実行する
という順序で行われるためだ。実効アドレスを計算するのは3)の時点だから，pcはすでに2バイトさきに進んでしまっているわけだ。
もっとも，ディスプレースメントの計算をアセンブラにまかせてしまえば，こんなことは知らなくても困らないが。

さて，pc相対形式の絶対アドレス形式に対する優位点としては，

1) プログラムを短くできる[2]。

2) プログラムを高速にできる[3]。

3) プログラムをリロケータブルにできる[4]。

といった点が挙げられる。とくに，リロケータブルなプログラムを作るうえではpc相対アドレッシングが不可欠であり，これこそがpc相対アドレッシングの存在理由でもある。

■2)必ずしも一般的にそうだというわけではない。68000の場合はディスプレースメント付きpc相対形式ではディスプレースメントが16ビットに“制限されているから”32ビットの絶対アドレスで指定するより1ワード分コードが短くなるという話であって，(実際にあるかどうかは知らないが)絶対アドレスと同じだけのビット幅でディスプレースメントを指定できるようなプロセッサがあればその差はないわけだ。

■3)これも68000では，の話。プロセッサによってはpc相対アドレッシングを使うとアドレス計算に時間がかかるのか，かえって遅くなる場合もある。

■4)リロケータブル(relocatable:直訳すれば再配置可能)というのは，メモリ上のプログラムをごっそり別のアドレスに移動しても手を加えることなく動く，つまり，どのアドレスにロードしてもそのまま動作するという意味だ。Human68kのXファイルもロードするアドレスを選ばないが，これはHuman68kがロード時にロードアドレスに応じてプログラムを書き換えている(そのための情報はXファイルが持っている)からであって，純粋な意味でのリロケータブルとは区別し，「ソフトウェアリロケータブル」という言葉で表現される。

ところが，残念ながら68000のpc相対アドレッシングにはちょっとした制限がある。というのも，デスティネーションには利用することができないのだ。

```
move.w  LABEL(pc),d0
```

はできても，

```
move.w  d0,LABEL(pc)
```

は許されない。これは，68000の設計思想にかかわる問題だったりする。

68000は，完全ではないにしろ，プログラムとデータの分離という考えを打ち出していて，詳しい話はしないが，メモリ空間などにもそういった思想が見える。その思想にしたがい，

1) pcが指しているのはプログラムが置かれた領域であり，プログラムカウンタ相対でアクセスされるのはそのプログラムが置かれた領域である。

2) プログラムは本来実行中に書き換えるべきものではない。

3) ゆえに，pc相対で指定されるアドレスが書き換えられるはずがない。

という見事な三段論法によって，pc相対アドレッシングではメモリの参照だけができ，書き換えができないように作られている。こういったお上品さが68000らしさでもあるわけだ。

COLUMN

リロケータブルなプログラム

端的な例を挙げよう。次のような一命令からなるプログラムが80000_H番地に置かれているとする。

```
jmp     $80000
```

えんえんと無限ループを続けるプログラムというわけだ。明らかにこのプログラムは，80000_H番地に置かれた場合にのみ正常動作する。80000_Hという絶対アドレスで分岐先を指定している以上，当然のことだ。

では，次のプログラムはどうだろう。

```
bra     -2(pc)
```

やはり無限ループを続けるプログラムだが，今度は絶対アドレスを使わずにpc相対で分岐先を指定している。そのため，このプログラムはどのアドレスに置いてもきちんと無限ループしてくれる。プログラムがどこに置かれようと，相対的な位置関係は変わらないわけだ。

もうネタは割れたと思うが，要するに，プログラムをリロケータブルにするには，とにかく絶対アドレスを使わなければよい。その場合，分岐先やデータアクセス時のアドレス指定にはひたすらpc相対を使う。

ただ,68000はあまりリロケータブルなプログラムを作るのに適しているとはいえない。本文でもふれたように,68000のpc相対アドレッシングにはデスティネーションオペランドには適用できないという制限がある。リロケータブルなプログラムを作ろうと思ったら，この制限をうまく回避する必要がある。具体的には，

```
move.w  d0,work
```

は，

```
lea.l       work(pc),a5
move.w      d0,(a5)
```

のような2命令に展開することになる。まず，pc相対で操作対象のメモリアドレスをアドレスレジスタに入れておいてから，アドレスレジスタ間接アドレッシングを利用するわけだ。

メモリに書き込むたびにこんなことをするのは冗長なので，現実のプログラムでは最初に1度だけアドレスレジスタを初期化して，以降のメモリアクセスはそのアドレスレジスタを通じて行うようにすることが多い。ワークが複数ある場合にもワークエリアがひとまとめになっていれば，ディスプレースメント付きアドレスレジスタ間接形式が利用できるので，アドレスレジスタは1本しかつぶさないですむ。ワークエリアが，

```
work1:      .ds.l   1
work2:      .ds.w   1
```

のような構成だったとすると，

```
lea.l       work1(pc),a5
```

と初期化しておけば，

```
move.l      d0,(a5)
```

によってwork1にアクセスできるだけでなく，

```
move.w      d1,4(a5)
```

によってwork2へも書き込みが行えるわけだ。

さて，Human68kの実行ファイルには，通常使われているソフトウェアリロケータブルなX形式以外に，完全リロケータブルなR形式，特定のアドレスに置かれたときにしか動作しないZ形式がある。そして，上記のような手法で絶対アドレス参照を行わずにリロケータブルに組まれたプログラムは，アセンブル・リンク後に，生成されたXファイルをコンバー

タCV.Xに通すことでR形式に変換することができる。R形式のファイルはX形式のように再配置情報をファイルに含まなくてすむため，若干ファイルサイズが小さくなるという利点がある。もっとも，それ以外にはたいしたメリットはない。公開ソフト等でときおりR形式のプログラムが見受けられるのは，実行速度を稼ぐ意味で絶対アドレス形式を使うのを避けた副作用でリロケータブルに〝なってしまった〟だけなのだ(後は美意識や趣味の問題だろう)。

では，最後にCV.XでR形式にコンバートする際の注意点を挙げてこの項を終わろう。

●R形式に変換するプログラムには，ブロックストレージセクションとスタックセクションがあってはならない。

これら2つのセクションは，Xファイルのヘッダ部分でその大きさ等が指定されるので，余分な情報を持たないRファイルでは利用できない。スタック領域や大きなバッファが必要であれば，プログラム側でmallocやsetblockなどを利用してメモリを確保して使う必要がある。

●.end疑似命令で実行開始アドレスを指定しても無視される。

.end疑似命令は後ろにラベルを置くことでプログラムの実行開始アドレスを指定することができる(この機能は2章あたりからこっそり使っている)のだが，この情報もRファイルでは削られてしまう。したがって，複数のオブジェクトをリンクして実行ファイルを作る場合などは，必ずメインルーチンがファイルの先頭にくるように，リンク順序にも注意を払わなければならない。

COLUMN

テーブル参照

前章では多方面への分岐をすっきり行う方法としてジャンプテーブルを利用し，そのもっともオーソドックスな形としてリストAのようなコーディングをしてみせた。また，ほとんど同じ手法で，〝文字列へのポインタのテーブル〟を参照する例も示した。テーブルの使用はプログラムをすっきりさせ，メモリ効率や実行速度を稼ぐ手段として有用なものだ。ここでは，テーブル参照の例をいくつか挙げてみたい。

リストA

```
 1:          moveq.l  #0,d0
 2:          move.b   処理番号,d0
 3:          add.w    d0,d0
 4:          add.w    d0,d0
 5:          lea.l    jmptbl,a0
 6:          adda.l   d0,a0
 7:          movea.l  (a0),a0
 8:          jsr      (a0)
 9:                   :
10: jmptbl:
11:          .dc.l    test0
12:          .dc.l    test1
13:          .dc.l    test2
14:                   :
```

リストBはpc相対アドレッシングを利用したジャンプテーブルの例だ。テーブルには分岐先アドレスそのものではなく，〝分岐先アドレスとテーブル先頭の差〟を並べてある。リ

ロケータブルなプログラムでジャンプテーブルを利用したい場合にはこのような方法も使う。

リストB

```
 1:          moveq.l  #0,d0
 2:          move.b   処理番号,d0
 3:          add.w    d0,d0
 4:          add.w    d0,d0
 5:          move.l   jmptbl(pc,d0.w),d0
 6:          jsr      jmptbl(pc,d0.l)
 7:                   :
 8: jmptbl:
 9:          .dc.l    test0-jmptbl
10:          .dc.l    test1-jmptbl
11:          .dc.l    test2-jmptbl
12:                   :
```

リストCは分岐できる距離に制限をつけて，リストBのメモリ効率，実行速度を向上させたものだ。テーブルがロングワードではなく，ワードになっている。

リストC

```
 1:          moveq.l  #0,d0
 2:          move.b   処理番号,d0
 3:          add.w    d0,d0
 4:          move.w   jmptbl(pc,d0.w),d0
 5:          jsr      jmptbl(pc,d0.w)
 6:                   :
 7: jmptbl:
 8:          .dc.w    test0-jmptbl
 9:          .dc.w    test1-jmptbl
10:          .dc.w    test2-jmptbl
11:                   :
```

リストDはメモリ効率を犠牲にして実行速度を稼ぐテーブルの使用例だ。Cの文字種判別関数相当のサブルーチンとなっている。ある値が英字の文字コードかどうか，数字の文字コードかどうかといった検査をする場合，ふつうなら文字コードの大小関係を比較するわけだが，リストDでは，あらかじめどの文字コードが英字で，どの文字コードが数字かといった情報をテーブルに用意してある。テーブルは1つの文字コードにつき1バイトで，第0ビットが0なら英大文字，第1ビットが0なら英小文字，……，というように決めた(0と1の意味がふつうの感覚の逆になっていることに注意)。英字かどうかを検査するサブルーチンisalphaでは，テーブルの該当位置の第0ビットと第1ビットをbtst命令で調べ，どちらかが0であれば英字であると判断するわけだ。結果はccrのZビットで返され，英字であればZ＝1で戻る。

リストD

```
 1: *        文字種検査サブルーチン集   ctype.s
 2:
 3:          .xdef    isalpha
 4:          .xdef    isalnum
 5:          .xdef    isupper
 6:          .xdef    islower
 7:          .xdef    isdigit
 8:          .xdef    isxdigit
 9: *
```

```
10: UPPER_BIT       equ     0
11: LOWER_BIT       equ     1
12: DIGIT_BIT       equ     2
13: XDIGIT_BIT      equ     3
14: *
15:         .text
16:         .even
17: *
18: *       英字かどうかの検査
19: *
20: isalpha:
21:         btst.b  #UPPER_BIT,ctypetbl(pc,d0.w)
22:         beq     retn
23:         btst.b  #LOWER_BIT,ctypetbl(pc,d0.w)
24: retn:   rts
25: *
26: *       英数字かどうかの検査
27: *
28: isalnum:
29:         btst.b  #DIGIT_BIT,ctypetbl(pc,d0.w)
30:         beq     retn
31:         btst.b  #UPPER_BIT,ctypetbl(pc,d0.w)
32:         beq     retn
33:         btst.b  #LOWER_BIT,ctypetbl(pc,d0.w)
34:         rts
35: *
36: *       英大文字かどうかの検査
37: *
38: isupper:
39:         btst.b  #UPPER_BIT,ctypetbl(pc,d0.w)
40:         rts
41: *
42: *       英小文字かどうかの検査
43: *
44: islower:
45:         btst.b  #LOWER_BIT,ctypetbl(pc,d0.w)
46:         rts
47: *
48: *       数字かどうかの検査
49: *
50: isdigit:
51:         btst.b  #DIGIT_BIT,ctypetbl(pc,d0.w)
52:         rts
53: *
54: *       16進数を意味する文字かどうかの検査
55: *
56: isxdigit:
57:         btst.b  #XDIGIT_BIT,ctypetbl(pc,d0.w)
58:         rts
59: *
60: ctypetbl:
61:         .dc.b   %11111111,%11111111,%11111111,%11111111 *00-03
62:         .dc.b   %11111111,%11111111,%11111111,%11111111 *04-07
63:         .dc.b   %11111111,%11111111,%11111111,%11111111 *08-0b
64:         .dc.b   %11111111,%11111111,%11111111,%11111111 *0c-0f
65:         .dc.b   %11111111,%11111111,%11111111,%11111111 *10-13
66:         .dc.b   %11111111,%11111111,%11111111,%11111111 *14-17
```

```
67:         .dc.b   %11111111,%11111111,%11111111,%11111111 *18-1b
68:         .dc.b   %11111111,%11111111,%11111111,%11111111 *1c-1f
69:         .dc.b   %11111111,%11111111,%11111111,%11111111 *20-23
70:         .dc.b   %11111111,%11111111,%11111111,%11111111 *24-27
71:         .dc.b   %11111111,%11111111,%11111111,%11111111 *28-2b
72:         .dc.b   %11111111,%11111111,%11111111,%11111111 *2c-2f
73:         .dc.b   %11110011,%11110011,%11110011,%11110011 *30-33
74:         .dc.b   %11110011,%11110011,%11110011,%11110011 *34-37
75:         .dc.b   %11110011,%11110011,%11111111,%11111111 *38-3b
76:         .dc.b   %11111111,%11111111,%11111111,%11111111 *3c-3f
77:         .dc.b   %11111111,%11110110,%11110110,%11110110 *40-43
78:         .dc.b   %11110110,%11110110,%11110110,%11111110 *44-47
79:         .dc.b   %11111110,%11111110,%11111110,%11111110 *48-4b
80:         .dc.b   %11111110,%11111110,%11111110,%11111110 *4c-4f
81:         .dc.b   %11111110,%11111110,%11111110,%11111110 *50-53
82:         .dc.b   %11111110,%11111110,%11111110,%11111110 *54-57
83:         .dc.b   %11111110,%11111110,%11111110,%11111111 *58-5b
84:         .dc.b   %11111111,%11111111,%11111111,%11111111 *5c-5f
85:         .dc.b   %11111111,%11110101,%11110101,%11110101 *60-63
86:         .dc.b   %11110101,%11110101,%11110101,%11111101 *64-67
87:         .dc.b   %11111101,%11111101,%11111101,%11111101 *68-6b
88:         .dc.b   %11111101,%11111101,%11111101,%11111101 *6c-6f
89:         .dc.b   %11111101,%11111101,%11111101,%11111101 *70-73
90:         .dc.b   %11111101,%11111101,%11111101,%11111101 *74-77
91:         .dc.b   %11111101,%11111101,%11111101,%11111111 *78-7b
92:         .dc.b   %11111111,%11111111,%11111111,%11111111 *7c-7f
93:
94:         .end
```

絶対分岐と相対分岐

じつをいうと，ディスプレースメント付きpc相対形式は初登場ではなく，すでに第2章から毎章必ず使っている。bra，bsr，dbraはアセンブリ言語の書式上は，

```
bra     LABEL
bsr     sub
dbra.w  d0, sub
```

というように絶対アドレス形式のような書き方をするが，本当はpc相対で分岐する命令だ。

```
bra     LABEL(pc)
bsr     sub(pc)
dbra.w  d0, sub(pc)
```

のように，後ろに(pc)が隠れていると思えばいい[5]。分岐先とのアドレスの差は，例によってアセンブラが自動的に計算してくれている。

また，bra, bsrにはディスプレースメントを8ビットに制限し命令長を1ワードに押さえる形式と，ディスプレースメントが16ビットで命令長が2ワードになる形式の2種類がある[6]。

■5)ただし，実際に，
```
bra     LABEL(pc)
```
というような指定をすると，アセンブル時にエラーになる(世の中には許してくれるアセンブラもあるのかもしれないが)。
■6)dbraの場合はつねに16ビットディスプレースメントが使用される。

AS.Xは，ディスプレースメントが8ビットの符号付き数で収まれば自動的に命令コードが短くなるように最適化してくれる。この最適化は該当するものがなくなるまで繰り返し行われるので，場合によってはアセンブルに必要以上の時間がかかることもある[7]。多少プログラムが大きくなってもかまわないのであれば，/Nスイッチを指定することでこの最適化を行わないようにもできるが，別の手としてはソースで明確にディスプレースメントのサイズを指定する方法がある。moveなどの命令でサイズを指定したときのように，

```
bra.b   LABEL
```

とか，

```
bra.w   LABEL
```

のように記述すればよい。とくに，

```
bra.b   LABEL
```

は，

```
bra.s   LABEL
```

というように〝.s(Short)〟という表記をすることが許されており，たいていはこっちを使う。

■7)わかりにくい表現だが，こういうことだ。分岐先のアドレスが決定されていないときには，アセンブラはとりあえず16ビットディスプレースメントと仮定してアセンブルする。その後，実際には8ビットディスプレースメントで届くということがわかれば，置き換える。すると，以下の命令がごっそり1ワード前にずれることになり，さきほどまでは8ビットディスプレースメントでは届かなかったのが届くようになる可能性が出てくる。それを調べるためには，もう1度ソースを最初までさかのぼって見直さなければならない，というわけだ。

braやbsrの真の姿を暴露したところで，ついでに絶対アドレスで任意のアドレスに飛べる分岐命令，サブルーチン呼び出し命令を紹介しておく。braに対応する無条件分岐命令はjmp(JuMP)，bsrに対応するサブルーチン呼び出しのほうは前章でも顔を出したjsr(Jump to SubRoutine)だ。どちらも絶対アドレスで分岐先を指定できるだけでなく，

```
jmp     (a0)
jsr     jmptbl(pc,d0.l)
```

のように，68000の他のアドレッシングモードを利用することができる。そのかわり，jmpは無条件ジャンプ専用で，条件分岐するためにはbcsなどと組み合わせて実現するしかない。近場に条件付き相対分岐して，そこからjmpで高飛びするわけだ。

クロックと実行速度

つづいては，マシン語命令の実行速度の話。これまでにも，0との比較にはcmpiを使うよりtstのほうが速いとか,ふつうのイミディエイト形式よりもクイック・イミディエイト形式のほうが名前からして速いとか，いくつか具体的な例は挙げてきたが，ここでその裏付けをしておこうと思う。

68000の命令の実行速度は，命令の種類，サイズ，アドレッシングモードによって変化するがだいたいの目安がある。順不同で並べてみると，

1) オペレーションサイズはロングワードよりワードのほうが速いが，ワードとバイトは同じ速度。
2) 乗除算のように感覚的に動作が複雑そうな命令は遅い。
3) (プロセッサ内部にある)レジスタに対するアクセスは(プロセッサ外部にある)メモリに対するアクセスより速い。
4) アドレッシングモードは複雑になるにつれて遅くなる。が，単純そうな絶対ロング形式がいちばん遅い。
5) メモリに何回もアクセスするほど遅くなる。
6) オペランドを含む命令コード全体の長さが長いほど遅い。
7) moveに対するclr, cmpiに対するtstのように，一部の特定条件でのみ使用できる専用命令は速い(少なくとも遅くはない)。
8) クイックイミディエイト形式はふつうのイミディエイト形式より速い(そのかわり，指定できる範囲に制限がある)。

上に挙げた条件は部分的には同じことをいっている。6)の命令コードが長いほど遅いというのは，その分メモリに多くアクセスするし(pcの指す位置からの命令取り込みもメモリアクセスだ)，命令コードが長いということはきっとアドレッシングモードも複雑だろうし，複雑なアドレッシングモードというからにはデータレジスタ直接などではなくてメモリが対象だろうから，またメモリアクセスがあるだろうし，といくつもの条件が重なっていると考えられる。

また，7)のclrやtstはオペランドがmoveやcmpiの2個に対して1個ですむ分，8)のクイックイミディエイト形式は1ワードの命令の中にオペランドを包含している分，それぞれ命令コードの全体長が短くなって〜，というぐあいだ[8]。

■8)clrでメモリを対象にする場合，68000は一度メモリ内容を読み出し，それからあらためて0を書き込む。この余分なメモリアクセス(1回)のために，結局はmoveで0を書き込むのと同じ程度の速度になってしまっている。

より正確な命令の実行時間を知りたければ，まともな命令表に載っている各命令のクロックサイクル数を調べてみるとよい。命令のクロックサイクル数というのは，プロセッサに与えら

れるクロック[9]のパルス(┌─┐)何個分の時間で実行が終了するかを表し，小さければ小さいほど速い命令であることを意味している。ハードウェアに密着したプログラムでは，ある処理を行ったら何ms待たなければならないとか，処理を何ms以内に完了しなければならない，などということもあり，そういう場合はクロックサイクル数にクロック周波数から計算した1クロックあたりの時間をかけて命令の実行に必要な時間を求めたりもする[10]。そこまでいかなくても，命令表を見ながら各命令の実行時間の差の原因を考えるのはなかなか楽しい遊び(?)だし，同じ動作をする2つの命令があった場合，どちらが速いかを調べてみるのも有意義だろう。

■9) clock：コンピュータの動作基準となる信号。
■10) X68000のクロック周波数は10MHzだから，1クロックは100万分の1秒＝0.1ms＝100nsとなる。

表1にこうして調べた常套句をいくつか挙げておく。表の最後に示した2つのパターンでは，絶対ショートアドレス形式というマイナーなアドレッシングモードを利用している。このアドレッシングモードは16ビットで表現できる範囲の絶対アドレスを指定するもので，実効アドレスを求める際には32ビットに符号拡張される[11]。

0の代入	move.b #0,d0 → clr.b d0 move.w #0,d0 → clr.w d0 move.l #0,d0 → moveq.l #0,d0 movea.l #0,a0 → suba.l a0,a0
0との比較	cmpi.w #0,d0 → tst.w d0
アドレスレジスタに対する定数の加減算	adda.l #1,a0 → addq.l #1,a0 suba.l #8,a0 → subq.l #8,a0
スタックへのプッシュ	adda.l #99,a0 → lea.l 99(a0),a0 move.l #99,-(sp) → pea.l 99.w clr.l -(sp) → pea.l 0.w

表1
より高速な命令への置き換え例

しかし，AS.XのVer.1.0では絶対ショートアドレス形式がサポートされていないため，このままの形で使っても意味がない[12]。どうしてもこのパターンを使いたければ，

```
PEAW    macro   data
        .dc.w   $4878,data
        endm
```

とマクロ定義しておき，

```
pea.l   10.w
```

のかわりに，

```
PEAW    10
```

のようにして使うといった工夫が必要になってくる。ここで4878_Hというコードは，

```
lea.l   XXXX.w
```

のマシンコードで[13]，これを.dc.wで直接プログラムに埋め込んでいるわけだ。

■11)これにより，絶対ショートアドレス形式で指定可能な範囲は，
00000000H～00007FFFH
FFFF8000H～FFFFFFFFH
というメモリの最下位32Kバイトと最上位32Kバイトに制限される。なお，68000のアドレスは下位の24ビットのみが有効であり，上位8ビットは無視されるから実質的には，
00000000H～00007FFFH
00FF8000H～00FFFFFFH
となる。

■12)AS.X Ver.1.0では，絶対ショートアドレス形式を指定しても絶対ロングアドレス形式と見なされてしまうため，かえってmoveを使うより遅くなってしまう。

■13)このコードは，デバッガDB.XのAコマンドで1行アセンブルして調べた。デバッガではVer.1.0からちゃんと絶対ショートアドレス形式がサポートされている。

DOSコールの秘密

最終章にしてようやく，DOSコールの仕組みが解き明かされる。どうして，

```
DOS     _EXIT
```

と書くとDOSコールが呼び出せるのかという話をしよう。

周知のことと思うが，〝DOS〟はDOSCALL.MACの中で，

```
DOS     macro   callno
        .dc.w   callno
        endm
```

のように定義されたマクロであり，また，〝EXIT〟はやはりDOSCALL.MAC内で，

```
_EXIT   equ     $ff00
```

と定義された記号定数だ。結局，

```
DOS     _EXIT
```

はアセンブル時にアセンブラによって，

```
.dc.w   $ff00
```

に展開される。さらに〝.dc.w〟は，オペランドに並べられたワードデータをそのままオブジェクトに吐き出すことをアセンブラに指示する疑似命令だから，この行は最終的にFF00Hというコードに変換され，オブジェクトプログラムに埋め込まれる。exit以外のDOSコールの場合も同様で，getcharならFF01H，putcharならFF02Hというようにコール番号そのままの形でプログラムに埋め込まれる。ここまではいいだろう。

さて，マシン語プログラムの実行は，pcを順次進めつつ，そのpcがポイントするメモリからビット列を取り出しては(それが命令だと仮定して)解釈・実行することで行われるのだった。特定のビット列は，ある動作と1対1で対応しており，概念的にはプロセッサはメモリから拾

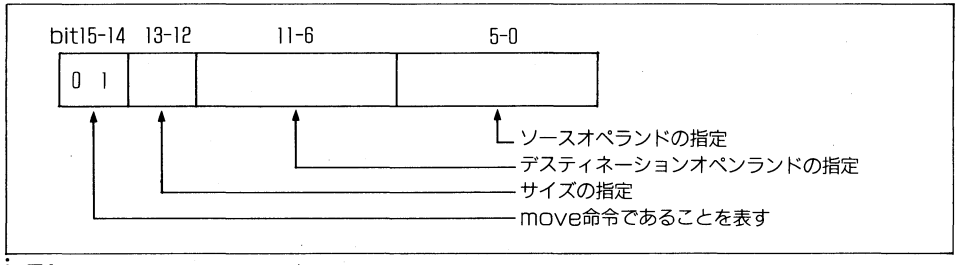


図1
move命令のマシン
コードフォーマット(概略)

ったビット列と内部に持った命令表とを照らし合わせて該当する動作を調べることになる。

実際にはマシンコードは0000_Hならこんな動作,0001_Hならこんな動作と個別に決められているわけではなく，解釈しやすいように，ある程度の規則性を持っている(参考までに，図1にmove命令のフォーマットを示しておく)。もちろん例外もあって，特殊な命令にはこの秩序から外れたマシンコードが割り振られていることもあるし，命令の数とマシンコードのビット長との兼ね合いから動作が割り当てられていないマシンコードの隙間もかなりある。プロセッサをバージョンアップする過程で命令を新設するときには，この空き部分に新命令を割り振ることになる。それでもまだ足りなければ，マシンコードの構成を2段構えにして回避したりもする。ある特定のコードを頭に置き，続くもう1つと組み合わせて動作を定義するという，ほとんどエスケープシーケンスのノリだ。こうしてなんとかザウルスができ上がる。

それはともかく，現在のコンピュータでは，pcの指すものはとにかく命令と見なすという上述の方式が主流中の主流になっている。いわゆる暴走というヤツの大部分はなにかのはずみでpcがあらぬところを指し，偶然そこに格納されているもの(プログラムの一部かもしれないし，データかもしれないし，メモリ上に残ったたんなるゴミかもしれない)を強引に実行しようとして予想外の結果を引き起こしてしまった状態なわけだ。

ここで，〝強引に実行しようとする〟のにも2通りが考えられる。pcの指すものが命令として解釈できる場合と，できない(動作が定義されていない)場合だ。前者であれば，〝pcの指すものは命令である〟の掟にしたがい，プロセッサは平気な顔で実行を続ける。その結果，変なメモリを書き換えたりして2次被害が発生する。ただし，68000の場合は，変なメモリを書き換えようとしたときにバスエラーやアドレスエラーで止まってくれる可能性が高い。また，後者の定義されていない命令を実行しようとした場合の動作はプロセッサによって(可能性としてはチップ1個1個ごとに)異なり，たんに無視するだけかもしれないし，なにか突拍子もない動作をするかもしれないし，「カイドクフノウ」とカタカナで喋って煙を吹くかもしれない。68000ではどうなるかというと，ちゃんと未定義命令を識別し，エラーを出してくれる。賢い。

さて，DOSコールの呼び出しに使われている$FFxx_H$というコードに話を戻そう。結論からいえば，68000では$FFxx_H$はおろか，上位4ビットが1111_Bのコード($Fxxx_H$)にはいっさい命令が割り振られていない。当然，pcがこのコードを指すようなことがあれば，68000は知らない命

令があるといって怒り出す。〝怒る〟なんていいかげんな表現ではなく，正しい言葉を使えば，〝例外処理〟を始める。

68000ではバスエラーやアドレスエラー，そして動作の定義されていない命令の検出などを含めて〝例外(exception)〟と呼び，なんらかの例外が発生すると，その時点で実行中だった処理を中断して，ある決まった手順で例外処理ルーチンに制御を移す。例外処理ルーチンは例外の種類ごとに用意される。で，X68000＋Human68kでは，バスエラー例外やアドレスエラー例外などのときには例のエラーメッセージを表示するルーチンが，$Fxxx_H$のコードが実行されそうになったときにはDOSコールの処理ルーチンが，例外が発生するのを待ち構えているわけだ。

例外が発生したときに制御が移るアドレスは，メモリの000_H～$3FF_H$番地に4バイトずつのテーブルとして用意されている。この4バイトごとのメモリを〝例外ベクタ〟と呼び，例外ベクタには00_H～FF_Hのベクタ番号が振られている。バスエラー例外のベクタ番号は2，アドレスエラー例外なら3というように，例外とベクタ番号は1対1に対応している。DOSコールの呼び出し時に発生するのは，ベクタ番号11に割り振られた〝1111系列未実装命令[14]の実行による例外〟だ。ベクタアドレスは$02C_H$番地となる。

■14)上位4ビットが1111_Bだから，こう呼ぶ。同様に68000では上位4ビット1010_Bのコード($Axxx_H$)も未実装になっており，こちらにはベクタ番号10が割り当てられている。1010系列はHuman68kではずっと空きになっていたが，SX-WINDOWのシステムコール用に使われるようになった。

さて，未実装命令の検出によって例外が発生すると，68000は自動的にスーパーバイザモードに移行し[15]，スタックに，

```
(sp) = sr
2(sp) = pc
```

となるように例外が発生した時点でのpcとsrを待避する。すでにスーパーバイザモードになっているから，このspはスーパーバイザスタックポインタ(ssp)だ。その後，ベクタ番号11の内容がpcに取り込まれ，DOSコールの処理が始まる。

■15)srの第13ビットが1ならスーパーバイザモード，0ならユーザーモードを表しており，このビットを操作することでスーパーバイザモード←→ユーザーモードの切り替えが行える。ただし，srの上位バイト(第7～15ビット)はスーパーバイザモードでなければ書き換えることができない。ユーザーモードからスーパーバイザモードへの切り替えは〝なんらかの例外を発生させる〟ことで行う。

ここでは，まず，DOSコール番号を調べる。未実装命令の実行による例外時には，スタックに待避されたpcは例外を引き起こした命令コード自身を指すことになっているから，たとえば，

```
movea.l  2(sp),a0
move.w   (a0)+,d0
```

```
move.l    a0,2(sp)
```

によってd0.wにDOSコール番号が取り出される。a0をポストインクリメントしてスタックに戻すことで，スタック上には，

(sp) ＝ DOSコール呼び出し前のsr

2(sp) ＝ 戻りアドレス

が保存された形となる。後でユーザープログラムに戻るときには，スタックからsrとpcを同時に取り出すrte(Return from Exception)という命令[16]を使い，srを復帰しつつ(これによりユーザーモードに移行することができる)，DOSコール呼び出し直後から処理を再開する。

■16) rteは“特権命令”であり，スーパーバイザモードでなければ実行できない。ユーザーモードで使うとエラー(特権違反による例外)が発生する。

DOSコール番号を取り出したら，上位バイトがまちがいなくFF_Hかどうかを確認する[17]。違っていたらエラー処理に即飛んでしまう。上位バイトがFF_Hであることが確認されたら，スタックに積まれているはずの引数の先頭アドレスをa6に入れたうえで，下位バイトに応じてジャンプテーブルの手法を使って各DOSコールの処理ルーチンへ分岐する。実際にはもう少し複雑なのだが，だいたいはこんな感じといえるだろう。

■17) $F0xx_H$や$F1xx_H$などの，DOSコールに関係のない未実装命令をはじく必要がある。

なお，FLOATn.Xによってサポートされる演算ファンクションコールは，DOSコールと同じ1111系列の未実装命令である$FExx_H$を使って呼び出される[18]。FLOATn.Xを組み込むと，例外ベクタ番号11の内容がFLOATn.Xの内部を指すように変更されて，その状態で1111系列の未実装命令が実行されそうになると，まず，FLOATn.Xによって$FExx_H$かどうかのチェックが入り，そうでなければ元のDOSコールの処理ルーチンに制御が移される。

■18) XCには“FEFUNC.H”というファイルが含まれており，この中でFLOATn.Xの内部ファンクション番号が定義されている。

ここで関連DOSコールを紹介しておく。

・DOSコール$FF35 intvcg

```
move.w    例外ベクタ番号，-(sp)
DOS       _INTVCS
addq.l    2,sp
```

例外ベクタの内容を読み出すDOSコールだ。例外ベクタ番号を指定すると，そのベクタ内容がd0.lに返ってくる。例外ベクタ番号のかわりにDOSコール番号を指定することもでき，その場合は該当するDOSコールの処理ルーチン先頭アドレス(これもベクタと呼ぶことがある)が返ってくる。

・DOSコール$FF25 intvcs

```
move.l    セットするベクタ内容,-(sp)
```

```
move.w  例外ベクタ番号, -(sp)
DOS     _INTVCS
addq.l  6, sp
```

intvcgの逆で，指定した例外ベクタの内容を変更する。リターン時のd0.lには変更する前のベクタ内容が返ってくる。やはり，DOSコール番号を指定することができ，これによって個々のDOSコールの処理ルーチンを自作のルーチンと差しかえたりすることができる。その例を次のプログラムで示そう。

最後のプログラムASX.X

最後に本章のまとめとして，また，この本のまとめとして実用プログラムを１本示す。AS.X Ver.1.0Xの不備を補い，アセンブルエラーの修正を手助けするプログラムとでもいおうか。AS.Xが出力するエラーメッセージを横取りして，ED.Xのタグファイル形式のエラーファイルを作成するプログラム，ASX.Xだ[19]。リスト１がソース，リスト２がその中で使う定数定義ファイルとなっている。

■19) AS.X Ver.2.0ではエラーメッセージがもともとED.Xのタグファイル形式になっているので，このプログラムは意味をなさない。

リスト1
ASX.S

```
 1: *       AS.X Ver.1.0xのエラーメッセージを横取りして
 2: *               エラーファイルを作成する
 3: *
 4:         .include        doscall.mac
 5:         .include        const.h
 6:         .include        files.h
 7: *
 8:         .xref   memoff
 9:         .xref   child
10: *
11: STRMAX  equ     256             *文字列の最大長
12: BUFSIZ  equ     300             *１行バッファの大きさ
13: *
14:         .text
15:         .even
16: *
17: ent:
18:         lea.l   mysp(pc),sp     *spの初期化
19:
20:         bsr     memoff          *余分なメモリを開放する
21:
22:         bsr     chkarg          *コマンドラインの解析
23:
24:         pea.l   break(pc)       *中断時の戻りアドレスを
25:         move.w  #_CTRLVC,-(sp)  *  セット
26:         DOS     _INTVCS         *
27:         move.w  #_ERRJVC,(sp)   *
```

```
28:         DOS     _INTVCS         *
29:         addq.l  #6,sp           *
30:
31:         bsr     do              *メイン処理
32:
33:         DOS     _WAIT           *AS.Xの終了コードを得て
34:         move.w  d0,-(sp)        *  それをそのまま持って
35:         DOS     _EXIT2          *  終了する
36:
37: *
38: *       メイン処理
39: *
40: do:
41:         bsr     set_vector      *DOSコールのベクタを書き換える
42:
43:         lea.l   cmdlin(pc),a0   *子プロセスを起動するために
44:         lea.l   prog(pc),a1     *  コマンドラインを作成する
45:         bsr     strcpy          *
46:         movea.l a2,a1           *
47:         bsr     strcpy          *
48:
49:         pea.l   cmdlin(pc)      *子プロセス起動
50:         bsr     child           *
51:         addq.l  #4,sp           *
52:         tst.l   d0              *
53:         bmi     error1          *負ならエラー
54:
55:         bsr     reset_vector    *ベクタを元に戻す
56:
57:         rts
58:
59: *
60: *       DOSコールwriteが発動されるとここに来る
61: *
62:         .offset 0
63: *
64: FNO:    .ds.w   1       *ファイルハンドル
65: DATPTR: .ds.l   1       *出力データへのポインタ
66: DATLEN: .ds.l   1       *出力データのバイト数
67: *
68:         .text
69: *
70: write:
71:         cmpi.w  #STDOUT,FNO(a6) *出力先は標準出力か？
72:         bne     write0          *そうでなければ
73:                                 *  オリジナルの処理ルーチンへ
74:
75:         movem.l d1-d2/a0-a3,-(sp)       * {
76:
77:         movea.l bufptr(pc),a0   *a0=バッファ内の次の位置
78:         movea.l DATPTR(a6),a1   *a1=出力データ
79:         move.l  DATLEN(a6),d1   *d1=出力データのバイト数
80:         lea.l   buff+BUFSIZ-1(pc),a2
81:                                 *a2=バッファの上限
82:                                 *（終端コードの分を考慮）
83: wrt0:   move.b  (a1)+,d0        *１バイト取り出し
84:         move.b  d0,(a0)+        *  バッファに転送する
```

```
 85:          cmpi.b  #LF,d0          *１行分溜まったか？
 86:          beq     wrt1            *  そうなら１行出力
 87:          cmpa.l  a2,a0           *バッファの上限に達したか
 88:          bcs     wrt2            *  そうでないならスキップ
 89: wrt1:    bsr     putline         *１行出力する
 90: wrt2:    subq.l  #1,d1           *データのカウンタを１減らし
 91:          bne     wrt0            *  <>0なら繰り返す
 92:
 93:          move.l  a0,bufptr       *ポインタ更新
 94:
 95:          movem.l (sp)+,d1-d2/a0-a3       *}
 96:          *
 97: write0: jmp      0               *オリジナルのwriteへ
 98: write_org        equ     write0+2
 99:
100: *
101: *        １行分の処理
102: *
103: putline:
104:          move.w  efno(pc),d2     *d2=エラーファイルのファイルハンドル
105:          bne     putln0          *d2<>0ならファイルはオープン済み
106:
107:          move.w  #ARCHIVE,-(sp)  *エラーファイルを新規作成
108:          pea.l   efname(pc)      *
109:          DOS     _CREATE         *
110:          addq.l  #6,sp           *
111:          move.w  d0,d2           *
112:          bmi     putln3          *エラーでなければ
113:          move.w  d2,efno         *  ファイルハンドルをワークへ
114:
115: putln0: clr.b    (a0)            *文字列の終端コードを書き込む
116:          lea.l   buff(pc),a0     *a0=出力するデータ先頭
117:
118:          lea.l   errstr(pc),a3   *エラーメッセージか？
119:          bsr     strcmp          *
120:          beq     putln1          *  そうなら細工してから出力
121:
122:          lea.l   wrnstr(pc),a3   *警告メッセージか？
123:          bsr     strcmp          *
124:          bne     putln2          *  そうでなければ生のまま出力
125:
126:                                  *エラー/警告メッセージだった場合
127: putln1: move.w  d2,-(sp)        *ソースファイル名を
128:          pea.l   fname(pc)       *  エラーファイルへ書き出す
129:          DOS     _FPUTS          *
130:          addq.l  #6,sp           *
131:                  *a0はエラーメッセージ中の行番号を指している
132:
133: putln2: move.w  d2,-(sp)        *残りのメッセージを
134:          move.l  a0,-(sp)        *  エラーファイルへ書き出す
135:          DOS     _FPUTS          *
136:          addq.l  #6,sp           *
137:
138: putln3: lea.l   buff(pc),a0     *ポインタ初期化
139:          rts
140:
141: *
```

```
142: *       a0の指す文字列先頭がa3の指す文字列と
143: *                一致するかどうか調べる
144: *       一致した場合     Z=1, a0=一致した部分の直後
145: *       不一致の場合     Z=0, a0は保存される
146: *
147: strcmp:
148:         move.l  a0,-(sp)          *a0=被比較文字列先頭
149:                                   *a3=比較部分文字列先頭
150: strcp0: tst.b   (a3)              *比較文字列がもうなければ
151:         beq     strcp1           *  一致した
152:                                   *そうでなければ
153:         cmpm.b  (a0)+,(a3)+       *  比較してみて
154:         beq     strcp0           *  一致している間は繰り返す
155:                                   *不一致が検出されたら
156:         movea.l (sp)+,a0          *被比較文字列を復帰して
157:         rts                       *  戻る
158: strcp1: addq.l  #4,sp             *待避してあったa0はもういらない
159:         rts                       *一致した
160:
161: *
162: *       DOS $ff40 writeのベクタを書き換える
163: *
164: set_vector:
165:         pea.l   write(pc)         *置き換える処理ルーチン先頭
166:         move.w  #_WRITE,-(sp)     *
167:         DOS     _INTVCS           *
168:         move.l  d0,write_org      *元のベクタを待避
169:         st.b    hooked            *ベクタを書き換えた印を立てる
170:         addq.l  #6,sp
171:         rts
172:
173: *
174: *       DOS $ff40 writeのベクタを元に戻す
175: *
176: reset_vector:
177:         tst.b   hooked            *ベクタを書き換えていないなら
178:         beq     rvec0             *  何もしない
179:
180:         move.l  write_org(pc),-(sp)       *WRITEの元のアドレス
181:         move.w  #_WRITE,-(sp)             *
182:         DOS     _INTVCS                   *
183:         addq.l  #6,sp                     *
184:
185:         clr.b   hooked            *フラグをクリア
186:
187: rvec0:  rts
188:
189: *
190: *       コマンドラインの解析
191: *
192: chkarg:
193:         addq.l  #1,a2             *a2=コマンドライン文字列先頭
194:         move.l  a2,-(sp)          *あとで使うから保存しておく
195:
196:         bsr     fstarg            *空白とスイッチをスキップする
197:         lea.l   temp(pc),a0       *最初のスイッチではない単語を
198:         bsr     getarg            *  ソースファイル名と見なし
```

```
199:                                    *  temp以下に取り出す
200:
201:            movea.l  (sp)+,a2       *a2=AS.Xへ渡す引数
202:
203: *fname以下にフルパスのソースファイル名＋TABの文字列を作成する
204:            lea.l    fname(pc),a0   *a0=ファイル名格納領域
205:            tst.b    (a2)           *引数がもともとなければ
206:            beq      ckarg1         *  すぐ抜ける
207:            pea.l    nambuf(pc)     *ファイル名を展開してみる
208:            pea.l    temp(pc)       *
209:            DOS      _NAMECK        *
210:            addq.l   #8,sp          *
211:            tst.l    d0             *
212:            bne      error2         *負ならエラー
213:            lea.l    nambuf(pc),a1  *
214:            bsr      strcpy         *ドライブ名＋パスに
215:            lea.l    nambuf+NAME(pc),a1
216:            bsr      strcpy         *  ファイル名を加える
217:            lea.l    nambuf+EXT(pc),a1
218:            tst.b    (a1)           *拡張子は省略されているか？
219:            bne      ckarg0         *あればよし
220:            lea.l    sext(pc),a1    *なければ'.S'を補う
221: ckarg0:   bsr      strcpy         *拡張子を加える
222:
223:            move.b   #TAB,(a0)+     *ついでにTABを付け加えておく
224:
225: ckarg1:   clr.b    (a0)           *終端コード
226:
227:            rts
228:
229: *
230: *          コマンドライン中の
231: *            スイッチではない最初の単語位置を得る(a2)
232: *
233: fstarg:
234:            bsr      skipsp         *スペースをスキップ
235:
236:            cmpi.b   #'/',(a2)      *引数の先頭が
237:            beq      farg0          *  /,-であれば
238:            cmpi.b   #'-',(a2)      *
239:            bne      farg1          *
240: farg0:    bsr      skipsw         *スイッチ１個をスキップ
241:            bra      fstarg         *スイッチがなくなるまで繰り返す
242:
243: farg1:    rts
244:
245: *
246: *          スイッチ１個をスキップする
247: *
248: skipsw:
249:            addq.l   #1,a2          *'/'や'-'の分を進めて
250:            lea.l    temp(pc),a0    *  temp以下に
251: *          bra      getarg         *  １語転送する
252:                                    *(転送した文字列は使わない)
253:
254: *
255: *          a2の指す位置から引数１つ分をa0の指す領域へコピーする
```

```
256: *
257: getarg:
258:         move.l   a0,-(sp)          *{レジスタ待避
259: gtarg0: tst.b    (a2)              *1)文字列の終端コードか
260:         beq      gtarg1            *
261:         cmpi.b   #SPACE,(a2)       *2)スペースか
262:         beq      gtarg1            *
263:         cmpi.b   #TAB,(a2)         *3)タブか
264:         beq      gtarg1            *
265:         cmpi.b   #'-',(a2)         *4)ハイフンか
266:         beq      gtarg1            *
267:         cmpi.b   #'/',(a2)         *5)スラッシュ
268:         beq      gtarg1            *
269:         move.b   (a2)+,(a0)+       *  が現れるまで転送を
270:         bra      gtarg0            *  繰り返す
271: gtarg1: clr.b    (a0)              *文字列終端コードを書き込む
272:         movea.l  (sp)+,a0          *} レジスタ復帰
273:         rts
274:
275: *
276: *       コマンドライン先頭のスペースをスキップする
277: *
278: skpsp0: addq.l   #1,a2             *ポインタを進め
279:                                    *繰り返す
280: skipsp:                  *サブルーチンはここから始まる
281:         cmpi.b   #SPACE,(a2)       *スペースか？
282:         beq      skpsp0            *  そうなら飛ばす
283:         cmpi.b   #TAB,(a2)         *TABか？
284:         beq      skpsp0            *  そうなら飛ばす
285:         rts
286:
287: *
288: *       文字列の複写
289: *       リターン時a0は文字列末の00Hを指す
290: *
291: strcpy:
292:         move.b   (a1)+,(a0)+       *1文字ずつ
293:         bne      strcpy            *終了コードまでを転送する
294:         subq.l   #1,a0             *a0は進み過ぎている
295:                                    *a0は文字列末の00Hを指す
296:         rts
297:
298: *
299: *       中断/エラー終了
300: *
301: *
302: break:  lea.l    brkmes(pc),a0     *^Cなどによる中断
303:         bra      errout
304: error1: lea.l    errms1(pc),a0     *AS.Xが起動できない
305:         bra      errout
306: error2: lea.l    errms2(pc),a0     *不正なファイル名
307:         *
308: errout: bsr      reset_vector      *ベクタを元に戻す
309:
310:         move.w   #STDERR,-(sp)     *標準エラー出力へ
311:         move.l   a0,-(sp)          *  メッセージを
312:         DOS      _FPUTS            *  出力する
```

```
313:          addq.l  #6,sp           *
314:
315:          move.w  #1,-(sp)        *終了コード１を持って
316:          DOS     _EXIT2          *  エラー終了
317:
318: *
319: *        データ＆ワーク
320: *
321:          .data
322:          .even
323: *
324: bufptr: .dc.l   buff            *バッファ内の次の書き込み位置
325: efno:   .dc.w   0               *エラーファイルのファイルハンドル
326:                                 *=0ならばオープンされていない
327: hooked: .dc.b   0               *ベクタ書き換え済みかどうかのフラグ
328: prog:   .dc.b   'AS.X ',0       *子プロセスとして起動するプログラム名
329: *
330: errms1: .dc.b   'ASX：AS.Xが起動できませんでした',CR,LF,0
331: errms2: .dc.b   'ASX：ソースファイル名の指定に誤りがあります',CR,LF,0
332: brkmes: .dc.b   'ASX：中断しました',CR,LF,0
333: crlfms: .dc.b   CR,LF,0
334: *
335: errstr: .dc.b   'line ',0              *AS.Xのエラーメッセージ冒頭
336: wrnstr: .dc.b   'Warning: Line ',0     *AS.Xの警告メッセージ冒頭
337: efname: .dc.b   'AS.ERR',0             *エラーファイル名
338: sext:   .dc.b   '.S',0                 *ソースファイルの拡張子
339: *
340:          .bss
341:          .even
342: *
343: cmdlin: .ds.b   STRMAX          *コマンドライン作成用
344: fname:  .ds.b   STRMAX          *アセンブルするファイル名（フルパス）
345: temp:                          *コマンドライン引数１個分のバッファ
346:                                 *  （ダミー）
347: buff:   .ds.b   BUFSIZ          *AS.Xのメッセージを１行溜め込むバッファ
348: nambuf: .ds.b   NAMBUFSIZ       *nameck用バッファ
349: *
350:          .stack
351:          .even
352: *
353: mystack:
354:          .ds.l   1024           *スタック領域
355: mysp:
356:          .end
```

リスト2

```
 1: *        nameck,files,nfiles用オフセット定義
 2:
 3:          .offset 0
 4: *
 5: DRIVE:  .ds.b   2       *ドライブ名  'A:'
 6: PATH:   .ds.b   64+1    *パス名      '¥BIN¥',0
 7: NAME:   .ds.b   18+1    *ファイル名  'ATTRIB',0
 8: EXT:    .ds.b   1+3+1   *拡張子      '.X',0
 9:          .even
10: NAMBUFSIZ:
```

```
11: *
12:         .offset 0
13: *
14: FORSYS: .ds.b   21      *システムが使用
15: FATR:   .ds.b   1       *ファイル属性
16: FTIME:  .ds.w   1       *ファイル最終更新時刻
17: FDATE:  .ds.w   1       *ファイル最終更新日
18: FLEN:   .ds.l   1       *ファイル長
19: PACKEDNAME:             *ファイル名
20:         .ds.b   18+1+3+1
21:         .even
22: FILBUFSIZ:
23: *
24:         .text
```

ASX ［スイッチ］　ソースファイル名

のようにして使用すると，アセンブルしながらカレントディレクトリにAS.ERRの名前でエラーファイルを作成する。

実際にアセンブルする部分は，チャイルドプロセスでAS.Xを起動することで行っているので，プログラム自体は(アセンブラを作ることを思えば)ごく短いものになっている。リスト3のテスト用プログラム(2ヵ所に誤りがある)をアセンブルしたときの実行結果を図2に示す。図2-aがAS.X Ver.1.0の出すエラーメッセージで，図2-bが同時にASX.Xによって作成されるエラーファイルだ。このエラーファイルをED.Xで読み込み，タグジャンプを利用すれば効率的にエラーを修正できる。

■20) ちなみに，ED.Xのタグジャンプ機能とは，テキストの任意の行の頭から，

　ファイル名　行番号

と書いておき，この行にカーソルをあわせてESC・Vを押すと，自動的にそのファイルを読み込んで指定行にカーソルを移動してくれるというものだ。

リスト3
TEST.S

```
1:          .include        doscall.mac
2:
3:          move.w  #'0',-(sp)
4:          DOS     _PUTCHR
5:          adda.b  #2,sp
6:          DOS     _EXIT
```

```
a) AS.Xのエラーメッセージ
X68k Assembler v1.01 Copyright 1987 SHARP/Hudson
line    4 underfined symbol error
line    5 illegal size error
undefined symbol
_PUTCHAR
   2 Fatal error(s)
```

```
b) AS.Xが作成するタグ付きエラーファイル
X68k Assembler v1.01 Copyright 1987 SHARP/Hudson
A:¥WORK¥TEST.S  4 undefined symbol error
a:¥WORK¥TEST.S  5 illegal size error
undefined symbol
_PUTCHAR
   2 Fatal error(s)
```

図2

AS.Xのメッセージを盗みとるには，AS.Xがメッセージ表示に利用しているDOSコールをintvcsでASX.Xの内部ルーチンに一時的に置き換えればよい。調べてみたところ，AS.XはDOSコールwriteを使って標準出力にメッセージを書き出していることがわかったので，ここを乗っ取る。理屈はこれでよいのだが，プログラムにするにあたっては，いくつかの小さな障害があった。

1) writeの呼び出しの中からメッセージ出力だけを抜き出す方法

AS.Xはオブジェクトファイルを作成するプログラムだから，writeの呼び出しが必ずしもメッセージの出力とはかぎらない。メッセージ以外は本来のwriteの処理ルーチンに飛ばしてやる必要がある。この処理の振り分けは，write呼び出し時にスタックに積まれた出力先のファイルハンドルが標準出力かどうかで判断できるだろう。また，write本来の処理ルーチン先頭アドレスはintvcsでベクタを書き換えたときの戻り値として得られる。

2) メッセージの中からエラーメッセージを抜き出す方法

AS.X Ver.1.0のエラーメッセージは必ず〝line〟＋1個以上のスペース＋エラー行で始まるから，行先頭の数文字を比較して一致するようならエラーメッセージと判断する。ASX.Xでは，ついでに警告メッセージも同様の方法で抜き取っている。

3) 安全性への配慮

乗っ取ったDOSコールはプログラム終了時に元に戻しておかなければならない。これは保存しておいた元のベクタ内容を終了前にふたたびintvcsでセットしなおすことで解決する。しかし，これだけではBREAKキーやINTERRUPTスイッチなどで中断されたりすると，ベクタを元に戻さないうちに親プロセスに帰ってしまう可能性があり，不十分だ。そこで〝プロセスを中断した際の戻りアドレス〟を設定して，ダイレクトに親に帰らないように細工する必要がある。これにもやはりintvcsを使う。

intvcsでDOSコール番号として$FFF1_H$を指定すると，〝エラーが発生し中断した際の戻りアドレス〟，$FFF2_H$を指定すると〝BREAKキーが押されてブレイクチェックに引っかかったときの戻りアドレス〟をそれぞれ設定することができる。リスト1では，24～29行がこの処理で，エラーやBREAKキーで中断されたときには301行に飛んでくるように設定している。301行以下ではwriteのベクタを元に戻してから中断された旨のメッセージを出力して，あらためてエラー終了している。なお，リスト1では〝ベクタを書き換える前に中

断された場合″に備え，ベクタを書き換え済みかどうかのフラグを設けて，万全を期している。

ポイントはこんなものだ。この本をここまで読み進んできた読者のことだから，細部については注釈をもとに読み切ってもらえるだろうと思う。

APPENDIX

本書を読むための用語集

本書を読むための用語集

ここには，本文で使用した用語/言い回し＋αのかんたんな解説を収めた。あくまで用語集にすぎず，用語の意味については〝本書における用法〟にかぎっている。また，カタカナ語については【 】内に元の英単語を記してあるが，中には和製英語的な嘘も含まれていると思われる(英和辞典を引くときの参考ぐらいにはなるだろう)。なお，…の後の数字は本文中の関連ページを示している。

数・記号

#▶68000のアセンブリ言語において，即値を指定するときに頭につける記号。

$▶68000など多くのプロセッサのアセンブリ言語やプログラミング言語で，16進数を表すときに頭につける記号。…43

%▶AS.Xにおいて，2進数を表すときに頭につける記号。…43

'▶プログラミング言語において，文字/文字列をくくるのに使用する記号。…43

"▶プログラミング言語において，文字列をくくるのに使用する記号。…43

.【dot】▶カレントディレクトリを意味する記号。また，ファイル名と拡張子の区切り記号。さらには，68000のアセンブリ言語では，オペレーションとサイズを区切る記号であり，疑似命令であることを明示するために疑似命令の先頭につけられる(AS.Xでは省略してもよい)記号でもある。

..【dot dot】▶1階層上の親ディレクトリを意味する記号。

<▶標準入力をリダイレクトするときに使う記号。…31

>▶標準出力をリダイレクトするときに使う記号。…30

>>▶標準出力の内容をファイルに追加するときに使う記号。

^▶ "^A" のように使う場合は，CTRLキーを押しながらAのキーを押すことで入力されるコントロールコードの意味。

^[▶CTRL+[や，ESCキーで入力されるコントロールコード。ESCコード。ASCIIコード1B$_H$。

|▶COMMAND.Xにおいて，パイプを指定するときに使う記号。…119

||▶COMMAND.Xにおいて，1行に複数命令を書くときに命令間を区切る記号。

1の補数【1's complement】▶n桁の無符号2進数において，足しあわせると2のn乗－1になる2つの数の組。1の補数をとる操作をビットレベルで考えると，全ビットを反転することに等しい。本書には登場しなかったが，68000ではnot命令によって1の補数を求めることができる。

1010系列未実装命令▶上位4ビットが1010_Bであるようなマシンコード($Axxx_H$)。Human68kでは，SX-WINDOWのシステムコールに使われている。

1111系列未実装命令▶上位4ビットが1111_Bであるようなマシンコード($Fxxx_H$)。Human68kでは，$FFxx_H$がDOSコール呼び出し，$FExx_H$がFLOATn.Xのファンクション呼び出しにそれぞれ利用される。…338

2進化10進数▶(→BCD)

2進数▶2進法で書き表された数。

2進法▶0と1の2文字だけで数を書き表す方法。2進では，1に1を足すと繰り上がって10になる。

2の補数【2's complement】▶n桁の無符号2進数において，足しあわせると2^nになる2つの数の組。2の補数をとる操作をビットレベルで考えると，全ビットを反転してから1を足すことに等しい。多くのプロセッサでは2の補数で負の数を表現する。本書には登場しなかったが，68000ではneg命令によって2の補数を求めることができる。…87

2バイト半角文字▶2バイトのコードで表現される半角文字。X68000では文字コード80xx$_H$，$Fxxx_H$に2バイト半角文字が割りつけられている。…126, 127

68系▶モトローラのマイクロプロセッサの総

称。6800, 6809, 68000, 68008, 68010, 68020, 68030, 68040等。時に，68000以降のみを指す。

68008▶68000の外部データバス8ビットバージョン。

68010▶68000の仮想記憶対応バージョン。

6809▶モトローラの8ビットマイクロプロセッサ。

8進法▶0～7の8文字で数を書き表す方法。8進では，7に1を足すと繰り上がって10になる。

80系▶インテルのマイクロプロセッサの総称。多くの場合，ザイログZ80系列も含む。

8080▶インテルの8ビットマイクロプロセッサ。

8086▶インテルの16ビットマイクロプロセッサ。

86系▶インテル8088, 8086, 80186, 80286, 80386, 80486等をまとめて指す言葉。多くの場合，日本電気V30も含む。

10進法▶0～9の10文字で数を書き表す方法。10進では，9に1を足すと繰り上がって10になる。

16進法▶0～9，A～Fの16文字で数を書き表す方法。16進では，Fに1を足すと繰り上がって10になる。

A アルファベット順

.A▶XC Ver1.0のライブラリファイルにつけられる拡張子。アーカイバAR.Xで作成/管理される。…221

A系列未実装命令▶(→1010系列未実装命令)

AC【Accumulator】▶アキュムレータを意味する略語。

Acc【Accumulator】▶アキュムレータを意味する略語。

ACK【ACKnowledge】▶ハンドシェイクによるデータ伝送時に，〝了解〟，〝OK〟の意味で相手に返す合図。肯定応答。

AD PCM【Adaptive Differencial PCM】▶パルス符号変調(PCM)のうち，サンプリングしたままのデータではなく，データ間の差分を符号化するのがΔPCM。この差分の重み付けをデータに応じて適合させるのがAD PCM。

ALU【Arithmetic and Logical Unit】▶(プロセッサの)演算装置。

AND▶論理積。

ANSI【American National Standard Institute】▶米国規格協会。

.ARC▶アーカイブファイルに慣例的に使われる拡張子。

AS.X▶Human68kの標準アセンブラ。…13, 24, 340

ASCII(コード)【American Standard Code for Information Interchange】▶標準的な文字コード体系。日本では，ASCIIコードをベースにJISでカタカナ等を拡張したコードを使用している。アスキーコード。

AUXデバイス▶RS-232Cポートを意味するデバイス名。入出力可。

B

.BAK▶バックアップファイルに慣例的につけられる拡張子。

.BAS▶BASIC(X-BASIC)のプログラムファイルにつけられる拡張子。

.BAT▶バッチファイルにつけられる拡張子。

BCD【Binary Coded Decimal】▶2進化10進数。4ビットでは通常0～15の数を表現できるわけだが，これをわざと0～9に制限して数を表す方法。10進数の1桁1桁を2進4桁(16進1桁)に直接対応させ，たとえば1234_Hで1234という10進数を表現する。実数をBCDで表す場合には，浮動小数点表現よりもメモリ効率/速度効率は落ちることになるが，浮動小数点表現では避けられない10進→2進変換過程で生じるまるめ誤差をなくすことができる。

また，BCDは数値←→文字列の相互変換が比較的容易なので，特定の用途では整数の表現にもBCDを使うことがある。本書では登場しなかったが，68000にはデータをBCDとして加減算するabcd, sbcdが用意されている。

BC.X▶X-BASIC→Cコンバータ。

BIOS【Basic Input/Output System】▶基

本入出力システム。低レベルルーチン集。IOCS。

BPB【BIOS Parameter Block】▶ブロックデバイスドライバが組み込み時にデバイスの特性をHuman68kに伝えるために渡すデータ。

BS【Back Space】▶後退コード。バックスペース。^H。ASCIIコード08_{H}。

C

C(言語)【C language】▶プログラミング言語の1つ。

Cビット▶ccrの第0ビット。Carryの略。演算の結果,繰り上がり/繰り下がりが生じた場合にセットされる。…89

Cフラグ▶ccrのCビットの通称。キャリフラグを指す一般的な言葉。

.C▶Cソースファイルにつけられる拡張子。

^C▶CTRL+Cや，BREAKキーにより入力されるコントロールコード。プログラムの実行中断を意味する。ASCIIコード03_{H}。

C compiler PRO-68K▶X68000用メーカー純正プログラム開発セット。通称XC。…13

CCR, ccr▶(→コンディションコードレジスタ)

CLOCKデバイス▶Human68k内部で内蔵時計の制御を行うデバイス名。ユーザーレベルでCLOCKというデバイス名が使えるわけではない。しかしまた，ファイルシステム上はCON等のデバイス名と同等に扱われているために，ファイルにCLOCKという名前をつけることも許されない。…275, 277

CONデバイス▶コンソール(画面とキーボード)を意味するデバイス名。入出力可。

COOKEDモード▶キャラクタデバイスドライバにおいて，データを1バイト単位で入出力するモード。…277, 290

CP/M▶8080用のDOS。

CP/M68k▶68000用のCP/M。

CPU【Central Processing Unit】▶中央演算装置。転じて，中央演算装置の機能を1チップに集積したもの(マイクロプロセッサ)。

CR【Carriage Return】▶復帰コード。キャリッジリターン。^M。ASCIIコード$0D_{H}$。

CRC【Cyclic Redundancy Check】▶巡回冗長符号。データ伝送時に，誤りの発見/修正目的で付加されるコード。

CRT【Cathode Ray Tube】▶ブラウン管。

CRTC【CRT Controler】▶画面制御用チップ/回路。

CTRL+～▶CTRLキーを押しながら他のキーを押すことを表現する。

CTRL-～▶CTRLキーを押しながら他のキーを押すことを表現する。

D

DB.X▶Human68k上のシンボリックデバッガ。…74

.DEF▶BC.Xが参照する関数定義ファイルにつけられる拡張子。

.DIC▶日本語入力FEP用辞書ファイルにつけられる拡張子。

DMA【Direct Memory Access】▶CPUを介さずに，メモリ←→メモリ(さらにはメモリ←→I/O，I/O←→I/O)のデータ転送を行うこと。

DMAC【Direct Memory Access Controler】▶DMAを実現する石。X68000では，フロッピーディスク/ハードディスクの入出力，AD PCMデータの転送に使われている。

.DOC▶慣例的にドキュメントファイルにつけられる拡張子。

DOS【Disk Operating System】▶ディスクに対する入出力サービスを提供するシステム。

DOSコール▶Human68kが提供する一連のサービス群。…21, 336

DPB【Drive Parameter Block】▶Human68kがブロックデバイスを管理するために内部的に使用するデータ。ユーザープログラムからはDOSコールgetdpbで参照できる。

DRAM▶(→ダイナミックRAM)

E

EBCDIC【Extended Binary Coded Deci-

mal Interchange Code】▶エビスディックと読む。IBM生まれの文字コード体系。

ED.X▶Human68k標準添付のフルスクリーンテキストエディタ。…12, 22, 347

EOF【End Of File】▶ファイルの終わり/ファイルエンドコードを意味する略語。

ESC【ESCape】▶エスケープコード。^[。ASCIIコード$1B_H$。

F

F【False】▶偽。

F系列未実装命令▶(→1111系列未実装命令)

FAT【File Allocation Table】▶ディスク上のどの部分が使用中であるか，また，ファイルがディスクのどの部分に収められているかを示す情報。…249, 255, 257

FF【Form Feed】▶(プリンタの)改ページを意味するコントロールコード。改ページコード。フォームフィード。^L。ASCIIコード$0C_H$。

FIFO【First In First Out】▶最初に入れたデータが最初に出てくるようなデータ構造。…314

.FNC▶X-BASICの外部関数ファイルにつけられる拡張子。

G

G【Giga】▶一般には10の9乗のこと。コンピュータの世界では2の30乗のこと。

.GL3▶X-BASICのimg_save()関数等で作成される65536色，横512ドット×縦512ドットのベタフォーマット画像ファイルにつけられる拡張子。また，末尾の数字によって，.GL0なら横256×縦256，.GL1なら横512×縦256，.GL2なら横256×縦512ドットを意味する。

.GM3▶256色，横512ドット×縦512ドットのベタフォーマット画像ファイルにつけられる拡張子。

GP-IB【general purpose interface bus】▶計測分野で広く使われているパラレル伝送インタフェイス規格。

.GS3▶16色，横512ドット×縦512ドットのベタフォーマット画像ファイルにつけられる拡張子。

H

H【High】▶電圧の高い状態。(正論理では)この状態を〝真〟とする。(←→L)

.H▶慣例上，Cのヘッダファイルや，時にアセンブリ言語のインクルードファイルにつけられる拡張子。Headerの意味。

.HLP▶慣例上，ヘルプファイルにつけられる拡張子。

I

.INC▶インクルードファイルにたまにつけられる拡張子。

I/O【Input/Output】▶入出力。I/O空間。I/Oポート。

I/O空間▶メモリ空間のようにアドレスがふられているが，メモリのかわりに各種デバイスがぶらさがっている空間。68系のマイクロプロセッサは独立したI/O空間をもたず，メモリ空間の一部をI/Oに割り当てるメモリマップドI/Oが採用される。

I/Oポート▶入出力の窓口。

IOCS【Input/Output Control System】▶基本的な入出力制御ルーチン集。BIOS。X68000の場合は一部を除き，本体内蔵のROMで供給されている。

ISO【International Standardization Organization】▶国際標準化機構。

J

JIS【Japanese Industrial Standard】▶日本工業規格。

JIS漢字コード▶漢字を表すためにJISで制定した文字コード。21_H～$7E_H$の範囲の値をとる2バイトの組で1文字を表現する。…126

K

k, K【Kiro】▶一般には10の3乗のこと(小文字で表記される)。コンピュータの世界では2の10乗(=1024)のこと(大文字で表記され，〝ケー〟と発音されることが多い)。

L

L【Low】▶電圧の低い状態。(正論理では)この状態を〝偽〟とする。(←→H)
.L▶XC Ver.2.0のライブラリファイルにつけられる拡張子。LIB.Xで作成/管理される。…221
.LB▶SX-WINDOWのリソースファイルにつけられる拡張子。
LF【Line Feed】▶改行コード。ラインフィード。^J。ASCIIコード$0A_H$。
LIFO【Last In First Out】▶最後に格納したデータが最初に出てくるようなデータ構造。スタックはLIFOバッファの典型例。…22, 314
LK.X▶Human68kの標準リンカ。…13, 25
LPTデバイス▶プリンタを意味するデバイス名。出力専用。PRNデバイスと異なり，いっさいのデータ変換を行わずに，出力データをそのままプリンタに送りつける。…275
LSB【Least Significant Bit】▶最下位ビットの意味で使われる略語。

M

M【Mega】▶一般には10の6乗のこと。コンピュータの世界では2の20乗のこと。
.MAC▶慣例上，アセンブリ言語のインクルードファイル/マクロ定義ファイルにつけられる拡張子。
MIDI【Music Instrumental Digital Interface】▶電子楽器制御用の標準シリアル伝送規格。
MML【Music Macro Language】▶音楽演奏データ記述用のマクロ言語。
MPU【Micro Processer Unit】▶モトローラではマイクロプロセッサのことをこう呼ぶ。
MSB【Most Significant Bit】▶最上位ビットの意味で使われる略語。
MS-DOS▶Human68kの設計に大きな影響を与えた8086マシン用のDOS。

N

Nビット▶ccrの第3ビット。Negativeの略。演算結果が負であれば(正でなければ)セットされる。…89, 133
Nフラグ▶ccrのNビットの通称。
NAK【Negative AcKnowledge】▶ハンドシェイクによるデータ伝送時に，〝駄目〟，〝変!〟の意味で相手に返す合図。否定応答。
NAND▶ANDの論理否定。
NOR▶ORの論理否定。
NOT▶論理否定。
NUL▶ヌルコード。ヌル文字。ASCIIコード00_H。
NULデバイス▶何もしないダミーのデバイス名。

O

.O▶Human68kのオブジェクトファイルにつけられる拡張子。
OPM▶X68000など，多くのパソコンに搭載されているYAMAHAのFM音源チップ。
.OPM▶OPMデバイスに出力する形式のMMLファイルに慣例的につけられる拡張子。
OR▶論理和。
OS【Operating System】▶コンピュータ資源を管理する基本ソフトウェア。
OS-9▶6809用のOS。
OS-9/68000▶68000用のOS-9。
OSK▶OS-9/68000の通称。

P

path▶コマンド検索パスを保持する環境変数名。pathはCOMMAND.XのPATHコマンド，またはSETコマンドで設定する。…236

PC，pc▶(→プログラムカウンタ)

PC相対▶(→プログラムカウンタ相対)

PCM【Pulse Code Modulation】▶パルス符号変調。アナログデータを一定間隔でサンプリングして，サンプル値の列にすること(曲線のグラフを，棒グラフを並べたものに置き換えるようなもの)。

.PCM▶AD PCMデータファイルに慣例的につけられる拡張子。

PDB▶何の略だか知らないが，PSPと同じものを指すと思われる略語。

.PIC▶PIC.R(柳沢 明氏作のフリーウェア)形式の画像データファイルにつけられる拡張子。

.PIX▶SX-WINDOWの画像ファイルにつけられる拡張子。

.PRN▶リストファイルにつけられる拡張子。

PRNデバイス▶プリンタを意味するデバイス名。入出力可。…275

PRNファイル▶AS.Xのリストファイルの通称。

prompt▶COMMAND.XのPROMPTコマンドで指定されたプロンプトの形式を保持する環境変数名。

PSP【Program Segment Prefix】▶Human68k/MS-DOSのプロセスの頭に置かれる管理情報。

R

Rファイル▶拡張子が〝.R〟であることから，Human68kの完全リロケータブルな実行形式ファイルを指す言葉。…328

RAM【Random Access Memory】▶読み書き可能なメモリ。

RAWモード▶キャラクタデバイスドライバにおいて，データをいっさい加工せず，まとめて生のまま入出力するモード。…277，290

README▶プログラムのかんたんな使い方を記述したドキュメントファイルに慣例上つけられるファイル名。

README.DOC▶(→README)

ROM【Read Only Memory】▶読み出し専用メモリ。

RS-232C▶シリアル伝送インタフェイスの標準的な規格。

S

Sビット▶srの第13ビット。スーパーバイザステータス。現在の走行モードがユーザーモード(Sビット＝0)か，スーパーバイザモードか(Sビット＝1)を表す。…338

.S▶AS.Xのアセンブリ言語ソースファイルにつけられる拡張子。

^S▶CTRL+SやSHIFT+BREAKキーにより入力されるコントロールコード。(標準出力を通じての)画面表示を一時停止する機能を持つ。ASCIIコード13_{Ho}。…71，161

SCD.X▶Human68k上のCソースコードデバッガ。

SASI【Shugart Associates System Interface】▶多くのパーソナルコンピュータのハードディスクインタフェイスに採用されているパラレル伝送インタフェイス規格。

SCSI【Small Computer System Interface】▶パラレル伝送インタフェイスの標準的規格。

SI【Shift In】▶たとえば，7ビットで256通りのコードを表現する場合に，コード表を2つに分割しておき(そうすれば，各々は7ビットで表現できる)，この2つの表の表裏を切り替えるコントロールコード。SIで表→裏を切り替えたら，SOで裏→表に戻す。

SO【Shift Out】▶SIの項参照。

SP，sp▶(→スタックポインタ)

SR，sr▶(→ステータスレジスタ)

SRAM▶(→スタティックRAM)

.SX▶SX-WINDOWのシステムファイルにつけられる拡張子。

SX-WINDOW▶X68000＋Human68k用ウィンドウシステム。

.SYS▶デバイスドライバ等，システム環境に関わるファイルにつけられる拡張子。

T

T【True】▶真。
Tビット▶srの第15ビット。トレースモードを表すビット。トレースモードでは一命令実行ごとにトレース例外が発生する。もともとデバッガ用の機能で，DB.XでもこのTビットを操作することでシングルステップ実行を実現している。
temp▶COMMAND.XのTEMPコマンドで指定された一時ファイル作成用ドライブ/ディレクトリを保持する環境変数名。…119
TOS【Tape Operating System】▶磁気テープに対する入出力サービスを提供するシステム。
.TXT▶テキストファイルに慣例的につけられる拡張子。

U

UNIX▶UNIX is a trademark of AT&T Bell Laboratories.

V

Vビット▶ccrの第1ビット。oVer flowの略。演算時にオーバーフローが発生したときにセットされる。…110，265
Vフラグ▶ccrのVビットの通称。
V30▶日本電気の8086上位互換/80186下位互換の16ビットマイクロプロセッサ。
Ver.，ver.▶versionを意味する略語。
.VS▶VS.X関連データファイルにつけられる拡張子。

X

.X▶Human68kのソフトウェアリロケータブルな実行形式ファイルにつけられる拡張子。
X68k▶X68000を意味する略語。
XC▶C compiler PRO-68Kの通称。
XOR【eXclusive OR】▶排他的論理和。68000のマシン語ではeor命令で求める。
Xアセンブラ▶AS.Xの正式名称……らしいが，誰もそんなふうに呼んだりはしない。同じような名称に「Xリンカ」，「Xデバッガ」などがある。
Xビット▶ccrの第4ビット。eXtendの略。加減算時の繰り上がり/桁借り，また，ビットシフト時のはみ出したビットを保持する。…266
Xファイル▶拡張子が〝.X〟であることから，Human68kのソフトウェアリロケータブルな実行形式ファイルを指す言葉。…75
Xフラグ▶ccrのXビットの通称。

Z

.Z▶Human68kのロードアドレス固定の実行形式ファイルにつけられる拡張子。
^Z▶CTRLキーを押しながらZキーを押すことで入力されるコントロールコード。Human68k/MS-DOSのテキストファイルの終端コード。ASCIIコード$1A_{H}$。…129
Z80▶ザイログのインテル8080上位互換8ビットマイクロプロセッサ。
Zビット▶ccrの第2ビット。Zeroの略。演算結果が0であればセットされる。…89
Zファイル▶拡張子が〝.Z〟であることから，Human68kのロードアドレス固定の実行形式ファイルを指す言葉。XファイルをCV.Xで変換することで得られる。
Zフラグ▶ccrのZビットの通称。
.ZIM▶ZsSTAFF(ツァイト)の画像ファイルにつけられる拡張子。

あ 五十音順

アーカイバ【archiver】▶複数のファイル(主としてテキスト)を1つのファイルにまとめるツール。書庫の管理ツール。…221

アーカイブ(する)【archive】▶ファイルを1つのファイルにまとめること。また，そうしてひとまとまりにされたもの。書庫。

アキュムレータ【accumulator】▶累算器，という訳がある。演算に使用する(使用できる)レジスタ。68000の場合は，すべてのデータレジスタがアキュムレータとなりうるが，プロセッサによっては1個だけ偉いレジスタ(それがアキュムレータ)があって，必ずデータをアキュムレータに転送してからでないと算術演算/論理演算が行えないこともある。

アクセス(～する)【access】▶ファイル内容/メモリ内容を使用すること。読み書きすること。

アクセスモード▶ファイルを読むのか，ファイルに書くのか，それとも読み書き両方を行うのか，といったファイルの使用法の指定。Human68kのDOSコールopenでは，引数としてファイル名とアクセスモードをとる。…137

上げる▶(→起動)

アセンブラ【assembler】▶アセンブリ言語で書かれたソースプログラムをマシン語に変換するプログラム。…12，24

アセンブリ言語【assembly language】▶本来はビット列にすぎないマシン語を，比較的人間にわかりやすい言葉で表記できるようにしたもの。アセンブリ言語の命令は，基本的にマシン語の命令と1対1に対応し，アセンブラによってマシン語に変換される。…12，37

アセンブル(～する)【assemble】▶アセンブリ言語で書かれたソースプログラムをマシン語に変換すること。…24

アタリ規格▶ジョイスティックインタフェイスの標準的な規格。X68000のジョイスティックポートもアタリ規格。

アドレス【address】▶(メモリ)空間の特定位置を指定するための番号。…34

アドレスエラー【address error】▶68000において，奇数アドレスからワード/ロングワード単位でのメモリアクセスをしようとしたときに発生する例外。…214

アドレスレジスタ【address register】▶68000において，主としてアドレスを保持する目的で用意されているレジスタ。a0～a7。…35，36

アドレスレジスタ間接(形式)【address register indirect】▶68000のアドレッシングモードの1つ。アドレスレジスタが指すアドレスを指定するモード。アセンブリ言語では〝(a0)〟のようにアドレスレジスタ名をカッコでくくって表記する。…47

アドレスレジスタ直接(形式)【address register direct】▶68000のアドレッシングモードの1つ。アドレスレジスタの内容そのものを指定するモード。アセンブリ言語では〝a0〟のようにアドレスレジスタ名で表記する。…42，323

アドレッシングモード【addressing mode】▶狭義ではアドレスの指定のしかた。広義では，オペランドの指定方法全般。…41，322

圧縮(～する)▶記憶域節約のために，冗長性/共通性を利用してデータ列を詰めること。データ圧縮。(←→解凍，展開)

アッパーケース【upper case】▶英大文字。

アッパーコンパチビリティ【upper compatibility】▶(→上位互換性)

アプリケーション【application】▶システムプログラムを除いたプログラム全般。

溢れる▶(→オーバーフロー)

アペンド(～する)【append】▶(データをデータ列末尾に)追加すること。

暗号化(～する)▶機密保持の目的で，テキスト/データ/プログラムの内容がかんたんには解読できないようにすること。

アンパック(～する)【unpack】▶パックされたデータを展開し，元に戻すこと。

暗黙▶とくに表立って指定しないこと(←→明示)

暗黙的な参照▶明示せずに参照すること。たとえば，68000のpea命令では暗黙のうちにspが参照されるし，braやbsrはpcが参照される。

い

以下▶x≦nのとき，xはn以下であるという。
石▶トランジスタ，IC，LSI，VLSIなどなどを指す言葉。チップ。
以上▶x≧nのとき，xはn以上であるという。x＞nのときは，〝xはnより大きい〟という表現を使い，数学/コンピュータの世界では両者は明確に区別される。
移植(〜する)▶ソフトウェアを他のシステムで動作させること/動作するように修正すること。
移植性▶移植のしやすさを意味する語。〝移植性が高い〟といえば，ほとんど手直しなしに移植できるという意味であり，また，〝移植性が低い(ない)〟といえば，移植が非常に困難であるということ。可搬性。ポータビリティ。
依存(する)▶ある条件Aが満たされていなければBが成り立たない/Bが行えないとき，BはAに依存しているという。
一時ファイル▶(→テンポラリファイル)
イニシャライズ【initialize】▶(→初期化)
イミディエイト(形式)【immediate】▶68000のアドレッシングモードの1つ。即値/定数を指定する形式で，アセンブリ言語では〝#10〟のように，頭に〝#〟をつけて表す。…43
入れる▶(変数等に値を)代入する。
インクリメント(〜する)【increment】▶値を増やすこと。1を足すこと。ポインタを進めること。(←→デクリメント)
インクルード(〜する)【include】▶(ソースファイル中に別のソースファイル等を)取り込むこと。
インストール(〜する)【install】▶ハードウェア(コンピュータシステム)を導入すること。ソフトウェアを個々のシステム上で使えるようにすること。
インタフェイス【interface】▶接点/界面。装置間を(つじつまをあわせて)つなぐもの。
インタプリタ【interpreter】▶コンパイラのようにソースプログラムをまとめて翻訳してから実行するのではなく，逐次的に(その場その場で)ソースを解釈しながら実行していくような言語処理系。X-BASICをはじめ，多くのBASICはインタプリタ型言語。
インデックス付きアドレスレジスタ間接(形式)【address register indirect with index】▶68000のアドレッシングモードの1つ。アドレスレジスタにインデックスと8ビットディスプレースメントを加えたアドレスを指定するモード。アセンブリ言語では，〝2(a0, d0)〟のように表記する。この例では〝2〟がディスプレースメント，〝d0〟がインデックス。…324
インデックス付きプログラムカウンタ相対(形式)【program counter with index】▶68000のアドレッシングモードの1つ。プログラムカウンタにインデックスと8ビットディスプレースメントを加えたアドレスを指定するモード。アセンブリ言語では，〝2(pc, d0)〟のように表記する。…326
インプリシット【implicit】▶(→暗黙的な参照)
インプリメンテーション【implementation】▶インプリメントすること。
インプリメント【implement】▶(→実装，実現)
隠蔽(〜する)▶(主として変数名やラベルといった識別子を)他から見えなくすること。

う

ウェイト【wait】▶待つこと。待ち時間。たとえば，高速なプロセッサと安価な低速メモリでシステムを構築する場合には，メモリアクセスごとに1〜数クロックの遅延時間を置く必要がある。また，プログラムの動作が高速すぎて人間がついていけないような場合にも，空ループなどで時間をつぶしたりする。

え

エコー▶(→エコーバック)
エコーバック(〜する)【echo back】▶入力を受け取った側が，その入力データ(入力文字)を送り返すこと。また，そうして送り返されたデータ。たとえば，キー入力時に，入力したデー

タがその場で画面に表示される場合，エコーバックされているという。…29

エスケープキャラクタ【escape character】▶(→エスケープ文字)

エスケープシーケンス【escape sequence】▶(おもにデータ列中において)通常のシーケンスから外れた部分。たとえば，C言語では"¥"ともう1文字の計2文字によって，文字定数/文字列中にコントロールコードを埋め込むことができるが，これがエスケープシーケンス。また，コンソールにESCコードとそれに続く数文字を送ることでさまざまな画面制御を行うことができるが，これもエスケープシーケンス。

エスケープ文字▶エスケープシーケンスの始まりを表す文字コード。とくにESCコード(ASCIIコード$1B_H$)。また，"直後の文字のもつ特別な意味"を打ち消す目的で使われる文字コード。

エディタ【editor】▶(文書・図形などを)編集するプログラム。とくに，テキストエディタ。…12，21

エディット(〜する)【edit】▶編集すること。

エバる【evaluate】▶評価すること。

エミュレーション【emulation】▶エミュレートすること。

エミュレータ【emulator】▶何かをエミュレートするもの(プログラム)。

エミュレート(〜する)【emulate】▶(おもにソフトウェアによってハードウェアを)模倣すること。疑似的に同じ動作をさせること。

エラー【error】▶誤り。異常。不具合。時に，プログラム側の都合。

エンコード【encode】▶符号化すること。(←→デコード)

エンディアン【endian】▶(→バイトエンディアン)

エンドコード【end code】▶(→終端コード)

お

押し込む▶(→プッシュ)

押し戻す▶(→プッシュバック)

オーバーフロー(〜する)【over flow】▶演算結果が結果格納領域に収まらなくなった等の理由で演算に失敗すること。桁あふれ。バッファがすでにいっぱいなのにさらにデータを追加しようとすること。もしくは，強引に追加してバッファ以外の領域にまでデータを書き込んでしまうこと。…110，265

オーバーフローフラグ▶(→Vビット)

オブジェクト(ファイル)【object file】▶アセンブルすることによって得られたマシン語ファイル。とくに，リロケータブルオブジェクトファイルを指すことが多い。…24，25

オフセット【offset】▶(アドレス等の)差。

オフセット表▶AS.Xの.offset疑似命令で記号定数に定義される，オフセットの表。…287

オプティマイズ(〜する)▶(→最適化)

オープン(〜する)【open】▶ファイルを読み書きできるようにするときの約束ごと。…137

オペコード(OP-code)▶マシンコード中，オペランドを除いた部分。狭義のマシンコード。オペレーションコード。

オペランド【operand】▶(マシン語の)演算/操作の対象。(数学でいう被演算数の意味で)式中の演算対象。…37，38

オペレーション【operation】▶操作。…37，38

親プロセス▶子プロセスを生成したプロセス。

下ろす▶(→ポップ)

か

改行(〜する)▶カーソルを次の行に移動すること。…26

改行コード▶改行を意味するコントロールコード。LFコード(ASCIIコード$0A_H$)。広義ではテキストファイルの行末を意味するコードで，Human68k/MS-DOSでは$0D_H$，$0A_H$の2バイトの組。…26

解決(〜する)▶リンク時において，まだ値の定まっていないシンボルの値を確定すること。…221

下位互換(性)▶ほぼ互換性はあるが，若干機能が不足していること。アンダーコンパチ。

外字▶ユーザーレベルでフォントを設定できる文字。…127

解釈(〜する)【interpret】▶(言語処理系等が)入力データの(文法上の)意味を解析すること。
解析(〜する)【parse, analyze】▶(言語処理系等が)入力データの(文法上の)意味を解釈すること。
解凍(〜する)▶圧縮されたデータを元通りに展開すること。
外部記憶(装置)▶コンピュータ外部に存在する記憶装置。
解放(〜する)▶確保したメモリ領域等を明け渡すこと。
カウンタ【counter】▶数を数えるもの。数えた数を保持する変数/レジスタ。
カウント(〜する)【count】▶数を数えること。
カウントアップ(〜する)【count up】▶数え上げること。カウンタをインクリメントすること。
カウントダウン(〜する)【count down】▶カウンタをデクリメントすること。
返す▶プログラム/サブルーチン/関数が呼び出し元になんらかの情報を引き渡すこと。
拡張子▶ファイル名の〝.〟以降の部分。
拡張シンボルテーブル▶SCD.Xでソースレベルのデバッグをするために,XC Ver.2.0がXファイルに埋め込む情報。
確保(〜する)▶メモリ領域等を自分以外が使えなくすること。使うことを他者に宣言すること。
可視性▶識別子のスコープがどうなっているか,ということ。…212
仮想記憶【virtual memory】▶実装されているメモリ容量以上のメモリがあるように見せる仕掛け。
仮想的【virtual】▶本当は実在しないものを,あたかも存在するかのように見せること。
可読性▶(プログラムの)読みやすさ。
可搬性▶(→移植性)
空ループ▶ループすること以外に意味のある動作を行わないループ。速度調節に使われる。
仮引数【parameter】▶マクロや高級言語の関数などを定義するときに,定義の都合上,引数につけた仮の名前。…285
カレントディレクトリ【current directory】▶階層ディレクトリ中,現在ユーザーがいる(作業をしている)ディレクトリ。
環境【environment】▶整えるためには手間と金がかかるもの。
環境変数【environment variable】▶シェルの変数。

き

偽【false】▶条件が成り立っていないこと。真でない状態。
キー【key】▶キーボードに並べられたボタン。また,ソート時の並べ替えの鍵(住所録をソートする場合なら,名前順にソートするか,名字順にソートするか,電話番号順にソートするか,といったキーの選択余地がある)。さらには,データ検索/抽出時の検索対象指定。あるいは,暗号化時の合言葉的な短い文字列。
機械語▶(→マシン語)
記号定数▶シンボルの形で表された定数。…135, 150
疑似命令▶アセンブリ言語上は命令のように見えるが,マシン語命令ではないもの。アセンブラに対する指示/指令。
機種依存性▶その機種特有の機能を使わずに,どんなシステムにもある基本的な入出力だけを使って書かれたプログラムは機種依存性が低い。逆に,高精度のグラフィックといった機種特有の機能をふんだんに使ったプログラムは機種依存性が高い。(←→移植性)
疑似乱数▶コンピュータでは真の乱数を作ることはできないので,計算によって一見乱数のように見える数列を作って利用する。そうして作られた乱数もどきのことを疑似乱数という。
基数▶記数法において,数を表す基本となる数。
奇数パリティ▶ビット列中の〝1〟であるビットの個数が奇数になるように,〝0〟か〝1〟のパリティビットを付加すること。こうしておくと,データ伝送時に受け取ったデータ中のパリティビットも含めた〝1〟の個数が奇数でなければ,ビット化けがあったものと判断できる。
記数法▶数を表記する方法。n進法。

キースキャン【key scan】▶キーボードのどのキーが押されているかを調べること。
起動(する)▶ソフトウェアを(外部記憶装置からメモリに読み込んで)実行すること。ハードウェアの電源を入れること。
キーバッファ▶(先行入力時に)入力されたキーを蓄えておくバッファ。…164
キャラクタ【character】▶文字。
キャラクタデバイス▶プリンタ等，文字単位でシリアルに入出力を行うデバイス。…147，277，278
キャリ【carry】▶繰り上がり。また，キャリフラグ。
キャリフラグ▶(→Cビット)
キュー【queue】▶待ち行列。…314
境界条件▶ある処理が成り立つか，成り立たないかの境目となる条件。2つの処理のどちらを選択するかの境目となる条件。…125
許可(〜する)▶(論理的に)できるようにすること。許すこと。
局所的【local】▶(プログラムの)一部でのみ通用/存在すること。…212
切りのいい数▶コンピュータの世界では2のn乗のこと。
(パスを)切る▶環境変数pathを設定すること。
禁止(〜する)▶(論理的に)できないようにすること。禁じること。

く

クイックイミディエイト(形式)【quick immediate】▶68000のアドレッシングモードの1つ。イミディエイト形式の一種で，小さな即値/定数を指定する形式。表現できる数値の範囲も適用できる命令も限られるが，命令コードは短く，実行速度も速い。…97，334
空行▶文字どおり，空の行。改行コードだけからなる行。
偶数パリティ▶ビット列中の〝1〟であるビットの個数が偶数になるように，〝0〟か〝1〟のパリティビットを付加すること。
空文▶文字どおり，空の文。高級言語において，文法上は〝文〟が必要なのだが，実際にそこに当てはめるべき処理がないような場合に使われる。C言語では〝;〟で表す。
区切り文字【separator，delimiter】▶文字(語)と文字(語)の間を区切る文字。COMMAND.XやAS.Xではスペースないしはタブ。
クリア【clear】▶(画面などを)消去すること。(レジスタ/メモリ/ビットなどを)0にすること。
クローズ(〜する)【close】▶ファイルをオープンし，読み書きした後で行う後始末。…140
クロック【clock】▶コンピュータの動作タイミングの基本となるパルス信号。…334
グローバル【global】▶(→大域的)
食わせる▶入力する。プログラムにデータを与える。

け

ゲタ▶処理を簡略化したりする目的で，データに加減算しておく定数。
検索(〜する)▶データ列の中から特定の条件に合致するデータを探し出すこと。

こ

広域的▶(→大域的)
高級(プログラミング)言語【high level programming language】▶アセンブリ言語以外のプログラミング言語の総称。高水準言語(こっちの訳語のほうが好きなのだが)。
高水準▶(処理が)ハードウェアから切り離されている(もしくは遠い)こと。
高水準言語▶(→高級言語)
構造体▶複数のデータ(同じ種類である必要はない)をひとまとめにして扱えるようにしたデータ構造(のC言語における呼び名)。
効率▶無駄がなければ(少なければ)効率がよいといい，無駄が多ければ効率が悪いという。
高レベル▶(→高水準)
互換性【compatibility】▶複数のハード/ソフトで，同一のデータ/プログラム/ハードが使用できるかどうか，ということ。

固定小数点数▶たとえば，小数点以下に16ビット，整数部に16ビットというように固定長の領域を割り当てて，実数を表現する方法。ちょうど実数データをシフトして整数化してあるようなものだから，加減算は整数とまったく同様に行うことができる。また，乗除算も整数よりも若干よけいな手間がかかる程度で行える。ただし，演算結果によっては有効数字が大幅に減ってしまう場合もあり，浮動小数点形式を使う場合よりも演算精度は落ちる。

コーディング(〜する)【coding】▶符号化すること。とくに，処理を具体的なプログラミング言語で書き表すこと。または，そうして書き表されたもの。狭義でのプログラミング。…21

コード【code】▶符号。意味を持ったビット列。時にプログラムのソースコード。

子プロセス【child process】▶他のプロセスによって生成されたプロセス。…224，234，237

コマンド行▶(→コマンドライン)

コマンドプロセッサ▶OSのうち，ユーザーからの指令を解釈・実行する部分。いわゆるシェル。

コマンドライン【command line】▶プログラムに対する指示を入力する行。もしくは，そうして入力された1行。コマンド行。

コマンドラインインタプリタ▶コマンドラインに与えられた指令を解釈・実行するプログラム。いわゆるシェル。

コメント【comment】▶(プログラムの読み手にわかるように)各部の動作/機能をソースプログラム中に記したもの。注釈。…37，38，199

コモンエリア【common area】▶Human68k Ver.2.0でサポートされたプロセス間通信用のメモリ領域。また，AS.Xにおいて.commで定義されたデータが置かれるメモリ領域。

コール(〜する)【call】▶サブルーチンを呼び出すこと。

壊す▶(→破壊)

コンディションコードレジスタ【condition code register】▶srの下位バイト。演算結果を反映する複数のフラグが寄せ集められており，条件分岐時に参照される。…88，265

コントローラ【controler】▶特定のハードを制御するもの(チップ)。

コントロールコード【control code】▶(おもにコンソールに対して)なんらかの制御を要求する決められたコード。ASCIIコードでは00_H〜$1F_H$(時に$7F_H$も)がコントロールコードに割り振られており，これらのコードをコンソールに出力することで，復帰/改行/カーソル移動等の画面制御を行うことができる。…26

コンパイラ【compiler】▶コンパイルするプログラム。

コンパイル【compile】**(〜する)**▶高級言語で書かれたソースプログラムをマシン語等に翻訳すること。

コンパチ▶(→互換性)

コンパチビリティ【compatibility】▶(→互換性)

ゴミ▶意味のない値。たとえば，電源投入直後のメインメモリの内容や，レジスタの内容は不定であり，ゴミである。

さ

再帰(〜する)▶サブルーチンが自分自身を直接あるいは間接的に呼び出すこと。〝自分の定義〟に自分自身を使うこと。この用語集もところどころ再帰しているかもしれない。

再帰的▶再帰する様子。数学でいう帰納的。

サイズ【size】▶大きさ。バイト数。とくに，68000ではオペランドのビットの長さが，8ビットなのか，16ビットなのか，32ビットなのか，という指定のこと。…44

最適化(〜する)▶(プログラムの)効率を上げること。メモリ効率重視の最適化と，実行速度重視の最適化があるが，多くの場合，両者は相反する。

再入可能▶(→リエントラント)

再配置可能▶(→リロケータブル)

逆上がり▶「マシン語プログラミングができるようになるのは，逆上がりができるようになるのと同じぐらい難しい」(村田談)。

サーチ(〜する)▶(→検索)

サブCPU【sub CPU】▶CPUの負担を軽くするために，特定のハードの制御専用に用意されたCPU。要するに，恐竜の神経節。X68000で

はキースキャンにサブCPUが用いられており，68000の状態にかかわらずつねにキーボードを見張っている。キーが押されると，そのことを68000に知らせるために割り込みをかける。

サブルーチン【subroutine】▶副プログラム。

サブルーチンコール(～する)▶サブルーチンを呼び出すこと。

サンプリング(～する)▶連続的に変化するデータを一定間隔で抜き出して，その抜き出した値(サンプル値)で区間全体を代表させること。

し

シェル【shell】▶コマンドラインインタプリタのUNIX流の呼び名。他のプログラムを起動/実行する機能を持ったインタプリンタ型言語。

識別子▶(プログラミング言語中)他の同種の存在と区別するためにつけられた名前/記号。ラベル名や，変数名等。

資源▶メモリ，外部記憶スペース，時間等，プログラムの動作に必要なもろもろのもの。

シーケンシャル【sequential】▶連続的な。順番に。

システム【system】▶組織的なひとまとまりのもの。時に，OS等の基本ソフトウェアのこと。

システムコール【system call】▶OS等のシステムが提供する汎用サブルーチン集。

システムバイト【system byte】▶srの上位バイト。

実現(～する)▶ある概念/処理を(ソフト/ハードで)具体化すること。

実効アドレス【effective address】▶マシン語命令において，操作対象となった(なる)メモリのアドレス。…99

実行(可能)ファイル【executable file】▶(OS上でコマンドとして)実行できるファイル。プログラムファイル。時には，バッチファイル等も含む。…25，26，75，328

実装(～する)▶あるもの/機能を(おもにハードに)搭載すること。

実引数【argument】▶プログラム/サブルーチン/マクロ等が呼び出されたときに与えられた引数そのもの。

自転車▶「マシン語プログラミングができるようになるのは，自転車に乗れるようになるのと同じぐらい難しい」(村田談)。

死ぬ▶プログラムが暴走した状態を表現する言葉の1つ。ソフト/ハードが壊れた状態を表現する言葉の1つ。

シフト(～する)【shift】▶ビット列/データ列をずらすこと。…266

シフトJIS漢字コード▶日本のパソコンでもっともふつうに使われている漢字コード。…61，126

シミュレーション【simulation】▶シミュレートすること。

シミュレータ【simulator】▶何かをシミュレートするもの(プログラム)。

シミュレート(～する)【simulate】▶(おもにソフトウェアによって他のソフトウェア/現象等を)模倣すること。

ジャンプ(～する)▶分岐すること。

ジャンプテーブル▶多方面分岐時に参照される分岐先アドレスを並べたテーブル。…303，329

終端コード▶データ列の最後を示すコード。エンドコード。

周辺【peripheral】▶(プロセッサの)周りを取り囲むもの。(プロセッサにとっての)外部デバイス。

終了コード▶プログラム終了時に呼び出し元に返す値。プロセスが終了時に呼び出し元に返す値。終了ステータス。Human68kの場合はDOSコールexit，exit2で終了コードを返す(0ならば正常終了を表す)。呼び出し元ではDOSコールexecの戻り値として終了コードを受け取るが，DOSコールwaitによって後から取得しなおすこともできる。また，終了コードはCOMMAND.XのIF中，ERRORCODE，EXITCODEにより参照することもできる。…58，236，297

終了ステータス▶(→終了コード)

主記憶▶(→メインメモリ)

出力▶コンピュータが生きている証。

準拠▶本来はある規格に完全に合致していること。通常の使い方では〝だいたい合致している〟こと。とくに，〝完全準拠〟といった場合に

は〝ほとんど合致している〟こと。

仕様▶ソフト/ハードの，機能/使い方/制限をまとめたもの。よく言い訳に使われる(「それはバグではありません。仕様です」)。

上位互換(性)▶互換性があり，さらに若干の機能が追加されていること。大が小を兼ねること。

条件付きアセンブル▶ソースの一部分を，ある条件が成り立っているときにのみアセンブルするようなアセンブラの仕掛け。AS.Xで条件付きアセンブルするときには，疑似命令.if～.elif～.else～.endifを使う。

条件(付き)分岐▶ある条件が成立している場合にのみ分岐すること。(←→無条件分岐)…40, 89, 133

初期化(～する)【initialize】▶動作/処理に必要なもろもろのセットアップをすること。変数/レジスタ/メモリに初期値を入れること。また，初期状態に戻すこと。イニシャライズ。

書庫▶(→アーカイブ)

処理系▶なんらかの処理を行うもの。システム。とくに，言語処理系(アセンブラ，コンパイラ，インタプリタ等)。

シリアル【serial】▶連続的な。順番に。直列の。

シリアル通信▶(→シリアル伝送)

シリアル伝送▶(制御用の信号線以外に)1本の信号線だけでデータを伝送すること。

真【true】▶条件が成り立つこと。偽でない状態。

シンボリックデバッガ【symbolic debugger】▶プログラム中で使われたシンボルの形で，アドレスや数値を表示/指定できるデバッガ。

シンボル【symbol】▶識別子として使われる文字列/語/記号。

シンボルテーブル▶シンボルのテーブル。言語処理系では定義されたシンボルをテーブルの形で管理するのがふつう。また，Human68kのXファイルには(DB.Xでシンボリックデバッグするときのために)外部定義されたシンボルのテーブルが付加される(LK.Xで/Xスイッチを指定してリンクすれば，シンボルテーブルは付加されない)。

真理値▶真偽。真偽を1と0で表したときのその値。

す

スイッチ【switch】▶プログラムの動作を変えるオプション指定。

スキャン(する)【scan】▶走査すること。端から片っ端にチェックしていくこと。

スコープ【scope】▶プログラミング言語において，識別子(ラベル，シンボル，変数名，関数名)等にアクセスできる範囲。通用範囲。

スタック【stack】▶LIFO構造をもつデータ構造。サブルーチンコール時に戻りアドレスを待避しておいたり，レジスタ内容を一時的に格納しておいたりするのに使う。…22, 49, 50

スタックセクション【stack section】▶プログラム中，主としてスタック領域を置くのに使われる部分。AS.Xでは.stack疑似命令によって，以降がスタックセクションであることを宣言する。Human68kのスタックセクションは，アセンブルの時点でブロックストレージセクションに併合される。…83, 219

スタックトップ【stack top】▶最後にスタックに積まれたデータ。もしくは，その格納場所。…23

スタックフレーム【stack frame】▶スタック上に確保したローカルエリアにアクセスしやすくするための枠組み。…210

スタックポインタ【stack pointer】▶スタックトップを指すポインタ。…23

スタティック【static】▶(→静的)

スタティックRAM【static RAM】▶一定期間ごとにリフレッシュしなくても内容が保持できるようなRAM。SRAM。通常，DRAMよりも記憶密度が低く，高価で，高速。

ステータスレジスタ【status register】▶68000が走行に必要な各種情報を保持しておくレジスタ。sr。…35, 36, 337

ストア(～する)【store】▶メモリにデータを格納すること。

ストラテジルーチン【strategy routine】▶Human68kのデバイスドライバ中，リクエストヘッダを受け取るだけのルーチン。…276, 298

スーパーバイザスタックポインタ【supervisor stack pointer】▶68000において，スーパーバイザモードで使用されるスタックポインタ。ssp。…80，229
スーパーバイザステータス【supervisor status】▶(→Sビット)
スーパーバイザモード【supervisor mode】▶68000の走行モードの1つ。おもに，OSなどのシステムプログラムを実行するためのモード。…80，229
スラッシュ【slash】▶〝/〟記号。
スレッド【thread】▶Human68k Ver.2.0からサポートされたバックグラウンド処理において，平行して実行されるそれぞれのプログラムを指す言葉。
スワッピング【swapping】▶交換すること。とくに，(仮想記憶を実現するために)メモリ内容とディスク内容を交換すること。

せ

制御キャラクタ▶(→コントロールコード)
制御文字▶(→コントロールコード)
静的【static】▶(プログラムの動作中，変数等が)動かずにずっと存在しつづける様子。…212
正論理▶0(電圧の低い状態)で偽を，1(電圧の高い状態)で真を表すこと。
絶対アドレス【absolute adderss】▶空間上の1点と1対1に対応する数字。…35
絶対ショートアドレス(形式)【absolute short address】▶68000のアドレッシングモードの1つ。16ビットで表現できる範囲の(0000_H～$7FFF_H$, $FF8000_H$～$FFFFFF_H$)絶対アドレスを指定するモード。アセンブリ言語では〝$800.w〟のように表記する。…335
絶対パス(名)▶ルートディレクトリを基点としたパス指定。
絶対分岐▶分岐先を絶対アドレスで指定する分岐。…332
絶対ロングアドレス(形式)【absolute long address】▶68000のアドレッシングモードの1つ。特定のメモリを32ビットの絶対アドレスで指定するモード。アセンブリ言語では〝$80000〟のように表記するが，とくにロングであることを明示したいときには〝$80000.l〟のように後ろに〝.l〟をつける。…43，334
セット(～する)【set】▶値を入れる。ある状態にする。ビットを1にする。フラグをON(真)の状態にする。(←→リセット)
セパレータ▶(→区切り文字)
セーブ(～する)【save】▶値/データを記憶装置に保存すること。
ゼロクリア【zero clear】▶レジスタ/メモリなどを0で埋めて初期化すること。
全角文字▶天地と横幅が等しい書体の文字。
先行入力▶キー入力待ちのとき以外に，キー入力すること。先行入力されたキーはキーバッファに溜め込まれ，後でプログラムがキー入力を要求したときに取り出される。…164
センス(～する)【sense】▶検知すること。狭義では，(割り込みによってデータが追加されるような)バッファにデータがあるかどうかを確認すること。
セントロ規格▶(→セントロニクス規格)
セントロニクス規格▶プリンタ用パラレルインタフェイスの標準的な規格。

そ

相対アドレス【relative address】▶基準アドレスからの差で表現されたアドレス。…35
相対パス(名)▶カレントディレクトリを基点としたパス指定。
相対分岐▶分岐先を現在位置からの相対的な距離で指定する分岐。…332
装置情報▶ファイルハンドルに割り当てられたものが，キャラクタデバイスであるか，ブロックデバイス上のファイルであるかといった種類を示す情報。…148，149，290
双方向リスト▶データ構造の一種。正逆両方向に辿ることができるように作られたリスト。つまり，直前のデータへのポインタと，直後のデータへのポインタの両方を持っているようなリスト。
即値【immediate data】▶定数。
ソース【source】▶転送/演算の元データ。転送

元。また，ソースファイル。ソースコード。…38
ソースコード▶コンパイラ/アセンブラでコンパイル/アセンブルする前のプログラム。
ソースコードデバッガ▶コンパイル後の実行ファイルをそのソースリストと関連づけることで，一見ソース上でデバッグしているように見せるデバッガ。
ソースファイル▶プログラムの入力となるファイル。とくに，コンパイラ/アセンブラに与える前のプログラムファイル。
ソースレベルデバッガ▶(→ソースコードデバッガ)
ソーティング【sorting】▶ソートすること。
ソート(〜する)【sort】▶ある規則にしたがってデータ列を並べ替えること。整列。

た

大域脱出▶深くネストしたサブルーチン呼び出しの奥底から，通常の手続によらず，いっきに呼び出し元へ戻ったり，プログラムの実行そのものを終了すること。
大域的【grobal】▶変数等がプログラムのどこからでも参照できる状態を表す言葉。広域的。…212
ダイナミック【dynamic】▶(→動的)
ダイナミックRAM【dynamic RAM】▶一定期間ごとにリフレッシュしなければ，内容が保持できないようなRAM。DRAM。通常，SRAMよりも遅いが，記憶密度が高く，安価なので，メインメモリに用いられている。
ダイナミックリンク【dynamic link】▶ライブラリのリンクを実行形式ファイル生成時にではなく，プログラム実行時に行う方法。OS-9等に見られる。ディスクなどの外部記憶に用意されたモジュールのうち，プログラムで使われるものだけがメモリ上に読み込まれる。複数のプログラムから使われるモジュールはメモリ上にはただ1つだけ存在すればよい。ただし，その場合，各モジュールはリエントラントに作られている必要がある。
待避(〜する)▶(後で使う事態に備えて)レジスタ/変数の内容などをどこかに一時的にしまっておくこと。
対話的【interactive】▶ユーザーの指示とプログラムの動作が交互に行えること。ユーザーの指示がその場で反映されること。(←→非対話的)
倒す▶フラグを偽の状態にすること。リセットすること。
タグジャンプ【tag jump】▶テキストエディタにおいて，あらかじめつけておいた識別用の名札によって，該当ファイルの該当位置にダイレクトにジャンプする機能。…347
ターゲット【target】▶対象。
ターゲットプログラム▶デバッグ対象となるプログラム。
タスク【task】▶仕事。(OSにおける)処理単位。
立ち上げる▶(→起動)
立つ▶フラグが真の状態になること。ビットが1になること。セットされること。
立てる▶フラグを真の状態にすること。ビットを1にすること。セットすること。
黙る▶(→ハングアップ)
ターミナル【terminal】▶端末(装置)。コンピュータとユーザーのインタフェイスとしての入出力装置。通常，キーボードとディスプレイ(ないしはプリンタ)からなる。
ターミナルモード▶マシンを端末として使うモード。X68000もTERM.Xによって端末として使える。
探索(〜する)▶(→検索)
ダンプ(〜する)【dump】▶メモリ内容/ファイル内容などを決まった書式で表示すること。とくに，16進数で表示すること。
端末▶(→ターミナル)

ち

チェイン(〜する)【chain】▶(データが)鎖状にリンクすること。
チェックサム【check sum】▶データ伝送時にデータが正しく伝送できているかどうかを調べる指針として付加される情報。データの総和。
チャイルドプロセス▶(→子プロセス)

中間コード▶記憶領域の節約や処理速度向上のために，符号化されたデータ/命令/プログラム。たとえば，BASICではプログラムを入力された文字列のままではなく，中間コードの形で保持することが多い。
注釈▶(→コメント)
逐次(的)▶順番に。次々に。〝まとめてやらずに〟というニュアンスがある。

つ

通用範囲▶(→スコープ)
積む▶(→プッシュ)
ツール【tool】▶道具としてのプログラム。

て

低水準▶(処理が)ハードウェアに密着している(もしくは近い)こと。〝コンピュータコンピュータしている〟こと。
定数【constant】▶(プログラム中で)値が一定不変の数。
定数式▶定数だけからなる式。アセンブル/コンパイルの時点で値が計算できてしまう(値が定まっている)式。…38
定数定義▶定数をシンボルに定義すること。…25，135，150
ディスプレースメント【displacement】▶(アドレス等の)変位。ずれ。
ディスプレースメント付きアドレスレジスタ間接(形式)【address register indirect with displacement】▶68000のアドレッシングモードの1つ。アドレスレジスタに16ビット符号付き数のディスプレースメントを加算したアドレスを指定するモード。アセンブリ言語上では，〝4(a0)〟のように表記する。…146，207，328
ディスプレースメント付きプログラムカウンタ相対(形式)【program counter with displacement】▶68000のアドレッシングモードの1つ。プログラムカウンタに16ビット符号付き数のディスプレースメントを加算したアドレスを指定するモード。アセンブリ言語上では，〝4(pc)〟のように表記する。
ディレクトリ【directory】▶ファイルをひとまとめにしたもの。…249
低レベル▶(→低水準)
テキストエディタ【text editor】▶テキストの作成/編集を行うためのツール。
テキストセクション【text section】▶プログラム中，主として命令コードそのものを置く部分。AS.Xでは.text疑似命令によって，以降の命令をテキストセクションに置くことを宣言する。…52，219
テキストファイル【text file】▶文書ファイル。文字と若干のコントロールコードにより構成され，typeコマンドで読むことができるファイル。
デクリメント(〜する)【decrement】▶値を減らすこと。1を減じること。ポインタを戻すこと。
デコード(〜する)【decode】▶符号化の逆変換を行うこと。とくに，プロセッサがフェッチした命令を解釈すること。
デスティネーション【destination】▶データの転送先。演算結果の格納先。…38
データ【data】▶datumの複数形。
データ構造【data structure】▶データの構造(説明になってない)。
データセクション【data section】▶プログラム中，主として定数データを置く部分。AS.Xでは.data疑似命令によって，以降をデータセクションに置くことを宣言する。…160，219
データバス【data bus】▶データ伝送用のバス。
データレジスタ【data register】▶68000において，主としてデータの保持/演算に使用する目的で用意されているレジスタ。d0〜d7。…35，36
データレジスタ直接形式【data register direct】▶68000のアドレッシングモードの内容を指定するモード。アセンブリ言語では〝d0〟のようにデータレジスタ名で表記する。…42
デッドコード【dead code】▶プログラム中に存在はしても，実際には使われることのない部分。
手抜き▶〝そういうことは誰もしない〟，〝そう

いうことは起こらない〟という仮定のもとに，処理を省略すること。未必の故意。

デバイス【device】▶(周辺)装置。

デバイス属性▶デバイスドライバの種別等を表す情報。…277

デバイスドライバ【device driver】▶なんらかの物理的/論理的装置を駆動するもの(プログラム)。時に，CONFIG.SYS中，#device～行でシステムに組み込めるプログラムのこと。…274

デバイス名▶キャラクタデバイスをあたかもファイルであるかのように扱うためにつけられた，デバイスドライバ固有の名前。…173，278

デバッガ【debugger】▶デバッグ作業を助けるプログラム。…74

デバッグ(～する)【debug】▶プログラムの誤り(バグ)を探し出し，修正すること。プログラミングの真髄。

デフォルト【default】▶特別な指定がなされなかったときに採用される値/動作。また，その状態。

テーブル【table】▶表。狭義では2次元配列状のデータを指すが，1次元の定数データ列のこともテーブルと呼んでいる。…303，309，329

デリミタ【delimiter】▶(→区切り文字)

電脳倶楽部▶満開製作所が毎月1度発売している自称〝2HDディスクに入ったX68000のための雑誌〟。

テンポラリ【temporary】▶一時的な，の意。

テンポラリファイル▶プログラム動作中に一時的にディスク上にとられる作業用のファイル。通常はプログラム実行終了時点で削除される。…119

と

同値▶等価，等値，同等を意味する論理/数学用語。pとqの真偽が一致しているとき，pとqは同値であるという。また，pならばq，かつ，qならばpのときも，pとqは同値であるという。

動的【dynamic】▶〝必要に応じて〟，〝その場で〟等の意味。とくに(変数等が)必要に応じて生成・破棄される様子。〝動的なメモリ確保〟といえば，メモリ領域が必要になったときにその場で(OS等に要求して)メモリを確保すること。…212

(パスを)通す▶OSのコマンド検索パス(環境変数path)に特定のディレクトリを追加すること。

ドキュメンテーション▶(資料，マニュアル的な)文書。

ドキュメント▶文書化された情報。文書。

独立(～する)▶ある条件Aが満たされているかどうかにかかわらず，Bが成り立つ/Bが行えるとき，BはAから独立しているという。

独立性▶ソフトウェアについていう場合，おもにサブルーチンがどれだけメインルーチンや他のサブルーチンから切り離されているかを意味する語。…199

閉じる▶(→クローズ)

特権命令▶68000において，スーパーバイザモードでのみ使用可能な命令。ユーザーモードで使おうとすると〝特権違反による例外〟が発生し，Human68kでは〝特権違反です〟のメッセージとともにプログラムの実行を中断する。…339

飛ぶ▶プログラムが暴走した状態を表現する言葉の1つ。とくに，プログラムが最後のあがきに何か一発芸的な動作をしたときに使うことが多い(例：画面が真っ暗になる。意味不明の文字が表示される。画面モードが切り替わる。COMMAND.Xが終了してしまってHuman68kのプロンプト〝#〟が出る。ディスクがイジェクトされる……. etc)。

ドライバ【driver】▶何か(の装置)を駆動するソフト/ハード。

取り込む▶(→インクルード)

トレース(～する)【trace】▶なぞること。とくに，プログラムの実行を追うこと。シングルステップ実行の意味。…85

に

ニーモニック【mnemonic】▶マシン語の命令の機能を簡潔に表現した略語。マシン語命令のアセンブリ言語における表現/呼び名/名前。…

38

ね

寝かせる▶フラグを偽の状態にすること。ビットを0にすること。
ネスティング【nesting】▶ネストすること。
ネスト(〜する)【nest】▶ループやサブルーチン呼び出しが入れ子になること。
寝る▶フラグが偽の状態になること。ビットが0になること。

は

ハードウェア割り込み▶(→割り込み)
排他的論理和【exclusive or】▶2つの真理値が等しければ0，等しくなければ1になるような論理演算。論理差。XOR。
バイト【byte】▶68000では8ビットのこと。…45
バイトエンディアン【byte endian】▶多バイトデータのメモリ上でのバイト順。68系プロセッサではアドレスの小さいほうに上位バイトがくるが，80系/86系ではアドレスの小さいほうに下位バイトがくる。…46
バイナリ【binary】▶2進。時に，テキストの対語。
バイナリファイル【binary file】▶Xファイルなど，typeコマンドで読むことのできないファイル。
パイプ【pipe】▶複数のプログラム間の(標準)出力と(標準)入力を直結する手段。…119
配列【array】▶同種のデータを連続するメモリ領域に並べたもの。
破壊(〜する)▶値/内容を変えること。メモリ内容/ディスク内容が不当に変わってしまうこと。
吐き出す▶出力すること。
バグ【bug】▶ソフトウェア/ハードウェアの不具合。
バグフィクス(〜する)【bug fix】▶見つかったバグを取り除くこと。
パケット【packet】▶通信内容をブロック単位で行うような(広義での)通信における，そのブロックのこと。
バージョン【version】▶第n版の〝版〟。
バージョン番号▶第n版の〝n〟。バージョン番号が大きく変わった直後の(バージョン番号小数点以下が0の)プログラムを使うのはなにかと怖い。
走らせる▶プログラムを実行すること。
バス【bus】▶装置間をつなぐ信号線。
パス(名)【path name】▶ドライブ名やディレクトリ名も含んだファイル名。時に，そのうちファイル名を除いた部分。
バスエラー【bus error】▶プロセッサ動作時のバスレベルでの障害。実装されていないメモリ領域にアクセスしようとしたときなどに発生する。
パック(〜する)【pack】▶(記憶域節約のため)データを詰めて格納すること。(←→アンパック)
バックアップ(〜する)【back up】▶いざというときの用心のために，ディスク内容やメモリ内容のコピーをべつに作って保存すること。また，その保存されたもの。
バックエンド【backend】▶プログラムのうち，とくにデータ処理を行う部分。(←→フロントエンド)
バックグラウンド処理▶現在実行中のプログラムの裏で平行して行われる処理。
バックスラッシュ【backslash】▶〝＼〟記号。JISキーボードにはこの文字がないので，〝¥〟で代用されることが多い。X68000ではSWITCH.Xで設定することで，〝¥〟を〝＼〟で表示することができる。
バックトラック【backtrack】▶(検索/推論の過程で)行き詰まったときに，後戻りすること。もしくは，1つの解が得られた後で，他の解を求めて後戻りすること。
バッチ(ジョブ)【batch job】▶1度に処理する仕事。とくに，シェルコマンドからなるプログラム。
バッチジョブ的▶あらかじめすべての指示を与えてから，いっきに処理すること。非対話的。…176

バッファ【buffer】▶緩衝記憶。入出力を円滑に行うために，データを一時的に溜め込む領域。
バッファリング(〜する)【buffering】▶入出力時に，バッファを使用すること。
パラメータ【parameter】▶本来は仮引数の意味。時に実引数の意味でも使われるが。
パラレル【parallel】▶平行，並列の意。時にパラレル伝送のこと。
パラレル接続▶(パラレル伝送を前提に)信号線を複数使って装置間をつなぐこと。
パラレル通信▶(→パラレル伝送)
パラレル伝送▶複数の信号線を使ってデータを伝送すること。
パリティ【parity】▶偶奇性。ビット列中の〝1〟であるビットが偶数個あるか，奇数個か，といったこと。
パリティビット▶データ伝送時に，データが正常に伝送できているかどうかを確認する指針としてパリティを偶数か奇数に揃える(それぞれ，偶数パリティ，奇数パリティ)ために付け加えられるビット。
半角文字▶横幅が天地の半分である書体の文字。1バイトコードで表現される文字(ANK)。
ハング(〜する)▶(→ハングアップ)
ハングアップ(〜する)【hang up】▶プログラムが暴走した状態を表現する言葉の1つ。何かの出力をするでもなし，キー入力を受けつけるでもなし，言葉どおり沈黙してしまった状態。ソフトだけではなく，ハードに対しても使う。
反転(〜する)▶ビットの0と1，条件の真と偽をひっくり返すこと。
ハンドアセンブル(〜する)▶アセンブラを使わずに，直接命令コードをメモリに書き込むことでマシン語プログラムを作成する方法。
ハンドシェイク【handshake】▶データ伝送時に，送信側と受信側が適当に応答しあうこと。
番人【sentinel】▶見張り番。アルゴリズムを実現する過程で処理を簡略化するために導入されるダミーデータ。
汎用性▶ソフトウェア/ハードウェアが，どれだけ多くの用途/場面で使えるかを意味する語。…198

ひ

引数【parameter, argument】▶サブルーチン等に渡す値。
非対話的【non-interactive】▶対話的でないこと。あらかじめプログラムにすべての指示を与えておき，いっきに処理する様子。バッチジョブ的。…176
ビット【bit：Binary digIT】▶2進1桁。0か1。偽か真。…45
ビット化け▶データ伝送時にノイズ等の影響で，データ中のビットが誤って伝わること。
ビットフィールド【bit field】▶短いビット長のデータをまとめたもの。記憶領域の節約のために用いられる。68000のccrは1ビットごとに意味を持つ複数情報を寄せ集めたビットフィールドになっている。
ビットマスク(〜する)▶ビット列中の特定のビット(部分ビット列)のみを操作する(多くの場合，0にする)こと。また，そのために使うビット列のパターン。
否定▶(→論理否定)
ヒープ【heap】▶(干し草の)山積みの意味。転じて，そんな感じにデータを積み上げておくようなデータ構造(大ざっぱにいうと，〝ソートされた二分木〟)。また，自由に使える空きメモリの山。
評価(〜する)【evaluate】▶(式等の)値を計算すること。
標準エラー出力【standard error output】▶オープン済みファイルハンドルの1つ。通常は画面(CON)に割り当てられている。COMMAND.Xからはリダイレクトできないので，エラーメッセージ等，リダイレクトされては困るようなメッセージの出力に用いられる。ファイルハンドル2。出力用。…142, 242
標準出力【standard output】▶オープン済みファイルハンドルの1つ。通常は画面(CON)に割り当てられているが，リダイレクションにより，任意のファイル/デバイスに切り替えることができる。ファイルハンドル1。出力専用。…30, 141
標準入力【standard input】▶オープン済みファイルハンドルの1つ。通常はキーボード

(CON)に割り当てられているが，リダイレクションにより，任意のファイル/デバイスに切り替えることができる。ファイルハンドル0。入力専用。…30，141

標準プリンタ【standard printer】▶オープン済みファイルハンドルの1つ。通常はプリンタ(PRN)に割り当てられている。ファイルハンドル4。出力専用。…142

標準補助装置【standard auxiliary device】▶オープン済みファイルハンドルの1つ。通常はRS-232Cポート(AUX)に割り当てられている。ファイルハンドル3。入出力可。…142

開く▶(→オープン)

ふ

ファイルエンドコード▶ファイルの終端を表すコード。Human68kのテキストファイルでは$1A_H$。

ファイルハンドル【file handle】▶Human68kのファイルシステムにおいて，ファイルオープン時に返される整数。オープン後は，このファイルハンドルによって入出力対象ファイルを指定し，読み書きを行う。…139

ファイル属性【file attribute】▶ふつうのファイルなのか，サブディレクトリなのかといったファイルの種類等を示す情報。ディレクトリ中にファイル名やファイルサイズ等とともに格納されている。ファイルアトリビュート。…138，252，254

ファミリ【family】▶(マイクロプロセッサと，その周辺チップからなる)シリーズとしての製品群。

ファームウェア【firmware】▶ハードの一部としてのプログラム。たとえば，マイクロプロセッサ内のマイクロプログラムや，拡張スロットに差すボード上に乗ったROMに内蔵されたプログラムなどをいう。時には，X68000のIOCS ROMのような，ROMで供給される低レベルルーチン集のこともファームウェアという。

ファンクション【function】▶関数。機能。

フィルタ【filter】▶なんらかのデータを受け取り，加工して出力するもの。狭義では，標準入力から読み込んだテキストデータを加工して，標準出力に書き出すプログラム。…118

フェッチ(〜する)【fetch】▶プロセッサがメモリから命令コードを読み込むこと。

フォント【font】▶同一書体の文字の1セット。

不可視(属性)【hidden】▶DIRコマンドでは表示されないことを示すファイル属性。

副作用▶ある処理/操作に付随して起こる事象/現象。時に，副作用はそれ自身が目的となりうる。たとえば，Cのprintf関数は出力した文字数を返すという〝関数〟としての機能よりも，書式付きで文字列や数値を表示するという副作用のほうが有用である。また，68000のcmp命令は〝減算を行うが，結果の値は捨てる，という演算〟を行う命令であるが，副作用としてccrに減算結果(=比較結果)が残る。

福袋▶初代X68000のシステムディスクにのみ存在したディレクトリ。このディレクトリにAS.X，LK.Xの初期バージョンがこっそり収められていた。

(THE)福袋V2.0▶Human68k上のマシン語プログラム開発セット。

符号拡張▶符号付き数をより長いビット長のデータに変換する際に，(変換によって追加される)上位ビットを(変換前のデータの)符号ビットで埋めること。…98

符号付き数▶正負の情報を持った数。

符号ビット【sign bit】▶数値データの符号(正か負か)を示すビット。通常使われている2の補数表現(2の補数表示)による符号付き数においては，最上位ビットが符号ビットとなる。…88

復帰(〜する)【carriage return】▶カーソルを行頭に移動すること。

復帰(〜する)【restore】▶待避しておいたレジスタ等の内容を元に戻すこと。

復帰(〜する)【return】▶プログラム/サブルーチンから呼び出し元に戻ること。

復帰コード▶カーソルを行頭に戻すことを意味するコントロールコード。CRコード(ASCIIコード$0D_H$)。

プッシュ(〜する)【push】▶スタックにデータを追加すること。積む，ともいう。…22

プッシュバック(〜する)【push back】▶(バッファから)読み込んだデータを(元のバッフ

ァに)押し戻すこと。

物理的▶実体をともなうこと。

浮動小数点数▶実数データのコンピュータ内部表現の1つにして，もっとも広く使われている形式。限りあるビット長でなるべく有効数字を多くとることができるように，数値をx×2のn乗の形で表現する。ここでxを仮数部，nを指数部と呼ぶ。

不当命令▶68000において，意味/機能の割り振られていない命令コードのうち，$Axxx_H$，$Fxxx_H$以外のこと。

フラグ【flag】▶ある特定の状態が成り立っているかどうか(真か偽)を保持する変数。

ブラックボックス【black box】▶中身がどうなっているかは外側からはわからないが，とにかく，なんらかの動作を行う存在。

フラッシュ(〜する)【flush】▶バッファを空にすること。

プリデクリメントアドレスレジスタ間接(形式)【address register indirect with predecrement】▶68000のアドレッシングモードの1つ。ふつうのアドレスレジスタ間接形式の動作に先立って，ポインタとして使うアドレスレジスタの値をオペレーションサイズ分減じるモード。アセンブリ言語では〝−(a0)〟のように表記する。…48，49

プリプロセッサ【preprocessor】▶メインの処理に先立って，なんらかの前処理を加えるもの(プログラム)。

フルスクリーンエディタ▶画面志向のテキストエディタ。

フルパス(名)▶ドライブ名〜ディレクトリ名〜ファイル名まで，いっさい省略なしに完全な形で記述/指定したパス名。

ブレイク(〜する)【break】▶プログラム/処理が中断すること。中断させること。とくに，^CやBREAKキーを押すことによってプログラムの実行を強制終了すること。

ブレイクチェック【break check】▶(Human68kがDOSコール実行時に)BREAKキーが押されているかどうかをチェックすること。…29，70，161

フレームポインタ【frame pointer】▶スタックフレームの基準アドレスを保持するポインタ。…210

プログラマブル【programable】▶固定動作しかしないのではなく，プログラムによって動作を変えることができることを指す言葉。

プログラムカウンタ【program counter】▶次に実行すべき命令が格納されたアドレスを保持するレジスタ。…35，36

プロセス【process】▶一連の処理工程。OSにおける基本的な処理単位。…224

プロセッサ【processor】▶なんらかの加工/処理を行うもの(ソフト/ハード)。狭義ではCPUと同義。

ブロックストレージセクション【BSS：Block Storage Section】▶プログラム中，主として初期値なしの作業用メモリ領域を置く部分。AS.Xでは.bss疑似命令によって，以降をブロックストレージセクションに置くことを宣言する。…160，219

ブロックデバイス【block device】▶ディスク装置等，まとまったブロック単位でランダムに入出力を行うことのできるデバイス。Human68kにおいては，〝A:〟や〝B:〟等のドライブとなりうるデバイス。

プロトコル【protocol】▶通信規約。データ伝送時の送信側と受信側のソフト的/ハード的な約束事。

フロる▶オーバーフローすること。

フロントエンド【frontend】▶プログラムのうち，とくにユーザーインターフェイス部分，データの入力/前処理を行う部分。(←→バックエンド)

プロンプト【prompt】▶入力を促すもの。たとえば，COMMAND.Xでは入力を受けつけるときに〝A>〟といった表示が出るが，これがプロンプト。

負論理▶0(電圧の低い状態)で真を，1(電圧の高い状態)で偽を表すこと。

分岐(する)▶処理の流れを変えること。処理が枝分かれすること。…40，332

へ

ベクタ【vector】▶なんらかの処理ルーチンを

指すポインタ(としての意味をもつメモリ)。ベクトル。…338

ベースアドレス【base address】▶アドレス指定の基準となるアドレス。

ベタフォーマット▶元のイメージそのまま細工なしに記録する方式を(多少蔑みつつ)いう俗語。

ヘッダ【header】▶(プログラムの)頭書き。(ファイル等の)頭に置かれる情報。

ヘッダファイル▶プログラミング言語において,各種宣言/定義を行うファイル。ソースファイルにインクルードして使われる。

ベリファイ(〜する)【verify】▶ディスク等にデータが正しく書き込まれているかどうかを確認するために,書き込んだデータと書き込む前の元データを比較すること。

変数【variable】▶プログラム中で値が変化するデータ。概念上は値を格納しておく容器。

ほ

ポインタ【pointer】▶データを指し示すもの。データが格納されたアドレスを保持する変数/レジスタ。

ポイント(〜する)【point】▶データを(その格納アドレスによって)指し示すこと。

暴走(〜する)▶ソフトウェア/ハードウェアが〝規定外の動作をすること。また,そういう状態。

保証(〜する)▶たとえば,あるサブルーチンからの戻り値が必ず正であると決まっているのであれば,〝戻り値は正であることが保証されている〟という。逆に,メインルーチンが正の戻り値を期待しているときに,サブルーチン側で戻り値が必ず正にするように細工すれば,〝戻り値が正であることを保証する〟ことになる。

ポストインクリメント・アドレスレジスタ間接(形式)【address register indirect with postincrement】▶68000のアドレッシングモードの1つ。ポインタとして使うアドレスレジスタの値をオペレーションサイズ分進めてから,アドレスレジスタ間接形式の動作をするモード。アセンブリ言語では〝(a0)+〟のように表記する。…48, 49

ポータビリティ▶(→移植性)

ポップ(〜する)【pop】▶スタックからデータを取り出すこと。…23

ボリューム名【volume name】▶ディスクにつけられる名札。Human68kのディスク中では,〝ボリューム名であるという属性〟をもった実体のないファイルとして記録されている。

ポーリング(〜する)【polling】▶割り込みによって変化するワークエリアや,I/Oポートの内容の変化を(定期的に読み込んで)監視すること。ネットワークのホストが,各端末に送信データがあるかどうか尋ねて回る(比喩!)こと。

ホワイトスペース【white space】▶空白。スペースキャラクタ(ASCIIコード20_H)だけではなく,タブ(ASCIIコード09_H)などもひっくるめて表現する言葉。

翻訳(〜する)▶なんらかのプログラミング言語で書かれたソースプログラムを他言語やマシン語や中間コード等に変換すること。

ま

マージ(〜する)【merge】▶複数の(ソート済み)データ列/ファイルなどを併合して1つにまとめること。

マイクロプロセッサ【micro processor】▶CPUとしての機能を1チップにまとめたもの。

マクロ【macro】▶複数の命令をまとめて一命令のように見せかけたもの。

マクロ定義▶マクロを定義すること。

マジックナンバー【magic number】▶ソースプログラム中に埋め込まれた生の数字のことをいう。

マシン【machine】▶コンピュータ界では,コンピュータ本体を意味する。

マシンコード【machine code】▶マシン語命令のコード。

マシン語【machine language】▶プロセッサが唯一理解できる(広義の)言語。機械語。

末端再帰▶定義の最後で再帰していること。サブルーチンの最後で自分自身を呼び出すこと。

マッピング(〜する)【mapping】▶割りつけ

ること。アドレスを振ること。
マネージャ【manager】▶特定の分野の管理を行うもの(プログラム)。
魔法の数字▶(→マジックナンバー)
マルチタスク【multi task】▶同時に複数の処理を平行して行うこと。もしくは，そのように見せかけること。
回る▶ループする。

み

未実装命令▶68000において，意味/機能の割り振られていない命令コードのうち，$Axxx_H$, $Fxxx_H$のことをとくにこう呼ぶ。
未満▶x＜nのとき，xはn未満であるという。数学/コンピュータの世界では，〝以下〟とははっきり区別して用いられる。

む

無限ループ▶脱出条件がないループ。永遠に回り続けるループ。
無条件分岐▶つねに分岐すること。…40
無符号数▶符号のついていない(つねに正の)数。演算時に符号を考慮する必要のない数。

め

明示(〜する)▶はっきりと示すこと。
メインメモリ【main memory】▶コンピュータが積んでいるメモリのうち，VRAMやROMなどを除いた領域。
メインルーチン▶主プログラム。
メモリ【memory】▶記憶(装置)。
メモリイメージ【memory image】▶データのメモリ上での姿。
メモリ管理ポインタ▶Human68kにおいて，メモリ管理を目的としてメモリブロックの先頭に置かれる双方向リスト構造をしたデータ。
メモリブロック【memory block】▶一連のメモリ領域のこと。狭義では，OSに要求して確保したメモリ領域のひとつひとつを指す。
メモリマップドI/O【memory mapped I/O】▶メモリ空間に割りつけられたI/O。

も

文字コード【character code】▶文字と1対1に対応した数値。
文字属性【character attribute】▶文字の表示色や強調/反転等の指定の総称。文字アトリビュート。…63
文字化け▶文字データ伝送時に生じたビット化けによって，結果として誤った文字が伝わること。時には，プログラムのバグによって処理データ中の文字コードが不当に変化してしまうこと。
モジュール【module】▶ある処理を行うプログラムのひとまとまり。狭義では(ライブラリとしての)オブジェクトファイルのこと。
文字列【string】▶文字の列。コンピュータ内部では連続したメモリ領域に文字コードを並べたもの(文字コードの配列)として表現される。…92
戻り値▶プログラム/サブルーチン/関数から呼び出し元に返される情報。

ゆ

ユーザーインタフェイス【user interface】▶ハード/ソフトとユーザーとの接点/界面。
ユーザースタックポインタ【user stack pointer】▶68000において，ユーザーモードで使用されるスタックポインタ。usp。
ユーザーバイト【user byte】▶srの下位バイト(=ccr)。
ユーザーモード【user mode】▶68000の走行モードの1つ。おもにユーザープログラムを実行するときのモード。…80

よ

要求(〜する)【request】▶下位のルーチン/

OS等に処理をさせること。また，データを送ってよこすように頼むこと。

ら

ライブラリ【library】▶狭義では，オブジェクトファイルを1つのファイルにまとめたもの。…221

ライブラリアン【librarian】▶ライブラリの作成/保守を行うツール。…221

ラインエディタ【line editor】▶行単位での編集を基調とするテキストエディタ。

ラッチ(〜する)【latch】▶デジタル回路中で，その瞬間のデータを(内部のレジスタに)保持すること。また，その保持するための小メモリ。

ラベル【label】▶あるものを間接的に表現するのに用いられる文字列/語。…37，38

乱数(列)▶相互に依存関係/規則性のない(ように見える)数字の列。

ランダム【random】▶順序が決まっていない(制限がない)こと。無作為。

り

リエントラント【re-entrant】▶再入可能と訳される。同じプログラム/サブルーチンを〝終了前に二重に呼び出すことができる〟という意味。たとえば，マルチタスクのシステムでは，あるサブルーチンが呼び出されてリターンする前に，よそから同じサブルーチンが呼び出されるといったことが起こる。このとき，サブルーチン内で使う変数/ワークがメモリの固定アドレスに置かれていたりすると，2つの呼び出しが干渉しあい，異常動作することになる(そのサブルーチンはリエントラントではない)。また，シングルタスクのシステムであっても，再帰的に呼び出されるサブルーチンでは，やはり同様の問題が生じる。プログラムがリエントラントであるためには，最低限，ワークをスタック上のローカルな領域に確保する必要があるわけである。

リカーシブコール【recursive call】▶(→再帰呼び出し)

リカージョン【recursion】▶(→再帰)

リクエスト(〜する)【request】▶(→要求)

リクエストヘッダ【request header】▶デバイスドライバにコマンドを渡すのに使われるメモリ領域。

リスト【list】▶同種のデータを(配列のように連続メモリに置かれることによってではなく)ポインタでつなぐことによって順序づけたデータ構造。また，プログラムリスト。一覧表。

リストファイル▶アセンブリ言語で書いたプログラムに，どのようなコードにアセンブルされたかという情報が付加されたもの。AS.Xでは/Pスイッチをつけることでカレントディレクトリに*.PRNのファイル名でリストファイルが作成される。

リセット(〜する)【reset】▶初期状態に戻すこと。とくに，RESETスイッチを押すなどして，再起動すること。ビットを0にすること。

リターン(〜する)【return】▶サブルーチン等から呼び出し元に復帰すること。また，サブルーチン等から呼び出し元に戻り値を返すこと。

リダイレクション【redirection】▶リダイレクトすること。

リダイレクト(〜する)【redirect】▶標準入出力をファイルやキャラクタデバイスに切り替えること。…30，31，118

リニア【linear】▶直線的な。線形の。デコボコでもズタズタでもない。本書では「68000のメモリ空間はリニアで云々」といった文脈で用いているが，メモリ空間とはもともとそうしたものであって，8086のようにズタズタになっているほうがおかしい。

リネーム(〜する)【rename】▶(ファイルの)名前を付け替えること。

リフレッシュ(〜する)【refresh】▶メモリチップは要するにコンデンサの集合であり，電荷が蓄えられているかどうかでビットの0/1を表している。コンデンサは時間がたつにつれ徐々に放電してしまうから，一定時間ごとに再充電してやらないとメモリ内容が失われてしまう。この，再充電作業をリフレッシュという。

リロケータブル【relocatable】▶ロードアドレスに依存しないこと。プログラムが，どのアドレスに置いても動作すること。再配置可能。

…327

リロケータブルオブジェクト(ファイル)【relocatable object file】▶(リンクされていない)まだアドレスが確定していないオブジェクトファイル。

リンカ【linker】▶複数のオブジェクトファイルをつなぎまとめ，実行ファイルを作るプログラム。

リンク(〜する)【link】▶つなぐこと。とくに，ポインタによってデータ間を関連づけること。また，アセンブル後のオブジェクトファイルをつなぎまとめて実行可能ファイルを作成すること。…25

リングバッファ【ring buffer】▶バッファの先頭と末尾が(論理的に)つながっているような形式のバッファ。キューを実現するのに用いられる。…316

リンクポインタ【link pointer】▶同種の情報が複数あるときに，続くデータを指し示すポインタ。データ間をつなぐポインタ。

る

ルーチン【routine】▶プログラム(の部分)。

ループ【loop】▶処理を繰り返すこと。また，繰り返し処理を行うプログラム構造。

ループカウンタ【loop counter】▶ループ回数を数える変数/レジスタ。

れ

例外【exception】▶(ソフトウェア実行過程で生じる)なんらかのハード的な障害。68000ではバスエラーや，アドレスエラー，0による除算，ハードウェア割り込み等を総称して例外と呼ぶ。…338

例外処理▶例外が発生したときに実行される(そのために準備された)処理。…338

例外ベクタ▶例外処理ルーチンの先頭アドレスを保持する特定のメモリ。…338

例外ベクタ番号▶バスエラーは2，アドレスエラーは3というように，68000の例外に振られた番号。4倍することで例外ベクタアドレスが得られる。…338

レコード【record】▶ひとまとまりのデータ。ディスクを読み書きするときの論理的な単位。Cでいう構造体のPascalでの呼び名。

レジスタ【register】▶プロセッサ(や，その他の周辺チップ)内に用意された小規模なメモリ。…35，36

ろ

ローカル▶(→局所的)

ローカルエリア【local area】▶局所的，かつ一時的な作業用のメモリ領域。とくに，68000においては，link命令でスタック上に確保され，unlk命令で解放されるメモリ領域。…212

ローカルストレージ【local storage】▶(→ローカルエリア)

ロケーションカウンタ【location counter】▶アセンブラがアセンブル時にかりにふるアドレス。AS.Xの場合はセクションごとに別々のロケーションカウンタをもつ。

ロジック【logic】▶(→論理)

ローテート(〜する)【rotate】▶ビット列/データ列の先頭と末尾がつながっているものとして，回転するようにずらすこと。…266

ロード(〜する)【load】▶値/データを内/外部の記憶装置から読み込むこと。狭義では，メモリからレジスタにデータを転送すること。

ロングワード【long word】▶68000では32ビットのこと。…45

論理▶コンピュータの動作原理。ロジック。

論理演算▶真理値で演算すること。論理積，論理和等の総称。

論理差▶(→排他的論理和)

論理積▶〝〜かつ〜〟の意味。AND。

論理的▶そう見なすこと。〝概念上〟，〝仮想的〟，〝理屈のうえでは〟等の意味。

論理否定【not】▶条件の真偽を反転すること。否定。

論理和▶〝〜または〜〟の意味。OR。

わ

ワーク▶(→ワークエリア)

ワークエリア【work area】▶プログラム動作時の作業用に使用するメモリ領域。

ワード【word】▶語。68000では16ビットのこと。…45

ワード境界整合【word alignment】▶68000では，ワード/ロングワードでアクセスする命令/データは，必ず偶数アドレスに置かれていなければならないわけであるが，そうなるようにアセンブラ/コンパイラ/プログラマが調節すること。…214

割り込み【interrupt】▶(おもに外部デバイスからの信号によって)プロセッサが現在実行中の処理を一時的に中断して，特定のルーチンに制御を移すこと。

割り込みルーチン【interrupt routine】▶割り込み発生時に実行される(例外処理)ルーチン。また，Human68kのデバイスドライバの入出力処理ルーチン本体。

悪さ▶作法/常識/道徳から外れたこと。

Human68kバージョンアップ履歴（1990年11月現在）

表はHuman68kのシステム/開発ツールに含まれるプログラム/ファイルの一覧である。上段がタイムスタンプ，下段がファイルサイズを意味している。どの程度の資料的価値があるのかはわからない。ただ，前からこんな表がほしかった。出荷直後にバグに気づいた慌てぶりや，開発ツールがアセンブラからCに切り替わる様子等，メーカーの内情を垣間見るという点では，なかなか興味深いものがある。

ちなみに，初代X68000ユーザーも，Human68k ver.2.0，SX-WINDOW，C compiler PRO-68K ver.2.0を購入すれば，ほぼフルセットの最新のシステムを手に入れることができる。しめて，61,400円なり(税抜き定価)。

●表の見方

初代(a)　→初代X68000同梱システムディスク
初代(b)　→初代X68000同梱システムディスク(後期バージョン)
XC V1.00　→C compiler PRO-68K
ACE　→X68000 ACE/ACE-HD同梱システムディスク
XC V1.01　→C compiler PRO-68K後期バージョン(登録ユーザーに無償で配布された版)
PRO　→X68000 PRO/PRO-HD/EXPERT/EXPERT-HD同梱システムディスク
HumanV2　→Human68k ver.2.0パッケージ版
PROII　→X68000 PROII/PROII-HD/EXPERTII/EXPERTII-HD同梱システムディスク
SUPER-HD　→X68000 SUPER-HD同梱システムディスク
SX-WINDOW　→SX-WINDOWパッケージ版
XC V2.0　→C compiler PRO-68K ver.2.0最初期バージョン
→　→バージョンアップはなかった

ファイル名＼	初代(a)	初代(b)	XC V1.00	ACE	XC V1.01	PRO	HumanV2	PRO II	SUPER-HD	SX-WINDOW	XC V2.0
HUMAN.SYS	87-03-15 44650	87-05-15 44660	87-11-03 44660	→	→	89-02-10 54174	89-04-04 54174	→	90-05-05 54240	→	→
AR.X			87-11-03 4352		→						→
AS.X	87-03-15 28062	87-05-15 28062	87-11-03 28194		88-04-02 28194						90-05-05 99572
ASK68K.SYS	87-03-15 106036	87-07-29 107070	87-11-03 107070	→	→	89-02-10 121360	89-04-04 121470	→	→	→	→
ATTRIB.X	87-03-15 922	87-05-15 922		→		→	→	→	→		
AUDIO.FNC	87-03-15 540	87-05-15 540	87-11-03 668	→	→	→	→	→	→		→
BACKUP.X				88-02-05 31046		89-02-10 35250	→	→	→		
BASIC.X	87-03-15 46362	87-05-15 46290	87-11-03 39818	→	→	89-02-10 39940	→	→	90-05-05 39940		→
BASLIB.A (BASLIB.L)			87-11-03 28684		88-05-15 28728						90-05-05 49704
BC.X			87-11-03 18496		88-05-15 18584						90-05-05 79904
BIND.X						89-02-10 7468	→	→	→		→
BUILTIN.LB								90-03-15 40366	90-05-15 44938	90-06-15 45582	
CASE.X			87-11-03 2022		→						→
CASH.X			87-11-03 3570		→						
CC.X			87-11-03 12510		88-05-15 12562						90-05-05 403974
CC0.X			87-11-03 35220		88-05-15 36098						
CC1.X			87-11-03 80090		88-05-15 81960						
CC2.X			87-11-03 23638		88-05-15 24502						
CCP.X			87-11-03 20992		88-05-15 20942						
CHKDSK.X	87-03-15 2298	87-05-15 2298		→		89-02-10 2568	→	→	→		
CLIB.A (CLIB.L)			87-11-03 59650		88-05-15 59738						90-05-05 114566
COMMAND.X	87-03-15 24736	87-05-15 24736	87-11-03 24736	→	→	89-02-10 28004	89-04-04 28022	→	90-05-05 28026		→
COPY2.X	87-03-15 2632	87-05-15 2632		→		89-02-10 36220	89-04-04 38056	→	→		
COPYALL.X						89-02-10 2134	→	→	90-05-05 2210		→
CTRLPNL.LB								90-03-15 112482	90-05-15 122368	90-06-15 113300	
CUSTOM.X	87-03-15 1744	87-05-15 1744		→		89-02-10 44348	89-04-04 44432	→	90-05-15 44692		
CV.X			87-11-03 2326		→						90-05-05 17570

ファイル名＼	初代(a)	初代(b)	XC V1.00	ACE	XC V1.01	PRO	HumanV2	PRO II	SUPER-HD	SX-WINDOW	XC V2.0
DB.X			87-11-03 25522		88-05-15 25518						90-05-05 31702
DEFSPTOOL.BAS (DEF.BAS)	87-03-15 34382	87-05-15 34382	87-11-03 34421	→	→	89-02-10 34394	89-04-04 34393	→	→		90-05-05 34352
DICM.X	87-03-15 36192	87-05-15 36284		→		89-02-10 38552	→	→	→		
DISKCOPY.X	87-03-15 2384	87-05-15 2384	87-11-03 2426	→	→	89-02-10 28144	→	→	90-05-15 34574		90-06-15 34578
DOSCALL.MAC			87-11-03 1836		88-05-15 1908						90-05-05 2708
DOSLIB.A (DOSLIB.L)			87-11-03 17084		88-05-15 17092						90-05-05 44256
DRIVE.X			87-11-03 2218	→	→	89-02-10 3292	→	→	90-05-15 3370		→
DUMP.X	87-03-15 1050	87-05-15 1050		→		89-02-10 1416	→	→	→		
ED.X	87-03-15 35608	87-05-15 35638	87-11-03 35658	→	→	89-02-10 35748	→	→	90-05-05 35746		→
FC.X	87-03-15 3468	87-05-15 3468		→		89-02-10 3588	→	→	→		
FIND.X	87-03-15 3422	87-05-15 3422		→		89-02-10 3492	→	→	→		
FLOAT1.X			87-11-03 11498	→	→	→	→	→	→	→	→
FLOAT2.X			87-11-03 12766	→	→	→	→	→	90-05-05 12844	→	→
FLOATEML.L											90-05-05 88462
FLOATFNC.L											90-05-05 33100
FORMAT.X	87-03-15 10392	87-05-15 10390	87-11-03 10416	88-02-05 13190	→	89-02-10 53202	89-04-04 53430	→	90-05-15 88492		90-06-15 95364
FSX.X								90-03-15 170052	90-05-15 180610	90-06-15 180998	
GRAPH.FNC	87-03-15 35134	87-05-15 35294	→	→	→	→	→	→	→		→
HDID.X									90-05-15 17488		
HDフォーマット.X									90-05-15 76876	90-06-15 83362	
HENWIN.X								90-03-15 20320	→	→	
HISTORY.X						89-02-10 27808	89-04-04 27830	→	→		→
ICON.LB								90-03-15 61418	90-05-15 71342	90-06-15 71494	
IMAGE.FNC	87-03-15 7450	87-05-15 7450	87-11-03 8974	→	→	→	→	→	→		→
IOCS.X								90-03-15 14056	90-05-15 14412	90-06-15 14420	→
IOCSCALL.MAC			87-11-03 3420		88-05-15 3493						90-05-05 3648
IOCSLIB.A (IOCSLIB.L)			87-11-03 22910		88-05-15 22910						90-05-05 45932

ファイル名＼	初代(a)	初代(b)	XC V1.00	ACE	XC V1.01	PRO	HumanV2	PRO II	SUPER-HD	SX-WINDOW	XC V2.0
KEY.X	87-03-15 2714	87-05-15 2714		→		→	→	→	→		
LIB.X											90-05-05 26352
LK.X	87-03-15 10220	87-05-15 10220	87-11-03 10302		88-04-02 10306						90-05-05 42598
MAKE.X											90-05-05 55368
MENU.X						89-02-10 40912	89-04-04 41750	→	→		
MESH.C			87-11-03 1154		→						90-05-05 884
MORE.X	87-03-15 1614	87-05-15 1614		→		→	→	→	→		
MOUSE.FNC	87-03-15 926	87-05-15 926	→	→	→	→	→	→		→	87-11-03 926
MOVE.X			87-11-03 2030	→	→	→	→	→	→		→
MUSIC.FNC	87-03-15 21528	87-05-15 21528	87-11-03 2628	→	→	→	→	→	→		
MUSIC2.FNC											90-05-05 3974
OPMDRV.X			87-11-03 26192	→	→	89-02-10 26294	→	→	90-05-15 26392	→	
OPMDRV2.X											90-05-05 35798
PCMDRV.SYS	87-03-15 416	87-05-15 416	→	→	→	→	→	→	→	→	→
PR.X	87-03-15 2992	87-05-15 3014		→		89-02-10 3178	→	→	→		
PRINT.X											90-05-05 8458
PRNCNF.X	87-03-15 1716										
PRNDRV.SYS	87-03-15 1816	87-05-15 1816	→	→	→	89-02-10 1816	→	→	→	→	→
PRNDRV1.SYS		87-05-15 3566	→	→	→	→	→	→	→	→	→
PRNDRV2.SYS		87-05-15 1816	→	→	→	→	→	→	→	→	→
PRNDRV3.SYS		87-05-15 1816	→	→	→	→	→	→	→		→
PROCESS.X			87-11-03 698		→	89-02-10 2438	→	→	→		→
RAMDISK.SYS	87-03-15 1816	87-05-15 1816	→	→	→	89-02-10 1816	→	→	→	→	→
RECOVER.X				88-02-05 24606		89-02-10 29276	→	→	→		
RESTORE.X				88-02-05 28524		89-02-10 32082	→	→	→		
RSDRV.SYS								88-06-10 3232	→	→	
SCD.X											90-05-05 77746

ファイル名＼	初代(a)	初代(b)	XC V1.00	ACE	XC V1.01	PRO	HumanV2	PRO II	SUPER-HD	SX-WINDOW	XC V2.0
SCSIDRV.SYS									90-05-15 1234		90-06-15 1274
SORT.X	87-03-15 2074	87-05-15 2074		→		→	→	→	→		
SPEED.X	87-03-15 1084	87-05-15 1084		→		89-02-10 24230	89-04-04 24244	→	→		
SPRITE.FNC	87-03-15 2656	87-05-15 2656	→	→	→	→	→	→	→		→
SRAMDISK.SYS	87-03-15 924	87-05-15 924	→	→	→	→	→	→	→	→	→
STICK.FNC	87-03-15 352	87-05-15 352	→	→	→	→	→	→	→		→
SUBST.X						89-02-10 1654	89-04-04 1806	→	→		
SWITCH.X	87-03-15 4078	87-05-15 4078		88-02-05 4704		89-02-10 46504	89-04-04 47108	→	90-05-15 52196		
SXWIN.X								90-03-15 11940	90-05-15 13894	90-06-15 14436	
SYS.X	87-03-15 764	87-05-15 764		→		89-02-10 1012	→	→	90-05-15 1022		
SYSTEM.LB								90-03-15 120770	90-05-15 122164	90-06-15 88990	
TAR.X			87-11-03 1524		→						
TERM.X			87-11-03 1424		→						→
TIMER.X						89-02-10 17046	→	→	→		
TITLE.X								90-03-15 23334	90-05-15 23334	90-06-15 23342	
TOUCH.X			87-11-03 2760		→						90-05-05 2766
TREE.X			87-11-03 840	→	→	89-02-10 884	→	→	→		→
USKCGM.X	87-03-15 10884	87-05-15 10900		88-02-05 10924		→	→	→	→		
VS.X	87-03-15 92832	87-05-15 92682	→	88-02-05 94850	→	89-02-10 97348	89-05-15 97540	→			
WHERE.X			87-11-03 1472	→	→	89-02-10 1526	→	→	→		→
WP.X	87-03-15 231052	87-07-01 231948		88-02-05 231948		89-02-10 232288		→	→		
アイコン情報.X								90-03-15 3564	90-05-15 3598	→	
カレンダー.X								90-03-15 8158	90-05-15 7826	90-06-15 7798	
キャンバス.X								90-03-15 2096	90-05-15 2112	→	
コントロール.X								90-03-15 11462	90-05-15 11540	90-06-15 11598	
サウンド.X								90-03-15 8580	90-05-15 7638	90-06-15 7610	
スイッチ.X								90-03-15 6854	90-05-15 7512	90-06-15 7484	

ファイル名＼	初代(a)	初代(b)	XC V1.00	ACE	XC V1.01	PRO	HumanV2	PRO II	SUPER-HD	SX-WINDOW	XC V2.0
タイプ.X								90-03-15 25850	90-05-15 26432	90-06-15 26804	
ダンプ.X								90-03-15 25192	90-05-15 25368	90-06-15 25718	
ノート.X								90-03-15 49786	90-05-15 63620	90-06-15 63648	
ピンボール.X								90-03-15 161304	→	→	
暁子.X								90-03-15 18470	90-05-15 18674	→	
時計.X								90-03-15 14154	→	→	
電卓.X								90-03-15 15670	90-05-15 15734	90-06-15 15734	
背景設定.X								90-03-15 4124	→	→	

●補足

1) サンプルは『Oh!X』編集部に保管されていたシステムのマスターディスクである。この他にも当然マイナーなバージョンアップ版があると思われる。

2) ディスク中の全ファイルを網羅しているわけではなく，BEEP.SYS等のシステム環境関連ファイル，コンフィグレーションファイル，ヘルプファイル，辞書ファイル，サンプル，また，CやBC.X用のヘッダファイル類，そしてグラディウスは削ってある。

3) 中には同内容のファイルに別のタイムスタンプがつけられている場合がある。たぶん，その逆はない。

4) ASK68K.SYSには，表に挙げたもの以外にも，

87-03-15(106004バイト)
87-07-22(107028バイト)

が確認されている。

5) VS.Xには，表に挙げたもの以外に，パッケージ版Human68k ver.2.0の最初期バージョンに含まれていた

89-04-04(97428バイト)

がある。

6) FORMAT.X, HDフォーマット.Xの最新バージョン(1990年11月現在)は，

FORMAT.X　90-09-01(95664バイト)
HDフォーマット.X　90-09-01(83472バイト)

であり，これはSCSIボードの付属ディスクに含まれる。

あとがき

あるいは言い訳の山

さてと。後はこのページを埋めさえすれば〝連載単行本化時における例外処理〟から抜けることができる。誰彼かまわず「ざまーみろ」と叫んでまわりたい気分だ。あ，自棄に適切な感情表現だな。も１度使お。ざまーみろっ，と。

ピンポーン。正解。たしかに僕はいまハイになってる。不満は山ほどあっても，とりあえずはじめての本だ。作業がほとんど終わったとくれば，浮かれもするさ。しかも，じわじわと得体のしれない不安が背中を這い上がってくるから，なおさら空騒ぎしたくなる。そう，怖いんだよ，本当は。

ブブー。はずれ。売れるかどうかなんてどうでもいい。そうだな。10，いや，１冊だけは売れてほしい(格好つけじゃないぞ)。で，そのただ１人の読者が一所懸命読んでくれる，と(誤解だって)。僕の野心なんてこんなものだ(信じてよ)。ああ，本当にそうならないかな(違うのに)。だってさ，こういうシチュエーションって，どことなく間の抜けた愁いがあったりなんかもして……かっこいい，じゃない?(オチは見えてたね)

そして，僕の不安は，この〝ただ１人の読者〟の一所懸命さに報いることができるかどうかに向けられる。

文章全般を支配する妙な重苦しさ，緊張感に押し潰されるんじゃないか，とか，大上段なとこや青くさいとこが鼻について途中でいやになるんじゃないか，とか，舌っ足らずな説明に頭を抱えるんじゃないか，とか，サンプルプログラムにバグが潜んでいるんじゃないか，とか，そもそも，この本を読んでマシン語プログラムが書けるようになるんだろうか，とかとか，心配の種はつきない。いましかないからまとめて言い訳しておこうか。若干本文のノリに戻りつつ，順不同でいく。

大上段な点については深く反省している。とくに冒頭のあたりは連載開始直後の気負いが露骨に出ていて自分で読んでいて赤面してしまうほどだ。でも，あえてそのまま残しておくことにした。何年も後になってから読み返して，かつての自分の青さを笑ってやりたいという，たぶんに個人的，感傷的な理由による(何にも変わっていなかったりして)。

文章の重さは僕のパワーバンドの狭さに起因する。ある程度自分を追い詰めないとまともな

(?)文章が書けないくせに，少し余分にプレッシャーがかかると，あっという間に煮詰まってしまうのだ。そのぎりぎりの感じが，素直に文章に反映されている。僕の頭の中には〝へらへらモード〟もあるにはあって，そっちのパワーバンドはかなり広いんだけど，この本はそれで書けるものじゃない(いきなり『Oh!X』のバックナンバーを開いてへらへらモードで書いた記事を探したりしないよーに)。

そういや，煮詰まって連載に2度ほど穴をあけたっけ。出版界用語の「著者急病につき〜」の意味をご存じでない正直な読者の方々からお見舞いのメッセージをいただき，たいへん恐縮した記憶がある(それとも，あれは皮肉だったのかな?)。だいたい，僕は風邪をひくと感覚が麻痺する分かえってプレッシャーには強くなる性質で，生まれてはじめて40度越えの熱を出したときにも平気で原稿を書いていた。もっとも，あの回は他の要因もあって数字の間違いがやたら多く，ひどい出来だったけれど（この本では直してある，はず)。

間違いの話が出たところでプログラムのバグについて。連載中，もっと発生するだろうと覚悟していたバグ君だが，なぜかほとんど出なかった。嫌われたのかもしれない。それならそれでいいけれど(でも，ちょっと寂しい)，まだ隠れている可能性のほうが高い。いちおう，実用プログラムの類は僕自身が日常的に使っているから，かなり安心だ，とは思う。気休めまでに補足しておくと，6章で作ったUPPER.Xはあれをベースにしたフィルタをいくつか作って毎日酷使しているし，A章のFILELIST.Xのメインルーチンもずいぶん使い回したが，問題は生じていない。B章のDEVICE.Xにもバグは出ていないし，TAPDRV.SYSは作って以来，組み込みっ放しで自分で作ったことも忘れて使い続けているが，うまく動いているようだ。

えーと，それから説明が不親切気味な点。この点については，あまり言い訳するつもりはない。開き直りにとられるかもしれないが，9割方は故意にその線を狙っている，といっておこう。残り1割は，純粋に僕の力不足による。というわけで，濃度10%のごめんなさい。

そして，この本を読むとマシン語プログラムが書けるようになるか，という大問題。無難に逃げようとすると天気予報になりそうだから，はっきりいってしまう。

なりません。

もし，この本を読んでいる途中で，また，読み終った後で，読者がマシン語プログラムをバリバリ書けるようになった，なんてことが起きたとしても，それは偶然。この本のせいじゃない。あなたがそれだけのことをやったということだ。ざまーみろっ(気に入ってしまった)と叫んでもらっていい(できれば，ここまで聞こえるように)。

そんなところで。

最後になりましたが，連載を支えてくださった読者の皆様，ならびに，編集スタッフの方々に感謝の意を表します。

1990年11月

村田　敏幸

X68000マシン語プログラミング　入門編

1990年12月14日　初版第１刷発行
1994年６月15日　初版第10刷発行

著　者……村田敏幸
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)8140
印刷所……株式会社厚徳社

ISBN4-89052-166-6 C0055