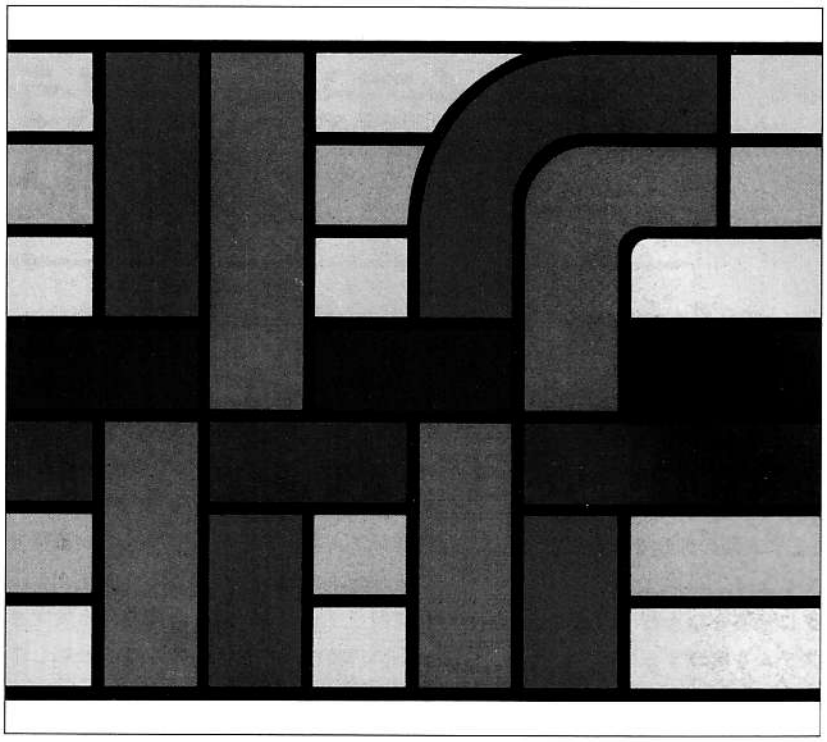
特集

マイコン・システム構築技術セミナー



150号記念号は，原点を確認する意味で，マイコン・システムを構築するための基本的な技術とノウハウをまとめてみる．一口にマイコン・システムといっても幅広いが，ここではわかりやすくするため，① パソコン＋拡張ボード(Basic，C，アセンブラ)システム，② 市販CPUボードによるシステム，③ オリジナルCPUボードによるシステム，④ 発展形態としてのASICシステム，の四形態に分類した．この四つの形態ごとに，バランスのとれたシステムをつくりあげていくにはどうすればよいかを研究していく．その際のキーワードは，「ソフトとハードのトレードオフ」と「階層化」と「時間スケール」と「信頼性」．四つの形態ごとに，「ソフトとハードをどこで分けるか」，「信頼性設計はどこに注意するか」などを検討するわけである．入門セミナーとして，あるいは現在もっている技術を整理するセミナーとして，ぜひご聴講ください．　(編集部)

《特集執筆　長嶋 洋一》

1. マイコン・システム構築技術とは

今世紀の半導体テクノロジーが生み出したコンピュータ技術は，人間の歴史の中で他に例のないスピードで成長してきました．とくにマイクロコンピュータは，〈同時処理〉〈正確・高速な処理〉〈低コスト〉〈連続的動作〉といった領域で，ますます人間の作業を肩代りし，普及・拡大していくことでしょう．

ところで，コンピュータに仕事を与えたり，仕事に応じたコンピュータ・システムを実現するのは，私たち人間の仕事です．したがって，現代のエンジニアは，進歩に応じたより高度なシステム技術を要求されてきています．とくに，「回路技術の延長としてマイコンのハードに至る」「プログラム技術の延長としてマイコンのソフトを考える」という従来の発想でなく，最初からバランス良く，システムをマイコン技術全体としてとらえるセンスが重要になっています．そこで，具体的な実例をまじえながら，段階を追ってマイコン・システムの構築技術をまとめてみることにしました．

（筆者）

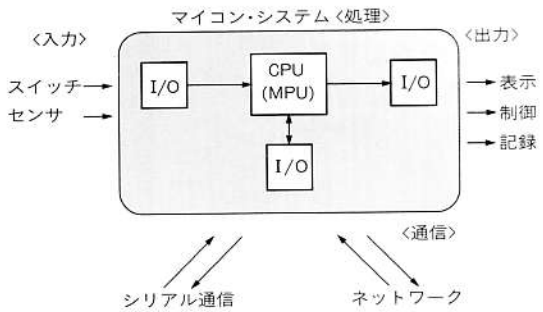
＊

まず，対象とするマイコン・システムの範囲を定義しておくことにします．つぎに，本セミナーで解説するマイコン・システムの四つの段階を概説します．そして，システムを構築する上でのいくつかの重要な視点の中から，ここではとくに，

① ハードウェアとソフトウェア
② 階層化
③ 時間スケール

の三つの点について，「キーワード」としてまとめて，さらに関連技術のいくつかを指摘することにします．

〔図1.1〕マイコン・システムとは



マイコン・システムとは

ここで取り上げるマイコン・システムとは，一般にCPU*とかMPU*と呼ばれる，**マイクロプロセッサ**を使用したシステムです．そして，**図1.1**のように，とくに外界とインターフェースする，独立したものを考えます．つまり，スイッチ・通信・センサなどの手段で外部の情報を**入力**し，システム内部で何らかの**処理**を行い，それを外部への**出力**として，表示・制御・記録・通信などの形式に変換するものです．このような製品を身近な周囲に探してみると，エアコン・テレビ電話・炊飯器・自動車・ファクシミリなどは，すべてこのマイコン技術を駆使していることがわかります．

コンピュータ・システムの歴史では先輩の汎用大型コンピュータや，ミニコン，EWS*なども，広義のマイコン・システムといえますが，ここでは対象から外します．外部との入出力をともなう，目的に応じた簡潔なシステムをどう構築していこうか，という本稿の趣旨においては，これらの大規模なシステムはあまりに高価で，小回りがきかないからです．

＊はp.253～の「アルファベット略語解説」参照

マイコン・システムの形態

ひとくちにマイコン・システムといっても，簡単なボードからパソコンのような複雑な装置まで，いろいろな形態があります．本セミナーでは，**マイコン技術者が，あらたなマイコン応用システムを構築しようというときに，どのような視点で，どんなノウハウとテクニックで，どのようなツールを活用して開発をするのか**について解説します．そこで，雑然とするのを避けるために，マイコン・システムの形態を，

① パソコンを応用したシステム
② 市販のボード・マイコンを利用したシステム
③ 専用(自作)ボードによるシステム
④ ASIC によるシステム

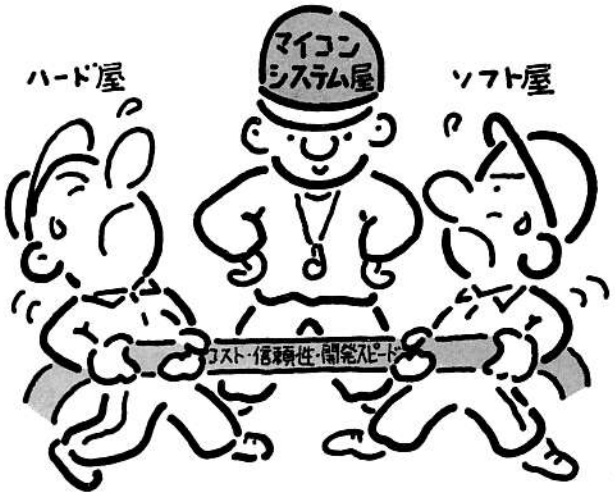
の四つの段階に分類して，それぞれをハードウェアの段階に対応したソフトウェア技術とともに説明してみることにします．そして，各章に共通する視点というのが，以下に述べる「キーワード」となるわけです．

ハードウェアとソフトウェア

さて，第1のキーワードは，ある仕事をマイコン・システムとして実現する際，**どの部分をハード化してどの部分をソフト化するか**，というトレードオフの視点です(**図1.2**)．一般に，同等の機能・仕様を実現するシステムの組み方，というのは何種類も考えられるものです．ところがその中から，〈コスト〉〈開発スピード〉〈信頼性〉といった種々の条件を考慮して，最適のハード/ソフト境界を定める，という作業がもっとも難しく，技術者の腕の見せどころとなるのです．

これは知識や経験によって次第に身についてくる，まさに「ノウハウ」の領域の話なので，ここでひと言でまとめることはできませんが，本セミナーの全体に関係しています．とくに，「ハードウェア技術者 vs ソフトウェア技術者」というセクト主義に反対したい筆者としては，分業の境界線というよりも，相互にオーバラップした大切な部分，と考えたいのです(むしろ「境界帯」ととらえるべきなのかもしれません)．

このハード/ソフト領域の判断は，まずはシステム設計の初期段階において，もっとも重要なファクタとなります．そして，何かトラブルや仕様変更が発生したときには，その対応策もまた，ハード/ソフトの両面から検討する場合が多く，マイコン・システムに最初から最後まで関係しているポイント，といえるでしょう．

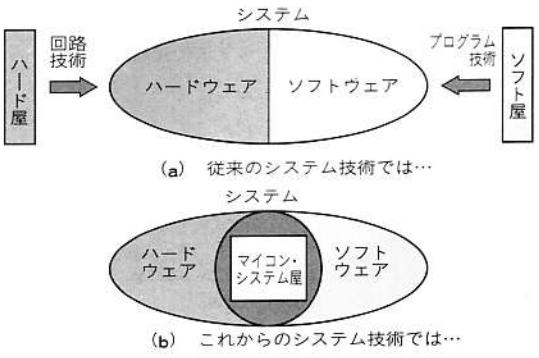


階層化

第2のキーワードは，上のハード/ソフトのそれぞれの面について存在する，**階層化**という視点です．たとえばハードウェアの場合，従来は頭で考えられる程度の規模の電子回路であったのに対して，半導体の集積度の向上はシステムの規模を著しく巨大化させ，**もはや階層的な発想なしにはシステムを理解できなくなりました**．ソフトも同様で，従来の細かいプログラミング・テクニックよりも，システム的に全体を理解・把握するソフトウェア感覚が重要になっています．

この状況を乗り越えるカギとなるのが，あらゆる面での階層化思想です．これは，ある階層について考えるときに，その下の階層は細かな内容を検討しなくて

〔図1.2〕ハードとソフトのトレードオフ



も自由に使える，という，いわばブラックボックスの発想です．日進月歩する技術が，つぎのステップではいちいち細部まで理解・再検討せずに利用できる，という技術の蓄積の秘密がここにあるのです(あとに実例を示します)．

時間スケール

第3のキーワードは，システムが関係する〈時間〉をとらえる感覚です．半導体技術の高速化の面での進歩もまた，従来のハード/ソフトの境界を変化させました．ごく身近な技術においても，かつては不可能とされたアイデアが現実のものとなったり，苦労して駆使してきた「ワザ」が不要になってきています．このため，システムに求める機能・仕様の時間スケールと，実現可能なシステム能力の時間スケールの両方を，最新の技術レベルで的確に把握しておく必要があります．

システム設計の際に，時間スケールの点で適切な判断を誤ると，システムの処理能力・開発効率・信頼性などを大きく変えてしまいます．具体的には，人間の感覚領域である「秒」の時間オーダから，ミリ秒，マイクロ秒，ナノ秒，場合によってはピコ秒までのレンジを適切にシステムに反映させます．また一方で，連続動作・信頼性の面では，「分，時，日，月，年」といったスケールをカバーしていることも必要です．

時間スケールの視点は，ハード/ソフト境界ばかりでなく，設計時にハードの構成を考えるときにも重要です．一般に，ソフトウェアで間に合わない領域をハード・ロジックで実現しますが，マルチCPUシステム化してソフトウェアで吸収できたり，高速CPUのマルチタスク化，というソフト技術で解決できる領域もあります．

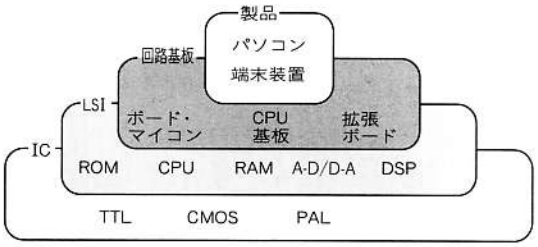
ハードウェアの階層

さて，おさらいの意味で，ここでディジタルのハードウェアの階層を考えてみましょう(図1.3)．まず，最下層にはIC*があります(この下のトランジスタ，ダイオードの階層は，マイコン・システムの技術者にとって，もはや省略しても当面は困らない，と思います)．プロとして最低限のICの階層を理解するための基準としては，ラッチ，デコーダ，アダーなどの標準的な論理ICについて，「必要なデータはマニュアルを読めばいつでも得られる」という状態であれば十分です．

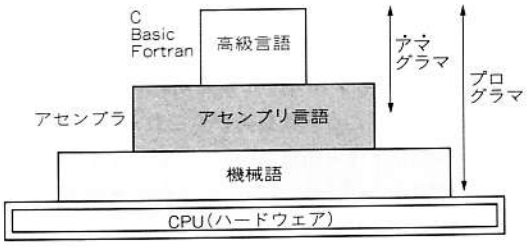
この上の階層にLSIがあります．ここでは，マイコン・システムの中心となるCPUや，CPU周辺のROM*，RAM*，PIO*，SIO*などのLSI*が，機能的にはICの組合せとして理解されます．また，アナログとの境界のADC*，DAC*といった素子も，この階層で扱います．内部回路の詳細を知らなくても，データ・シートにしたがえば手軽に使える，という高機能なLSIが各社から続々と発表されています．

つぎの階層は回路基板のレベルです．汎用の**ボード・マイコン**や，製品に実装される**CPU基板**，実験用の**ブレッド・ボード**，パソコン用の**拡張ボード**などがこの階層です．具体的にはLSIやICによって回路が構成され，マイコン・システムの中核として動作します．ここまではおもにハードの領域でしたが，CPUメモリをRAMで構成して外部からプログラムをロードするようなシステムでは，このへんからソフトと大きく関係してきます．なお，市販のボード・マイコンでは，ボード上のLSIの詳しい知識がなくてもソフトを開発できますが，専用の基板を設計する場合には，個々のLSIに関する詳細な理解と回路技術とが必要になります．

〔図1.3〕ハードウェアの階層



〔図1.4〕プログラム言語の階層



〔リスト1.1〕ROM内の機械語プログラム

Address	Data :bit7	6	5	4	3	2	1	0
102	0	0	0	0	1	0	0	0
103	1	1	0	1	1	0	0	1
104	1	1	0	0	1	1	0	1
105	1	0	0	0	1	0	1	1
106	0	0	0	0	0	0	0	0
107	1	1	0	1	1	0	0	1
108	0	0	0	0	1	0	0	0
109	1	1	1	0	1	1	0	1
110	0	1	0	0	0	1	0	1
111	0	0	1	0	0	0	0	1
112	0	0	0	0	0	0	0	0
113	0	1	0	0	0	0	0	0
114	0	0	1	1	0	1	1	0
115	0	0	0	0	0	0	0	0
116	0	0	1	0	0	0	1	1
117	1	0	1	1	1	1	0	0
118	1	1	0	0	0	0	1	0
119	0	1	1	1	0	0	1	0
120	0	0	0	0	0	0	0	0
121	0	0	1	0	0	0	0	1
122	0	0	0	0	0	0	1	1
123	1	0	1	0	0	0	0	0
124	0	0	1	1	0	1	1	0
125	1	0	0	0	0	0	1	0
126	0	0	1	0	0	0	0	1
127	0	0	0	0	0	0	0	1
128	1	0	0	0	0	0	0	0
129	0	0	1	1	0	1	1	0
130	0	1	0	0	1	1	1	0
131	0	0	1	1	0	1	1	0
132	0	0	0	1	0	1	0	1
133	1	1	1	1	1	0	1	1
134	1	1	0	0	1	0	0	1

〔リスト1.2〕16進表示したプログラム

Address	Data
0066	08
0067	D9
0068	CD
0069	8B
006A	00
006B	D9
006C	08
006D	ED
006E	45
006F	21
0070	00
0071	40
0072	36
0073	00
0074	23
0075	BC
0076	C2
0077	72
0078	00
0079	21
007A	03
007B	A0
007C	36
007D	82
007E	21
007F	01
0080	80
0081	36
0082	4E
0083	36
0084	15
0085	FB
0086	C9

〔リスト1.3〕アセンブリ言語によるプログラム

Address	Machine	ニモニック	
0066	08	EX	AF,AF'
0067	D9	EXX	
0068	CD8B00	CALL	008B
006B	D9	EXX	
006C	08	EX	AF,AF'
006D	ED45	RETN	
006F	210040	LD	HL,4000
0072	3600	LD	(HL),0
0074	23	INC	HL
0075	BC	CP	H
0076	C27200	JP	NZ,0072
0079	2103A0	LD	HL,A003
007C	3682	LD	(HL),82
007E	210180	LD	HL,8001
0081	364E	LD	(HL),4E
0083	3615	LD	(HL),15
0085	FB	EI	
0086	C9	RET	

〔リスト1.4〕マクロ・アセンブラによるプログラム

```
macro   &exchange_push
        ex      af,af'
        exx
        endmac
                        ←マクロ定義

macro   &exchange_pop
        exx
        ex      af,af'
        endmac
                        ←マクロ定義

        org     nmi_area
nmi:
        &exchange_push  ←マクロ・コール
        call    uart_nmi
        &exchange_pop   ←マクロ・コール
        retn

initialize:
        ld      hl,work
ram_clear_loop:
        ld      (hl),0
        inc     hl
        cp      h
        jp      nz,ram_clear_loop
        ld      hl,pia+3
        ld      (hl),10000010b
        ld      hl,uart+1
        ld      (hl),01001110b
        ld      (hl),00010101b
        ei
        ret
```

その上の階層としては，単体の製品というレベルになります．これは，この境界から内側は，使用するユーザにとって動作が保証されているものです．ときにはマイコン・システムの構成要素(部品)として使われる〈単体のパソコン〉や〈ポータブル端末装置〉なども，この階層にあたります．

ソフトウェアの階層

つぎに，CPUのプログラム言語という視点から，ソフトウェアの階層について調べてみます(図1.4)．最下層は**リスト1.1**のような，CPUメモリに格納されたプログラムです．これはディジタルの2進数で，たんなる1か0の羅列という味気ないものです．このデータを16進数表示して，アドレス順に並べたものが**機械語プログラム**・リストとなりますが，普通はこれを読んでCPUの動作を追うのは非常に困難です(**リスト1.2**)．

この一つ上の階層には，アセンブリ言語のレベルがあります．デバッガというツールを使うと，機械語プログラムをニモニック記号による**アセンブラ・プログラム**表示に変換できます(**リスト1.3**)．アセンブリ言語は，理解が困難な機械語のソフト開発を，人間に近づけて改善するために作られており，つぎの階層の高級言語とは別の使い勝手があるために，現在でも重要なプログラム開発環境です．この階層の言語は，アセンブラというツールによってCPUの機械語に変換されますが，アセンブラを強化する**マクロ機能**なども相当に充実しています(**リスト1.4**)．

さらに上の階層には，C*やFortran*といったコン

パイラ言語があります(**リスト1.5**).これらの**高級言語**は，プログラムをより人間の言語に近づけたもので，コンパイラというツールによって機械語に変換されます．人間が理解しやすいのならすべてのCPUソフトを高級言語で書けばいいのでは，と思うかもしれませんが，高級言語には逆にCPUのハードに密着した細かな処理の記述が困難になるというデメリットがあり，現実にはシステムの要求によって，高級言語を使わない場合も非常に多いのです．

また，コンパイラで変換するタイプ以外に，Basic*のような高級言語のグループとして，ソース・プログラムをそのまま実行時に解釈して動作する**インタプリタ**言語もあります(**リスト1.6**).この種類の言語にも特有の長所と短所があります．また，<Basicのコンパイラ>や，RUN/Cのような<インタプリタ的に使えるC>などもあるので，なれてきたら必要に応じてうまく使い分けたいものです．

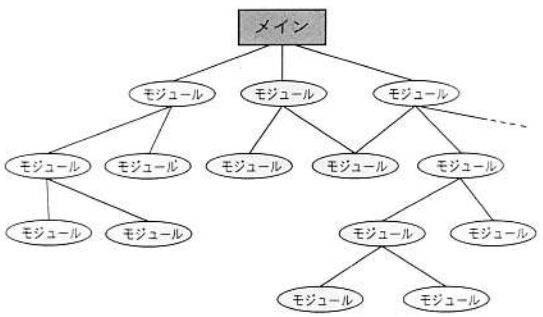
〔リスト1.5〕高級言語の一つCによるプログラムの例

```
#include  <stdio.h>
#define   out_board      0xe0d0
#define   in_board       0xe0d2

main(argc,argv)
        int argc;  char *argv[];
{
        char c,d;
        puts("Start !!");
        while(1){
                c=csts();
                if(c==0x1b) exit();
                else if(c=='p'){
                        outportb(out_board,0x3f);
                        d=inportb(in_board);
                        outportb(out_board,0x00);
                        display(d);
                }
                else if(c=='o'){
                        outportb(out_board,0xc0);
                        d=inportb(in_board);
                        outportb(out_board,0x00);
                        display(d);
                }
        }
}

display(d)
        char d;
{
        if((d&0x01)==0) puts(" - ");
        else puts(" + ");
}
```

ソフトウェア関連技術

以上の個々のプログラム言語と関連したプログラミングの技術として重要なものに，プログラム自身の階層構造化があります(図1.5)．これは，プログラムの**モジュール化**とか**ライブラリ利用**によるソフト開発，と

〔図1.5〕階層化プログラム



〔リスト1.6〕
インタプリタ言語
Basicによる
プログラムの例

```
1000 'save "sample".a
1010      DEFINT A-N:DEFDBL P-Z:KEY OFF:SCREEN 1
1020      INPUT "Input Center (X) = ",P1:INPUT "Input Center (Y) = ",P2
1030      INPUT "Expand Scale = ",P3:INPUT "Plot Step = ",J
1040      CLS:LINE (0,29)-(1023,767),1,B:LOCATE 1,8,0
1050      PRINT "Center = (";P1;",";P2;")     Scale =";P3;"   Step =";J;
1060      X0=P1-P3:Y0=P2+P3:X1=P3/512#
1070    FOR IX=1 TO 1023 STEP J
1080      X2=X0+CDBL(IX)*X1
1090      FOR IY=143 TO 881 STEP J
1100          Y2=Y0-CDBL(IY)*X1
1110          GOSUB 1150:IF F=1 THEN PSET (IX,IY-114)
1120      NEXT IY:LOCATE 1,73,0:PRINT IX;
1130    NEXT IX:LOCATE 1,73,0:PRINT "      ";
1140    A$=INKEY$:IF A$<>" " THEN 1140 ELSE SCREEN 0:KEY ON:LOCATE 1,1,1:END
1150 '### Subroutine ###
1160      F=0:Z1=0#:Z2=0#
1170      FOR K=1 TO 100
1180          ZZ=Z1*Z1-Z2*Z2+X2:Z2=2#*Z1*Z2+Y2:Z1=ZZ
1190          IF Z1*Z1+Z2*Z2>4# THEN RETURN
1200      NEXT K:F=1:RETURN
```

いった手法のことで，プログラムを階層的に，さらに個々のモジュールに，と分割して，それぞれの単位でプログラマの目がよく届くように構成するものです．この構造化の手法によって，ソフトの信頼性が向上したり，チームプレイによる巨大プロジェクトの遂行に役立つ，といった例は多く見られます．

また，CPUの処理能力の向上により，パソコンなどのソフトウェアには，さらに高機能なシステムも出現しました．1本の完結したプログラムを一つの**タスク**と呼び，複数のタスクを一見同時に処理してしまう，という**マルチタスク**がその一つで，マイコンの操作者をユーザとして，複数のユーザの処理を一見同時に実行してしまう，という**マルチユーザ**がもう一つです(**図1.6**)．これは現実には，

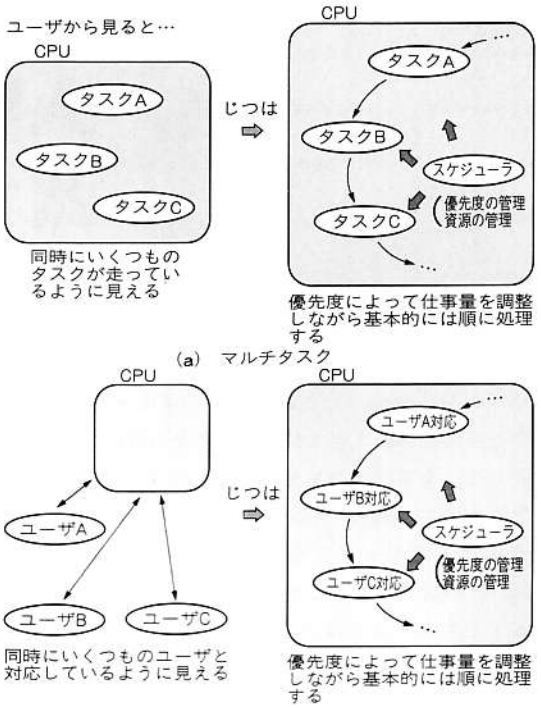
① 個々の処理に関する優先度の調停
② 共有するハードウェア資源(周辺機器など)の調整
③ 個々の処理に関するデータの相互通信サービス
④ ランダムな外部からの要求への対応

などを実行しつつも，基本的にはCPUは順番に処理しているだけなのですが，ソフトウェアの階層構造とCPUの高速化とによって，人間の時間感覚からは同時処理といえるような，美しいシステムがつぎつぎに実現されてきているようです．

CPUというハードウェアの世界では，最近のCISC(シスク)とRISC(リスク)の話題が注目されています．これはもちろん，外見上はハード関連の技術なのですが，本質は，むしろソフトウェアに重点をおく新技術といえるものです．あまりにも集積度が上がりすぎてCISCの問題点が表面化する一方で，ソフトウェア技術の大いなる進歩が，RISCという単純な思想のハードウェアを支えている，というのが実際のところでしょう．これまでCISCでハードウェア上に押し込んできた複雑な処理を，外部のソフトに出してしまおう，という発想を現実のものとするためには，外側からサポートするソフトへの要求がかなりの規模になるので，システムとしてのRISCはまだ完成されてはいないでしょう．今後も両方のタイプのCPUが共存するのでしょうが，これまでのCPUでは従属的であったソフトウェアが，ここにきてむしろCPU自体のブレークスルーを助けている，という図式がおもしろいと思います．

また，そのほかのCPUに関するソフトウェア技術としては，プログラムとは別に，扱うデータに着目した視点もあります．たとえば，ワープロ，データベー

〔図1.6〕マルチタスクとマルチユーザ



スなどの目的に応じたデータの表現形式・処理形式を意識的に指向して検討したり，あるいは通信処理ソフトならばデータ圧縮やエラー修正能力に重点を置く，といった考え方です．また，これとは別のもう一つの発想として，あらゆるデータをすべて簡単なテキスト形式で表現することで，どんなソフトウェアであっても環境が統一され，データ処理の手順が共通化される，という規定をもつシステムもあります．

さらに，オブジェクト指向のプログラミング環境，AI関連言語，ファジィなシステム，あるいはハードウェアと結びついたデータフロー・プロセッサ，といった技術もつぎつぎに現実になってきています．これらは，ノイマン型逐次処理による機械的動作，という従来のCPUシステム(の延長線上)よりも高度な，高速同時処理や人間の思考プロセスに近づいた動作などが必要とされる場合には，強力な候補となるものです．これらの新しい技術というのは，一般に普及するまでは開発リスクも大きいだけに，なかなか当面の対象としては採用しにくいのですが，つねに進歩の状況にアンテナを立てて，機会があればトライしていきたいものです．

2．入門レベル：パソコン＋拡張ボード

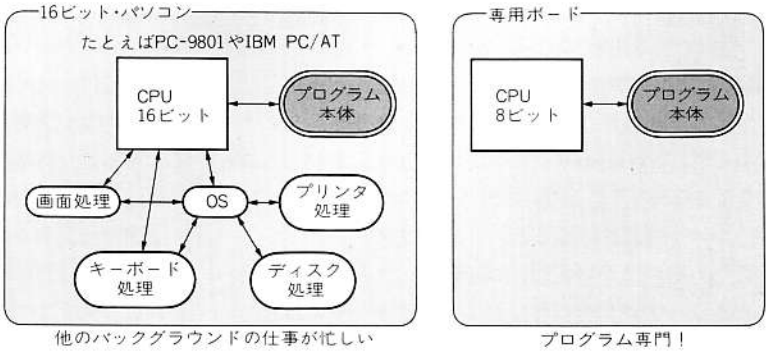
パソコンのBasicと拡張ボードによるシステムでもかなりの仕事ができるので，システム設計の第1段階として，まずパソコン＋市販の拡張ボード＋Basicをとりあげる．Basicプログラミングは，ゲーム・プログラムと入力ルーチンを題材にポイントを述べる．Basicの遅さ解決のため，機械語ルーチンによる高速化とBasicによる高速化にふれ，つぎにBasicコンパイラ/DOS環境/バッチ処理へと移り，さらに構造化の限界からC言語による開発へと進んでいく．最後にCでの開発の問題点として割込みに焦点をあて，C言語とアセンブラをどんなバランスで使いこなすかといったテーマを研究する．（編集部）

マイコン・システムに初めて接する人にとって，一番わかりやすいのはパソコンを使用したシステムでしょう．パソコン内部は完全に動作が保証されていて，ソフトウェアにしぼって開発できるからです．とはいっても，このシステムでも相当の応用範囲と実用性をもっています．ここでは第1段階として，パソコン応用のシステムについて，プログラム言語の段階に分けてその可能性を検討してみます．言語としては，まずBasicから始まり，C，アセンブラへと進むことになります．

パソコン使用システム

まず，ここでのサンプルとしては16ビット・パソコンを選ぶことにします．その理由を説明しましょう．パソコンの心臓部のCPUの仕事には，画面，キーボード，タイマ，ディスク，OS*などの，直接，ユーザに見えないバックグラウンドの仕事がかなりあります．ユーザの印象としては，自分が組んだプログラム本体の処理を担当している時間は，全体のごく一部という感じになります．つまり，専用の8ビットCPUボードが専用の周辺ハードだけを常時使用するような処理量のシステムを，MSXのような8ビット・パソコンで実行させようとすると，汎用パソコンであるために多種にわたる余分なバックグラウンド処理を行う必要があり，かなり荷が重くなります．そこで，PC-9801やIBM PC/ATのような16ビット・パソコンを使うことにな

〔図2.1〕
パソコンvs専用ボード



るのです(図2.1).

さて，前章で述べたように，今回のマイコン・システムでは外界とのインターフェースを目的にしますから，パソコン単体ではこの目的をはたせません．そこで，外部と信号をやりとりするための，拡張スロットに挿入した拡張ボードによって実現します．まず，例題として，

例1

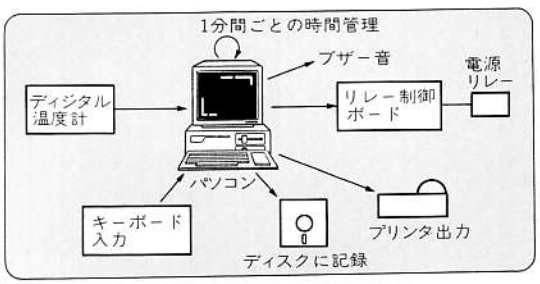
外部のディジタル温度計からの信号を受けて，刻々とそのデータを表示しながらディスクに記録し，さらに1分間ごとの平均温度をプリンタから打ち出す．温度が設定値を超えたら，ブザーが鳴ってヒータの電源リレーを切る．

というようなシステムを考えてみます．

まずはBasicから

パソコンを使うとなるとなんらかの言語が必要となりますが，ここでは，たいていのパソコンに標準装備されているBasicを使うことにします．今どきBasicなんて，と馬鹿にすることなかれ，まだまだこの言語は実験レベルでは有効なのです．筆者の場合，新しいシステムの実験やハードウェアのデバッグの際に，まずはパソコンのスロットに差し込んだ実験用基板とBasicによって，疑似的なシステム信号を与えたり，回路の出力信号の状態モニタとして，ときどき使用して

〔例1〕パソコン応用の例――温度計測システム



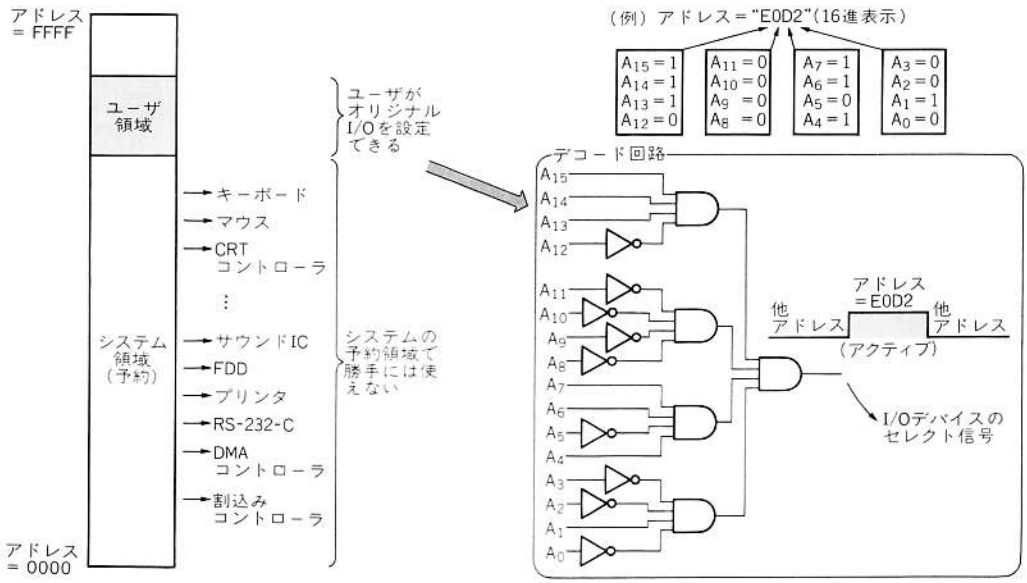
コラム2.1 パソコン環境の安全性

パソコンが突然に致命的なトラブルを起こして，泣きたくなったような経験は，おそらく筆者ばかりでなく，パソコンを使っている人なら誰でも一度はあるのではないでしょうか．

考えてみると，パソコンというのは，落雷による停電のような天災から，他の設備の電源オン/オフによる瞬間的電圧低下，瞬間停電，さらにタバコの煙によるハード・ディスクやFDD*の故障，といった人災まで，いろいろな危険性にさらされているのです．また，一度パソコンのケースを開けてみるとわかりますが，送風ファンによって相当の埃，ゴミが内部にたまっているような，周囲の劣悪な環境もよく見かけます．また，とくにプログラム開発用のパソコンでは，1日に電源スイッチを何百回も入れる，といったハードな使い方を長期間行うこともあるので，機種によっては本体のマザーボードが簡単に異常を起こします．

こうしてみると，ツールとしての**パソコンは非常に弱いものであって，ときどき故障するのが当然なのだ**，データは致命的に破壊されるものなのだ，という意識で接する必要性が浮かび上がってきます．筆者の場合，具体的には，無停電バッテリ電源装置や，故障したときに代わりのパソコンを確保できる環境，そしてデータの定期的なバックアップ(ハード・ディスクをフロッピ20枚に収めるのに1時間以上かかるので，これは1ヵ月に一度とか，毎日の作業ファイルをフロッピに収めるバッチ・プログラムとかの，よくある方法)という対策です．しかし，ある意味で最大の方策というのは，パソコンをあまり信じないようにする，という上記の姿勢であると思います．

〔図2.2〕
I/O アドレスの例



います．なんといってもBasicはコンパイルなどの余計な手間もなく，結果を見て即座に手直しできるのが助かります．

ここでは簡単のために，ディジタル温度計からの出力は8ビット・パラレル・データで，これが入力ポート用の拡張ボードに供給されるとしましょう．また，ブザーはパソコン内蔵のブザーとし，電源リレーの制御には出力ポート用の拡張ボードを用いることにします．これらの拡張ボードを使用する場合には，まず最初に，このボードの**I/O*アドレス**を決めます(**図2.2**)．一般に，パソコンのアドレス空間の中には，ユーザに解放されて自由に使えるアドレスがあるので，拡張基板・拡張ポートがここに割り当てられるようにアドレスをデコードしてやります．市販のボードであればジャンパやDIPスイッチによって設定できますが，自作基板の場合には丁寧にフル・デコードしてやらないと，思わぬシステム予約アドレスと衝突したりします(**コラム2.2**参照)．

以上の準備ができれば，あとはプログラムの問題ということになります(**リスト2.1**)．ここでは，普通のBasicコマンドに加えて，拡張ボードに対する**入出力**

〔リスト2.1〕パソコンによる温度計測システムのプログラム例

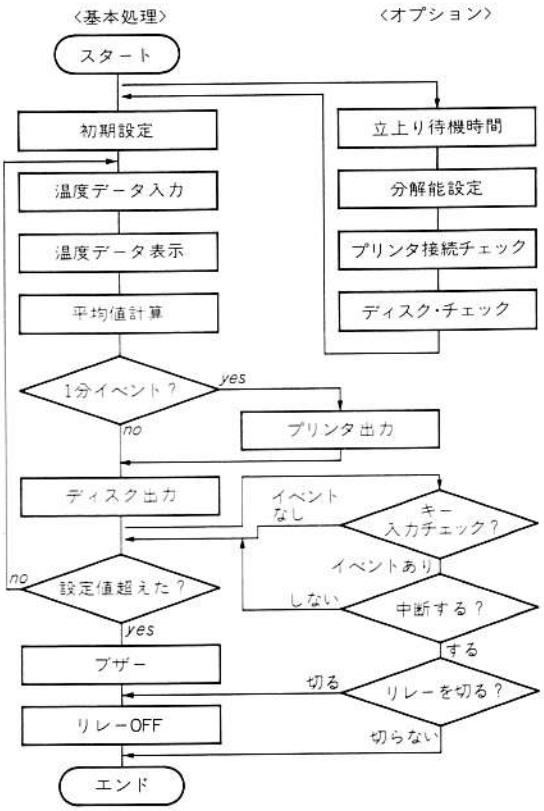
```
1000 'save "sample",a
1010    TEMP=&HE0D0:POWER=&HE0D2   'Port Address
1020    OUT POWER,&HFF
1030    OPEN "c:test.dat" FOR OUTPUT AS #1
1040    LIMIT=50
1050    SUM=0:I=0
1060    OLD$=LEFT$(TIME$,5)
1070 '##### Main Loop #####
1080    X1=INP(TEMP)
1090    X2=INP(TEMP)
1100    IF(X1<>X2) THEN GOTO 1080 ELSE PRINT X1;
1110    SUM=SUM+X1:I=I+1
1120    MEAN=INT(10*SUM/I)/10
1130    T$=LEFT$(TIME$,5)
1140    IF(T$<>OLD$) THEN GOSUB 1200  'Print Out
1150    WRITE #1,X1
1160    IF X1=<LIMIT THEN 1070
1170    OUT POWER,0
1180    BEEP:BEEP:BEEP:BEEP:BEEP
1190    CLOSE #1:END
1200 '##### Printer Output Routine #####
1210    LPRINT "Time = ";T$;"   ,   Temp =";MEAN
1220    OLD$=T$
1230    RETURN
```

コマンドが使われる以外には，とくに変わった部分もありません．ディジタル温度計からの出力は，入力ポートを読んでいる間に変化する可能性があるので，続けて2回読み込んで比較するといいでしょう．ディスプレイ表示，ディスク書込みなどの処理は普通のコマンドで書けます．また，ミリ秒単位の誤差が問題となるわけではないケースなので，1分ごとの処理も，割込みとかを考えずに，時刻の関数を呼んで比較すればよいでしょう．ブザーのコマンド，プリント・アウトのコマンドも普通のものです．**拡張スロットのボード上で外部機器の電源を切る，というのは危険なので，出力ポートはできればフォト・カプラでアイソレートして小型のリレーを制御し，さらに外部に置いた大型の電源リレーをオン/オフする，というのが定石**です．

また，メイン処理に付加するものとしては，途中で処理を中断するためのキー入力ルーチンを設けて，ここでは電源リレーを切るか切らないかを聞いてくるとか，初期の立上り時に対象の装置の温度がある程度安定するまで，実際のプログラム実行の開始を待つための待機時間を設定できるとか，データを「変化」と見なす分解能や，計測のサンプリング時間間隔について，メニュー形式で設定する機能などが考えられます(**図2.3**)．

パソコンのBasicと拡張ボードによるシステムでも，以上のようにかなりの仕事を実現できます．ややお役所的かもしれませんが，単純なシステムを非常に短期間に構築する必要があり，コスト的な制約があまりなく，むしろ信頼性・再現性を重視するのであれば，市販のパソコン＋市販の拡張ボード＋Basic，という組合せはかなり強力な候補なのです．

〔図2.3〕処理プログラムのフローチャート例



Basicプログラミングのポイント

ここで，いくつかのサンプルによって，筆者が気に入っているBasicプログラム上のテクニック(という

コラム 2.2

I/Oアドレスの失敗例

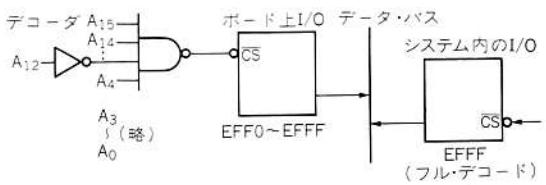
デコードのときに**図A**のように下位アドレスを省略しても，あるときはうまくいきます．じつは，たまたまこのI/Oは入力ポートであり，そのときのソフトではシステム内のI/Oが呼ばれずにすんでいたのです．ところが別のソフトではシステム用の内部I/Oとバス・ファイトしてしまっていて，はじめて不完全デコードの問題に気づいたわけです．

図Aの例では，

・たまたまよかったソフト…EFF0をアクセスしてOKだった

・つぎのソフト…システムがEFFFコールを使うものだったので，ボード上のI/Oもバスに出力をのせようとしてシステムが正常に動かなかった

〔図A〕不完全デコードの例



〔リスト2.2〕入力ルーチンのいろいろ

［入力ルーチンの例（A）］

```
1000 'save "input-1.bas",a
1010  CLS
1020  PRINT "##### Select Menu #####":PRINT:PRINT
1030  PRINT "       (1) Program <1>":PRINT
1040  PRINT "       (2) Program <2>":PRINT
1050  PRINT "       (3) Program <3>":PRINT
1060  PRINT "       (4) Job Finish !":PRINT
1070  PRINT:PRINT "                    Select --- ";
1080  INPUT X
1090  IF X=1 THEN GOTO 2000' Jump to Program <1>
1100  IF X=2 THEN GOTO 3000' Jump to Program <2>
1110  IF X=3 THEN GOTO 4000' Jump to Program <3>
1120  IF X=4 THEN END
1130  GOTO 1010
```

［入力ルーチンの例（B）］

```
1000 'save "input-2.bas",a
1010  CLS
1020  PRINT "##### Select Menu #####":PRINT:PRINT
1030  PRINT "       (1) Program <1>":PRINT
1040  PRINT "       (2) Program <2>":PRINT
1050  PRINT "       (3) Program <3>":PRINT
1060  PRINT "       (4) Job Finish !":PRINT
1070  PRINT:PRINT "                    Select --- ";
1080  X$=INKEY$:IF X$="" THEN 1080
1090  IF X$="1" THEN GOTO 2000' Jump to Program <1>
1100  IF X$="2" THEN GOTO 3000' Jump to Program <2>
1110  IF X$="3" THEN GOTO 4000' Jump to Program <3>
1120  IF X$="4" THEN END
1130  GOTO 1010
```

ほどのものでもありませんが)を紹介しましょう．それぞれ，一般のBasicプログラムにおいて活用できるポイントとなってくれるとうれしいのですが．

●入力ルーチンのいろいろ

リスト2.2にあるのは，Basicで入力を待つための代表的な方法です．(A)はもっともシンプルなもので，メニューを並べたあとで，その番号の入力を待ちます．ここでは，数字の入力のあとでリターン・キーを押す必要があり，合計2回のキーインを必要とします．こ

［入力ルーチンの例（C）］

```
1000 'save "password.bas",a
1010  KEY OFF
1020  CLS
1030  LOCATE 10,10
1040  BEEP
1050  B$=""
1060  PRINT "Input Password --- ";
1070  A$=INKEY$:IF A$="" THEN 1070
1080  PRINT " ";
1090  B$=B$+A$
1100  IF LEN(B$)<>9 THEN 1070
1110  IF B$<>"interface" THEN 1040
```

〔リスト2.3〕ゲーム(1)「惑星X上の投石ゲーム」①

```
10 'save "a:game1.bas",a
20   DEFINT I-N:DIM MT(640)                                  ' ﾍﾝｽｳ ﾉ ﾃｲｷﾞ
30 *START                                 ' ｼｮｷ ｾｯﾃｲ
40   CLS 2:GOSUB *INITIAL.PARA:GOSUB *INITIAL.SCREEN
50 *MAIN                                  ' ﾒｲﾝ ﾙｰﾌﾟ
60   GOSUB *CLOCK:A$=INKEY$:IF A$="" THEN *MAIN             ' ｷｰ ﾆｭｳﾘｮｸ
70   IF A$=CHR$(&HD) THEN ON PHASE GOTO *CR1,*CR2,*CR3,*CR4 ' ｴﾝﾀｰ ｷｰ ?
80   IF A$=CHR$(253) THEN GOSUB *INITIAL.SCREEN:GOTO *MAIN  ' ﾍﾝｶﾝ ｷｰ ?
90    '  IF A$=CHR$(&H1B) THEN CLS 2:END                    ' (ﾃﾞﾊﾞｯｸﾞ)
100  B$=B$+A$:PRINT A$;:CX=CX+1:GOTO *MAIN
110 *THROW2                               ' ﾋﾀﾞﾘ ﾉ ﾋﾄ ｶﾗ ﾐｷﾞ ﾆ ﾅｹﾞﾙ ﾙｰﾁﾝ
120   X0=JJ2-4:Y0=MT(JJ2)-8:ST=0
130    ST=ST+1:GOSUB *CALC2:PSET (X1,Y1)                    ' ﾌﾟﾛｯﾄ
140    IF X1<0 OR Y1>400 OR X1>640 THEN 180                 ' ｶﾞﾒﾝﾖ ﾃﾞﾀ?
150    IF Y1>MT(X1) THEN 180                                ' ﾔﾏﾆ ｱﾀｯﾀ ?
160    IF (X1-JJ1)^2+(Y1-MT(JJ1)+12)^2=<4 THEN PX=JJ1:GOTO *HIT ' ﾊﾝﾃｲ
170    GOTO 130                                             ' ﾙｰﾌﾟ
180   GOSUB *MESS:GOTO *MAIN                                ' ﾊｽﾞﾚﾃﾞｼﾀ..
190 *CALC2                                ' ﾎｳﾌﾞﾂｾﾝ ﾉ ｼｮﾘ
200   TH=ANG#3.1415926535#/180                              ' ｶｸﾄﾞ ﾍﾝｶﾝ
210   X1=X0+ST#(WIN-SPD#COS(TH))/200                        ' X-ﾎｳｺｳ
220   Y1=Y0-ST#SPD#SIN(TH)/200+ST#ST#GGG/2000               ' Y-ﾎｳｺｳ
230   RETURN
240 *THROW1                               ' ﾐｷﾞ ﾉ ﾋﾄ ｶﾗ ﾋﾀﾞﾘ ﾆ ﾅｹﾞﾙ ﾙｰﾁﾝ
250   X0=JJ1+4:Y0=MT(JJ1)-8:ST=0
260    ST=ST+1:GOSUB *CALC1:PSET (X1,Y1)                    ' ﾌﾟﾛｯﾄ
270    IF X1<0 OR X1>640 OR Y1>400 THEN 310                 ' ｶﾞﾒﾝﾖ ﾃﾞﾀ?
280    IF Y1>MT(X1) THEN 310                                ' ﾔﾏﾆ ｱﾀｯﾀ ?
290    IF (X1-JJ2)^2+(Y1-MT(JJ2)+12)^2=<4 THEN PX=JJ2:GOTO *HIT ' ﾊﾝﾃｲ
300    GOTO 260                                             ' ﾙｰﾌﾟ
310   GOSUB *MESS:GOTO *MAIN                                ' ﾊｽﾞﾚﾃﾞｼﾀ..
320 *CALC1                                ' ﾎｳﾌﾞﾂｾﾝ ﾉ ｼｮﾘ
330   TH=ANG#3.1415926535#/180                              ' ｶｸﾄﾞ ﾍﾝｶﾝ
340   X1=X0+ST#(WIN+SPD#COS(TH))/200                        ' X-ﾎｳｺｳ
350   Y1=Y0-ST#SPD#SIN(TH)/200+ST#ST#GGG/2000               ' Y-ﾎｳｺｳ
360   RETURN
370 *CR2                                  ' ﾆｭｳﾘｮｸ ﾃﾞｰﾀ ｼｮﾘ (2)
380   SPD=VAL(B$):IF SPD>0 AND SPD<1000 THEN PHASE=3:GOTO *THROW1
390    BEEP:GOSUB *MMESS:GOTO *MAIN
```

〔リスト2.3〕ゲーム(1)「惑星X上の投石ゲーム」②

```
400 *CR4                                  ' ニュウリョク データ ショリ (4)
410   SPD=VAL(B$):IF SPD>0 AND SPD<1000 THEN PHASE=1:GOTO *THROW2
420    BEEP:GOSUB *MMESS:GOTO *MAIN
430 *CR1                                  ' ニュウリョク データ ショリ (1)
440   ANG=VAL(B$):IF ANG>0 AND ANG<100 THEN PHASE=2:GOSUB *MESS:GOTO *MAIN
450    BEEP:GOSUB *MMESS:GOTO *MAIN
460 *CR3                                  ' ニュウリョク データ ショリ (3)
470   ANG=VAL(B$):IF ANG>0 AND ANG<100 THEN PHASE=4:GOSUB *MESS:GOTO *MAIN
480    BEEP:GOSUB *MMESS:GOTO *MAIN
490 *MESS1                                ' メッセージ ディスプレイ <カクト">
500   LOCATE 15,23:PRINT SPACE$(60);
510   LOCATE 15,23:PRINT "Player(LEFT): Angle(1-99)=";
520   CY=23:CX=41:RETURN
530 *MESS2                                ' メッセージ ディスプレイ <スピート">
540   LOCATE 45,23:PRINT "Speed(1-999)=";
550   CY=23:CX=58:RETURN
560 *MESS3                                ' メッセージ ディスプレイ <カクト">
570   LOCATE 15,23:PRINT SPACE$(60);
580   LOCATE 15,23:PRINT "Player(RIGHT): Angle(1-99)=";
590   CY=23:CX=42:RETURN
600 *MESS4                                ' メッセージ ディスプレイ <スピート">
610   LOCATE 46,23:PRINT "Speed(1-999)=";
620   CY=23:CX=59:RETURN
630 *MMESS                                ' メッセージ ノ ショウキョ
640   LOCATE 15,23:PRINT SPACE$(60);
650 *MESS                                 ' フェーズ ニ ヨッテ メッセージ ヲ エラブ
660   B$="":ON PHASE GOSUB *MESS1,*MESS2,*MESS3,*MESS4:RETURN
670 *INITIAL.SCREEN                       ' ショキ ガメン ノ セッティング
680   CONSOLE ,,0:CLS 2                                          ' カーソル OFF
690    LINE (0,0)-(639,399),,B:LINE (1,1)-(638,398),,B           ' ガメン クリア
700    LINE (3,3)-(636,396),,B:LINE (4,4)-(635,395),,B
710    LOCATE 7,1:PRINT "<< STONE THROWING GAME >>";             ' タイトル
720    PRINT "  ---  Produced by Y.Nagashima"
730    COLOR@ (5,1)-(63,1),4                                     ' ハンテン ヒョウジ
740   FOR I=6 TO 633:PSET (I,MT(I)):NEXT I                      ' ヤマ ノ ヒョウジ
750    FOR I=JJ1-5 TO JJ1+5:PSET (I,MT(JJ1)-1):NEXT I           ' アシバ ノ ヒョウジ
760    FOR I=JJ2-5 TO JJ2+5:PSET (I,MT(JJ2)-1):NEXT I           ' アシバ ノ ヒョウジ
770   PX=JJ1:PY=MT(JJ1):GOSUB *PERSON                           ' ヒト ノ ヒョウジ
780    PX=JJ2:PY=MT(JJ2):GOSUB *PERSON                          ' ヒト ノ ヒョウジ
790   GOSUB *MESS                                               ' 1st メッセージ
800   RETURN
810 *INITIAL.PARA                 ' ヤマ ノ チケイ ト カゼ ト シュウリョク ヲ ランダム ニ ツクル
820   LOCATE 30,10:PRINT "wait a moment ..."
830   RANDOMIZE VAL(RIGHT$(TIME$,2))
840    GGG=9.8+9.8*(RND-.5)                                      ' シュウリョク
850    WIN=100*(RND-.5)                                          ' カザムキ,オオキサ
860    JJ1=30+220*RND:MT(JJ1)=360-90*RND                         ' ヤマ ノ チケイ
870     FOR I=JJ1-5 TO JJ1+5:MT(I)=MT(JJ1):NEXT I
880     MT(6)=MT(JJ1)+(393-MT(JJ1))*(RND-.5)*1.5
890      FOR I=7 TO JJ1-6:NR=2*(RND-.5)
900       MT(I)=MT(6)+(I-6)*(MT(JJ1-5)-MT(6))/(JJ1-12)+NR:NEXT I
910    JJ2=610-220*RND:MT(JJ2)=360-90*RND
920     FOR I=JJ2-5 TO JJ2+5:MT(I)=MT(JJ2):NEXT I
930     MT(633)=MT(JJ2)+(393-MT(JJ2))*(RND-.5)*1.5
940      FOR I=JJ2+6 TO 632:NR=2*(RND-.5)
950       MT(I)=MT(JJ2+5)+(I-JJ2-5)*(MT(633)-MT(JJ2+5))/(627-JJ2)+NR:NEXT I
960   IF MT(JJ1)>MT(JJ2) THEN MAX=MT(JJ2) ELSE MAX=MT(JJ1)
970   JP3=260+120*RND:MT(JP3)=MAX-200*RND-30
980    FOR I=JJ1+6 TO JP3-1:NR=3*(RND-.5)
990     MT(I)=MT(JJ1+5)+(I-JJ1-6)*(MT(JP3)-MT(JJ1-5))/(JP3-JJ1-5)+NR:NEXT I
1000    FOR I=JP3+1 TO JJ2-6:NR=3*(RND-.5)
1010     MT(I)=MT(JP3)+(I-JP3)*(MT(JJ2-5)-MT(JP3))/(JJ2-5-JP3)+NR:NEXT I
1020    PHASE=1
1030   RETURN
1040 *HIT                                 ' アタッタ トキノ ディスプレイ ( チョット エクボカッタ... )
1050   PY=MT(PX)
1060   FOR I=14 TO 45
1070    CIRCLE (PX,PY-I),(I-8)/2                                  ' タンコブ?
1080   NEXT I
1090    BEEP:BEEP                                                 ' コレガ ウルサイ !
1100   LOCATE 15,23:PRINT SPACE$(60);
1110   LOCATE 15,23:PRINT "#####  Hit  #####    ( Next Map --> [Esc] )"
1120    GOSUB *CLOCK:IF INKEY$=CHR$(&H1B) THEN *START ELSE 1120
1130 *PERSON                              ' ヒト ノ スガタ ノ ディスプレイ
1140    CIRCLE (PX,PY-12),2                                       ' アタマ
1150    LINE (PX,PY-10)-(PX,PY-4)                                 ' トウタイ
1160    LINE (PX,PY-4)-(PX+3,PY-1)                                ' アシ
1170    LINE (PX,PY-4)-(PX-3,PY-1)                                ' アシ
1180    LINE (PX,PY-7)-(PX+4,PY-8)                                ' テ
1190    LINE (PX,PY-7)-(PX-4,PY-8)                                ' テ
1200    RETURN
1210 *CLOCK                               ' リアルタイム トケイ ヒョウジ ルーチン ( オマケ )
1220    IF CLK$=TIME$ THEN RETURN ELSE CLK$=TIME$                 ' アラタナ データ ?
1230    LOCATE 67,1:PRINT "[";CLK$;"]":LOCATE CX,CY:RETURN        ' ヒョウジシテ モトル
```

の手間の反面，いったん入力した番号が表示された状態で，バックスペース・キーによって「打ち直し」ができる，というメリットもあります．(B)のようにINKEY$関数を使うと，なにかの数字を押した瞬間，すぐさまその処理に飛んでいきます．ただし，キー・タッチは1回ですむのですが，うっかりミス・タッチをしても修正できない，という欠点もあるのです．(C)はこの応用の例で，パスワードをきいてくるプログラムです．このプログラムを後述する方法によってEXEファイルとして，ルート・ディレクトリのAUTOEXEC.BATの冒頭に入れておくと，一見するとUnixへのログインのような感じになります(もっとも，このパスワード・チェックは簡単に突破できてしまうのですけれど)．ここでは，1文字ずつをINKEY$で入力しつつ，カーソルを文字の分だけ移動して表示する，というところが，いかにもそれっぽくなる秘訣です．

●シンプルなゲーム(1)

リスト2.3は，だいぶ昔に作った，非常にシンプルなゲーム・プログラムの全リストです．中央にランダムに描かれる地形(山)をまたいで，左右2人のプレイヤーがお互いに石を投げあって相手にぶつける，というものです．石の運動は打上げ角度と初速によって放物運動となりますが，地形とともに風向きと風速が毎回ランダムに選ばれ，さらに重力加速度もそのつど変化します．つまり，地球上だけでなく，惑星X上の危険

コラム2.3

データ・コンバージョンのすすめ

パソコン上のソフトというものは，人間の想像力をふりしぼって，あらゆる種類のものが企画・開発されています．この業界に初めて参加したプログラマの卵にとって，対象となる世界と先達の業績の広大さに圧倒されて，いったい自分がこれから何を理解すればこの世界に追いつくのだろうか，と不安になってしまいます．そんな時期に筆者がある人から聞いた，いい話を紹介します．それは，

「ソフトの大半はエディタである」

という意見です．考えてみると，ワープロやデータベース，あるいは事務ソフトや通信ソフト，そしてCAD*ソフトや音楽ソフトといった各ソフトウェアの仕事の本質は，それぞれのフォーマットの世界での「データ処理」であり，入出力の形態を別にすると，ある意味のエディタである，といえるのです．事実，使い慣れたエディタをワープロやデータベースの前半作業用に使ったり，画像データの処理のある部分は，遅いCADツールよりも，バイナリ・データをいったんテキスト形式に変換してでも，高機能エディタで処理したほうが速い，という場合もあります．

この秘密を知ってしまうと，いろいろな開発ターゲットを目前にしたときに，どのような画面にしてどのようなコマンドを設けようか，といった迷いもなくなります．本質は処理されるデータとその処理形態にあるので，まずはデータそのものの表現可能性を検討すればいいわけです．また，何種類かのソフトをこのような視点で組んでみると，パソコン・ソフトのかなりの部分に共通した思想のようなものを実感できるものです．

そして，つぎのステップとして，各種の同一ジャンルのソフト間において，自由にデータのコンバージョンができるようになります．たとえば，ワープロAとワープロBとデータベースCという3種類のソフトを例にとると，これらの間のデータ・コンバージョンというのは6種類も必要になります．そのような専用の変換ソフトがいくつも発売されているくらいなので，それほど簡単というわけでもないのですが，しかしこれは容易に製作できる種類のソフト，勉強用のいい材料，といえます．なにより，対象となったソフトがどのようにデータを表現し，処理しているかを明確に理解できるようになるため，今後の自分にも生きてくる収穫となるのです．この意味で，自分が日常使用しているソフトについて，他ソフトとのデータ・コンバージョン・ソフトというものを製作してみること(それも出来ればいろいろな言語によって)は大いにおすすめできます．

なゲームをシミュレーションしているのです．個々のBasic文法としては，とくにテクニックを使っているところもありません．ただ，毎日少しずつプログラムを作っていったために，一応の「構造化」というか，共通ルーチンの活用を心がけた手法となっています．筆者のクセで，最初にあるようにメイン・ルーチンを極力小さくして，大きなサブ・モジュールを呼ぶだけにしていますが，こうすることが見やすくするコツのようです．

●シンプルなゲーム(2)

リスト2.4もかなり前の作品で，「マスターマインド」と呼ばれるゲームのオリジナル版です．ここではまず，5種類のマークから重複を許して選んだ目標の組合せが，ソフト内にランダムに設定されます．プレイヤーがそれを予想してキー入力すると，指定位置とマークの種類が両方とも合致したときに1$(ドル)コイン，位置はちがってもその種類が合致したものに1¢(セント)コインを示してくれます．たとえばある入力に対して，5点

〔リスト2.4〕ゲーム(2)マスターマインド(別名「1ドル1セント」)①

```
10 'save "a:game2.bas",a
20   DEFINT I-N:DIM MM(5),IM(5),DM(5,5),SD(5),SA(5),SB(5)        ' ﾍﾝｽｳ ｶｸﾎ
30 *START                                    ' ｼｮｷ ｾｯﾃｲ ( ﾊﾟﾗﾒｰﾀ,ｶﾞﾒﾝ )
40   GOSUB *INITIAL.PARA:GOSUB *INITIAL.SCREEN
50 *MAIN                                     ' ﾒｲﾝ ﾙｰﾌﾟ
60   GOSUB *CLOCK:A$=INKEY$:IF A$="" THEN *MAIN          ' ﾄｹｲ + ｷｰ ﾆｭｳﾘｮｸ ﾏﾁ
70   IF A$=CHR$(30) THEN KIN=1:GOTO *EVECHK              ' ｳｴ ﾔｼﾞﾙｼ ｷｰ
80   IF A$=CHR$(28) THEN KIN=2:GOTO *EVECHK              ' ﾐｷﾞ ﾔｼﾞﾙｼ ｷｰ
90   IF A$=CHR$(29) THEN KIN=3:GOTO *EVECHK              ' ﾋﾀﾞﾘ ﾔｼﾞﾙｼ ｷｰ
100   IF A$=CHR$(31) THEN KIN=4:GOTO *EVECHK             ' ｼﾀ ﾔｼﾞﾙｼ ｷｰ
110   IF A$=CHR$(13) THEN KIN=5:GOTO *EVECHK             ' ﾘﾀｰﾝ ｷｰ
120    GOTO *MAIN                                        ' ｿﾚ ｲｶﾞｲﾊ ﾓﾄﾞﾙ
130 *HANTEI                                  ' HIT ﾉ ﾊﾝﾃｲ ﾙｰﾁﾝ
140   SD(5)=0:FOR I=1 TO 5:SA(I)=1:SB(I)=1:NEXT I:FOR I=1 TO 5
150    IF IM(I)=MM(I) THEN SD(5)=SD(5)+10:SA(I)=0:SB(I)=0
160   NEXT I:FOR I=1 TO 5:IF SA(I)=0 THEN 200
170    FOR J=1 TO 5
180     IF SB(J)=1 AND IM(I)=MM(J) THEN SD(5)=SD(5)+1:SB(J)=0:GOTO 200
190    NEXT J
200   NEXT I:IF SD(5)=50 THEN KEKKA=1
210   RETURN
220 *EVECHK                                  ' ｷｰ ｲﾍﾞﾝﾄ ﾖ ﾁｪｯｸ ｽﾙ ﾙｰﾁﾝ
230   LINE (X+1,Y+1)-(X+40,Y+40),0,BF                    ' [?] ｦ ｹｼﾃ
240   CHA=KIN:GOSUB *CHADRAW:IM(PTX+1)=KIN:DM(5,PTX+1)=KIN   ' ｴｦ ｶｲﾃ
250   PTX=PTX+1:IF PTX<5 THEN GOSUB *QUEST:GOTO *MAIN ELSE PTX=0   ' ﾂｷﾞﾉ [?]
260   GOSUB *HANTEI:PTY=4:GOSUB *SCORE                    ' 5ｺﾅﾗ ﾊﾝﾃｲ
270   IF KEKKA=1 THEN GOSUB *HITHIT:GOTO *START           ' HIT ｶﾅ ?
280   TTM=(VAL(RIGHT$(TIME$,2))+2) MOD 60                 ' ｶｽﾞﾚﾀﾗ...
290   TTN=(VAL(RIGHT$(TIME$,2))) MOD 60
300   GOSUB *CLOCK:IF TTM<>TTN THEN 290                   ' ﾄﾘｱｴｽﾞ ﾄｹｲ
310   FOR JJ=0 TO 3:Y=53+JJ*69:LINE (63,Y-6)-(540,Y+50),0,BF   ' ﾜｸｺﾞﾄ ｹｼﾃ
320    PTY=JJ:GOSUB *FLAME:FOR K=1 TO 5:DM(JJ+1,K)=DM(JJ+2,K)   ' ﾜｸｦ ｶｲﾃ
330     Y=53+PTY*69:X=70+(K-1)*50
340     IF DM(JJ+1,K)<>0 THEN CHA=DM(JJ+1,K):GOSUB *CHADRAW     ' ｴｦ ﾋｮｳｼﾞ
350    NEXT K:SD(JJ+1)=SD(JJ+2):PTY=JJ:GOSUB *SCORE        ' ｽｺｱ ﾋｮｳｼﾞ
360   NEXT JJ:PTY=4:GOSUB *CHACLR:Y=53+4*69:LINE (335,Y-6)-(540,Y+50),0,BF
370   PTX=0:PTY=4:GOSUB *QUEST:GOTO *MAIN                 ' ﾂｷﾞﾉ [?]
380 *INITIAL.SCREEN                          ' ｼｮｷ ｶﾞﾒﾝ ﾉ ｾｯﾃｨﾝｸﾞ
390   CONSOLE ,,0:CLS 2:LINE (0,0)-(639,399),,B:LINE (1,1)-(638,398),,B
400    LINE (3,3)-(636,396),,B:LINE (4,4)-(635,395),,B
410    LOCATE 7,1:PRINT "<< M A S T E R M I N D >>";        ' ﾀｲﾄﾙ
420    PRINT "  ---  Produced by Y.Nagashima":COLOR@ (5,1)-(63,1),4
430    LINE (550,50)-(611,380),,B:LINE (551,51)-(610,379),,B
440    X=560:FOR I=0 TO 4:Y=60+I*65
450     LINE (X+29,Y+41)-(X+41,Y+53),,B:LINE (X,Y)-(X+41,Y+41),,B:NEXT I
460    RESTORE *DAT1:READ E:FOR I=1 TO E:READ A,B,C,D:LINE (A,B)-(C,D):NEXT I
470    X=560:FOR I=0 TO 4:Y=60+I*65:CHA=I+1:GOSUB *CHADRAW:NEXT I    ' ﾒﾆｭｰ
480    FOR PTY=0 TO 4:GOSUB *FLAME:NEXT PTY:PTX=0:PTY=4:GOSUB *QUEST:RETURN
490 *SCORE                                   ' ｽｺｱ ヲ 1ﾄﾞﾙｺｲﾝ ﾄ 1ｾﾝﾄｺｲﾝ ﾃﾞ ﾋｮｳｼﾞ ｽﾙ
500    Y=57+PTY*69:PP=SD(PTY+1):FOR PTX=0 TO 4:X=340+PTX*35
510     IF PP>=10 THEN PP=PP-10:CHA=6:GOSUB *CHADRAW:GOTO 530   ' ﾊﾟｼｮ ﾓ Hit
520     IF PP>=1 THEN PP=PP-1:CHA=7:GOSUB *CHADRAW          ' ﾄﾞｺｶﾆ ｱﾙ
530    NEXT PTX:RETURN
540 *CHACLR                                  ' ﾊｺﾉ ﾅｶﾉ ｴｦ ｹｽ ﾙｰﾁﾝ
550    Y=53+PTY*69:FOR PTX=0 TO 4:X=70+PTX*50
560     LINE (X+1,Y+1)-(X+40,Y+40),0,BF:NEXT PTX:RETURN
570 *QUEST                                   ' ﾂｷﾞﾉ ﾆｭｳﾘｮｸ ﾊﾞｼｮﾃﾞ ﾃﾝﾒﾂ ｽﾙ [?]ﾏｰｸ
580    Y=53+PTY*69:X=70+PTX*50:CIRCLE (X+20,Y+14),8,,0,3.58,.8
590     CIRCLE (X+20,Y+14),8,,4.71,6.28,.8:CIRCLE (X+20,Y+14),7,,0,3.58,.8
600     CIRCLE (X+20,Y+14),7,,4.71,6.28,.8:CIRCLE (X+20,Y+14),6,,0,3.58,.8
610     CIRCLE (X+20,Y+14),6,,4.82,6.28,.8:LINE (X+20,Y+21)-(X+20,Y+28)
620     LINE (X+21,Y+21)-(X+21,Y+28):LINE (X+22,Y+21)-(X+22,Y+28)
```

〔リスト2.4〕ゲーム(2)マスターマインド(別名「1ドル1セント」)②

```
630     CIRCLE (X+21,Y+33),2,,,,,F:RETURN
640 *INITIAL.PARA                          ' ﾗﾝﾀﾞﾑﾆ ﾊｲﾁﾖ ｾｯﾄｽﾙ ｼｮｷｶ ﾙｰﾁﾝ
650   RANDOMIZE VAL(RIGHT$(TIME$,2))
660    FOR I=1 TO 5:IM(I)=0:SD(I)=0:SA(I)=0:SB(I)=0:NEXT I       ' ﾊｲﾚﾂ ｸﾘｱ
670    FOR I=1 TO 5:FOR J=1 TO 5:DM(I,J)=0:NEXT J:NEXT I         ' ﾊｲﾚﾂ ｸﾘｱ
680    FOR I=1 TO 5:MM(I)=(RND*.99999)*4+1:NEXT I:RETURN         ' ﾊｲﾁﾖ ｹｯﾃｲ !
690 *FLAME                                 ' ｲﾁｵｳ ﾘｯﾀｲｶﾝ ﾉ ｱﾙ ﾊｺ ﾖ ｴｶﾞｲﾃｲﾙ ﾂﾓﾘ..
700    Y=53+PTY*69:LINE (66,Y-3)-(318,Y+47),,B
710     LINE (321,Y)-(321,Y+50):LINE (69,Y+50)-(321,Y+50)
720     LINE (318,Y+47)-(321,Y+50):LINE (321,Y)-(318,Y-3)
730     LINE (69,Y+50)-(66,Y+47):FOR I=0 TO 4:X=70+I*50
740     LINE (X,Y)-(X+41,Y+41),,B:LINE (X+42,Y+1)-(X+42,Y+42)
750     LINE (X+1,Y+42)-(X+42,Y+42):LINE (X+43,Y+2)-(X+43,Y+43)
760     LINE (X+2,Y+43)-(X+43,Y+43):NEXT I:RETURN
770 *CLOCK                                 ' ﾘｱﾙﾀｲﾑ ﾄｹｲ ﾋｮｳｼﾞ ( ｵﾏｹ )
780    IF CLK$=TIME$ THEN RETURN ELSE CLK$=TIME$                  ' ｲﾍﾞﾝﾄ ｱﾘ ?
790    LOCATE 67,1:PRINT "[";CLK$;"]":RETURN                      ' ﾋｮｳｼﾞ
800 *DAT1                                  ' ｼｮｷ ｶﾞﾒﾝ ﾉ ｽﾒﾉ ｻﾞﾋｮｳ ﾃﾞｰﾀ
810   DATA 17,595,103,595,111,595,103,598,106,595,103,592,106,591,172,599
820   DATA 172,596,175,599,172,596,169,599,172,591,237,599,237,591,237
830   DATA 594,234,591,237,594,240,595,298,595,306,592,303,595,306,598
840   DATA 303,595,306,598,363,598,366,596,369,598,366,596,369,591,369
850   DATA 594,366,591,369,594,372,591,369
860 *HITHIT                                ' HIT ｼﾀﾄｷﾉ ｺﾞﾎｳﾋﾞ ﾙｰﾁﾝ
870   KEKKA=0:LINE (63,45)-(540,320),0,BF
880   TILE$=CHR$(&HAA)+CHR$(&H44)+CHR$(&HAA)*CHR$(&H11)
890   LOCATE 12,4:PRINT "! ! ! C O N G R A T U A T I O N ! ! !"
900   LOCATE 10,18:PRINT "You have finished !!  You are the great GAMER !!"
910   GOSUB *CLOCK:GOSUB *HITDRAW:A$=INKEY$:IF A$="" THEN 910 ELSE RETURN
920 *HITDRAW                               ' ｵｵｷｷﾅ [HIT]ﾉ ﾃﾝﾒｳ ﾙｰﾁﾝ
930   IF VAL(RIGHT$(TIME$,2)) MOD 2=0 THEN LINE (59,99)-(481,261),0,BF:RETURN
940   RESTORE *DAT2:FOR I=1 TO 6:READ A,B,C,D
950   LINE (A,B)-(C,D),,BF,TILE$:NEXT I:RETURN
960 *DAT2                                  ' ｵｵｷﾅ [HIT]ﾉ ｻﾞﾋｮｳ ﾃﾞｰﾀ
970   DATA 60,100,80,260,160,100,180,260,80,170,160,190
980   DATA 260,100,280,260,400,100,420,260,340,100,480,120
990 *CHADRAW                               ' ｷｬﾗｸﾀｰ ﾖ ｾﾝﾀｸｼﾃ ﾋｮｳｼﾞ ｽﾙ
1000   ON CHA GOSUB *CH1,*CH2,*CH3,*CH4,*CH5,*CH6,*CH7:RETURN
1010 *CH1  ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( ﾎﾞｰﾙ )
1020    CIRCLE (X+20,Y+20),15
1030    CIRCLE (X+20,Y+20),14
1040    CIRCLE (X+20,Y+20),11,,5.6,6.2
1050    CIRCLE (X+20,Y+20),9,,5.6,6.2
1060    CIRCLE (X+20,Y+20),10,,6.2,6.2
1070    CIRCLE (X+20,Y+20),11,,5.1,5.3
1080    CIRCLE (X+20,Y+20),9,,5,5.3
1090    CIRCLE (X+20,Y+20),10,,5.3,5.3
1100    PSET (X+23,Y+29)
1110    LINE (X+25,Y+37)-(X+31,Y+37)
1120    LINE (X+23,Y+36)-(X+34,Y+36)
1130    LINE (X+25,Y+35)-(X+36,Y+35)
1140    LINE (X+28,Y+34)-(X+37,Y+34)
1150    LINE (X+30,Y+33)-(X+37,Y+33)
1160    LINE (X+31,Y+32)-(X+36,Y+32)
1170    LINE (X+32,Y+31)-(X+34,Y+31)
1180   RETURN
1190 *CH2  ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( ｻｲｺﾛ )
1200    LINE (X+5,Y+10)-(X+18,Y+5)
1210    LINE (X+18,Y+5)-(X+35,Y+12)
1220    LINE (X+22,Y+17)-(X+35,Y+12)
1230    LINE (X+22,Y+17)-(X+5,Y+10)
1240    CIRCLE (X+20,Y+11),4,,,,,4,F
1250    LINE (X+22,Y+17)-(X+22,Y+36)
1260    LINE (X+5,Y+10)-(X+5,Y+28)
1270    LINE (X+22,Y+36)-(X+5,Y+28)
1280    LINE (X+35,Y+12)-(X+35,Y+30)
1290    LINE (X+22,Y+36)-(X+35,Y+30)
1300    LINE (X+23,Y+36)-(X+36,Y+30)
1310    LINE (X+24,Y+37)-(X+36,Y+31)
1320    LINE (X+26,Y+37)-(X+37,Y+31)
1330    LINE (X+27,Y+37)-(X+37,Y+32)
1340    CIRCLE (X+31,Y+20),2.5,,,,2,F
1350    CIRCLE (X+27,Y+27),2.5,,,,2,F
1360    CIRCLE (X+19,Y+29),2.5,,,,2,F
1370    CIRCLE (X+14,Y+23),2.5,,,,2,F
1380    CIRCLE (X+9,Y+17),2.5,,,,2,F
1390   RETURN
1400 *CH3       ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( ﾌﾞﾄﾞｳ )
1410    CIRCLE (X+23,Y+20),3
1420    CIRCLE (X+18,Y+22),3
1430    CIRCLE (X+19,Y+16),3
1440    CIRCLE (X+13,Y+18),3
1450    CIRCLE (X+12,Y+24),3
1460    CIRCLE (X+20,Y+28),3
1470    CIRCLE (X+14,Y+30),3
1480    CIRCLE (X+26,Y+25),3
1490    CIRCLE (X+8,Y+29),3
1500    CIRCLE (X+10,Y+34),3
1510    CIRCLE (X+29,Y+20),3
1520    CIRCLE (X+26,Y+15),3
1530    CIRCLE (X+21,Y+11),3
1540    CIRCLE (X+14,Y+12),3
1550    LINE (X+25,Y+12)-(X+30,Y+7)
1560    LINE (X+24,Y+11)-(X+29,Y+6)
1570    LINE (X+24,Y+4)-(X+35,Y+9)
1580    LINE (X+24,Y+3)-(X+35,Y+10)
1590    LINE (X+37,Y+12)-(X+35,Y+9)
1600    LINE (X+37,Y+13)-(X+35,Y+10)
1610    CIRCLE (X+23,Y+35),8,,,,.25,F
1620   RETURN
1630 *CH4       ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( ｺﾏ )
1640    CIRCLE (X+20,Y+20),14,,,,.4
1650    CIRCLE (X+20,Y+21),14,,3.14,6.28,.4
1660    CIRCLE (X+20,Y+22),14,,3.14,6.28,.4
1670    CIRCLE (X+20,Y+26),10,,3.14,6.28,.4
1680    CIRCLE (X+20,Y+30),6,,3.14,6.28,.4
1690    CIRCLE (X+20,Y+33),2,,3.14,6.28,3,F
1700    CIRCLE (X+20,Y+20),12,,,,,.3
1710    CIRCLE (X+20,Y+20),10,,,,.25
1720    CIRCLE (X+20,Y+20),7,,,,.25
1730    LINE (X+18,Y+20)-(X+22,Y+20)
1740    LINE (X+19,Y+20)-(X+21,Y+6),,BF
1750    CIRCLE (X+34,Y+25),5,,5,6.28,
1760    CIRCLE (X+31,Y+24),5,,5,6.28,
1770    CIRCLE (X+7,Y+15),5,,1.5,3.5,,.8
1780    CIRCLE (X+9,Y+17),5,,1.5,3.5,,.8
1790   RETURN
```

```
1800 *CH5 ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( ｼﾝｾｻｲｻﾞｰ )
1810    LINE (X+4,Y+7)-(X+36,Y+35),,B
1820    LINE (X+5,Y+36)-(X+37,Y+36),,B
1830    LINE (X+37,Y+8)-(X+37,Y+36),,B
1840    LINE (X+6,Y+18)-(X+10,Y+33),,B
1850    LINE (X+8,Y+18)-(X+16,Y+27),,BF
1860    LINE (X+14,Y+18)-(X+10,Y+33),,B
1870    LINE (X+14,Y+18)-(X+18,Y+33),,B
1880    LINE (X+22,Y+18)-(X+18,Y+33),,B
1890    LINE (X+20,Y+18)-(X+32,Y+27),,BF
1900    LINE (X+22,Y+18)-(X+26,Y+33),,B
1910    LINE (X+30,Y+18)-(X+26,Y+33),,B
1920    LINE (X+30,Y+18)-(X+34,Y+33),,B
1930    LINE (X+12,Y+19)-(X+12,Y+27),0
1940    LINE (X+24,Y+19)-(X+24,Y+27),0
1950    LINE (X+28,Y+19)-(X+28,Y+27),0
1960    CIRCLE (X+10,Y+12),4,,,,,8
1970    CIRCLE (X+10,Y+12),2
1980    CIRCLE (X+30,Y+12),4,,,,,8
1990    CIRCLE (X+30,Y+12),2
2000    LINE (X+16,Y+10)-(X+24,Y+14),,B
2010  RETURN
2020 *CH6       ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( 1ﾄﾞﾙ )
2030    CIRCLE (X+16,Y+17),14,,,,,6
2040    CIRCLE (X+16,Y+17),12,,,,,5
2050    CIRCLE (X+16,Y+18),14,,3.1416,6.283,,6
2060    CIRCLE (X+16,Y+19),14,,3.1416,6.283,,55
2070    CIRCLE (X+16,Y+20),14,,3.1416,6.283,,5
2080    CIRCLE (X+17,Y+15),5,,0,4.5,,35
2090    CIRCLE (X+15,Y+19),5,,0,1.7,,35
2100    CIRCLE (X+15,Y+19),5,,3.1416,6.283,,35
2110    CIRCLE (X+17,Y+15),4,,0,4.5,,4
2120    CIRCLE (X+15,Y+19),4,,0,1.7,,4
2130    CIRCLE (X+15,Y+19),4,,3.1416,6.283,,4
2140    LINE (X+17,Y+12)-(X+12,Y+22)
2150    LINE (X+19,Y+12)-(X+14,Y+23)
2160  RETURN
2170 *CH7       ' ｷｬﾗｸﾀｰ ﾖ ｶﾞﾒﾝﾆ ｴｶﾞｸ ( 1ｾﾝﾄ )
2180    CIRCLE (X+16,Y+17),9,,,,,6
2190    CIRCLE (X+16,Y+17),7,,,,,45
2200    CIRCLE (X+16,Y+18),9,,3.1416,6.283,,6
2210    CIRCLE (X+16,Y+19),9,,3.1416,6.283,,6
2220    CIRCLE (X+16,Y+17),3,,1.3,4.5,,35
2230    CIRCLE (X+16,Y+17),2,,1.5,4.5,,35
2240  RETURN
```

のうち1点が位置も種類も当たり，あと2点は位置はちがっていて種類が合っていると，“1$2¢”というコインの絵が並ぶのです．これをくり返しながら，“5$”をなるべく早く出すという，けっこう頭の体操になるゲームです．リストのうしろ半分は，じつは5種類のマーク(ボール，サイコロ，ブドウ，コマ，シンセサイザ)と2種類のコインの絵を描くのに使われています．ここではかなり意地になって，LINE文とCIRCLE文だけで，相当にこった絵を作っていて，プログラム製作時間のほとんどが，ここの試行錯誤に費やされました．判定の処理はなるべくシンプルにと，いちいち配列から配列へとパラメータを移動させています．390行あたりを変えるだけで，PC-9801シリーズならばすべてで走るはずです．

●さらに高度な入力ルーチン

リスト2.5にあるのは，実験用につくりかけたプログラムの未完成品です．ここではメニューの選択方法として，上下の矢印キーによってメニュー項目自体を反転表示させる，というかなり本格的な方法をとっています．1480～1570行がこのポイントで，パソコンによってコマンドが異なりますが，同じような手法は使えるでしょう．また，このプログラムでは非常に単純なRS-232-Cの処理も含まれています．未完成とはいってもこのソフトはちゃんと走り，今後必要があればまた項目を追加して使用できる，というものなので，むしろ開発途上品といえるのかもしれません．

〔リスト2.5〕メニューを選択して実行するプログラム(の開発途上品)のリスト①

```
10 'save "select.bas",a
20    GOSUB *PARAINIT:GOSUB *PORTINIT:GOSUB *DISPINIT:COM ON:GOTO *MAIN
30 *MAIN                ' ﾒｲﾝ ﾙｰﾌﾟ : ｳｴ ﾄ ｼﾀ ﾆ ｳｺﾞｲﾃ、ﾘﾀｰﾝ ｷｰ ﾖ ﾏﾂ
40    GOSUB *CLOCK:A$=INKEY$:IF A$="" THEN *MAIN                ' ｷｰ ｲﾍﾞﾝﾄ ﾏﾁ
50    IF A$=CHR$(30) THEN GOSUB *UPMOVE:GOTO *MAIN              ' ｶｰｿﾙ ｳｴ
60    IF A$=" " OR A$=CHR$(31) THEN GOSUB *DOWNMOVE:GOTO *MAIN  ' ｶｰｿﾙ ｼﾀ
70    IF A$=CHR$(253) OR A$=CHR$(13) THEN 90                    ' ﾘﾀｰﾝ ｷｰ ?
80    IF A$=CHR$(&H1B) THEN *FINISH ELSE *MAIN                  ' ｴｽｹｰﾌﾟ ?
90    ON MASKOLD GOSUB *TRANS,*SIOSIM,*MOUSESIM,*FREMEM,*TEXTEDIT,*DUM
100    GOSUB *DISPINIT:GOTO *MAIN
110 *DUM:CLS 2:LOCATE 20,10:PRINT "I'm sorry."                   ' ( ﾀﾞﾐｰ )
120     LOCATE 20,13:PRINT "This screen-mode is non-finished."
130     IF INKEY$<>"" THEN RETURN ELSE 130
140 *CLOCK              ' ﾘｱﾙﾀｲﾑ ﾄｹｲ ﾋｮｳｼﾞ ( ｵﾏｹ )
150    IF CLK$=TIME$ THEN RETURN ELSE CLK$=TIME$
160    LOCATE 65,3:PRINT "[";CLK$;"]":LOCATE 0,25:RETURN
170 *FREMEM             ' ﾒﾓﾘ ﾉ ﾉｺﾘ ﾖｳﾘｮｳ ﾉ ﾃﾞｨｽﾌﾟﾚｲ
180    CLS 2:LOCATE 12,1:PRINT "Remain Memory Bytes Display":GOSUB *SUBTITLF
190    PRINT:PRINT:FOR I=0 TO 3:PRINT
200    PRINT "                  FRE (";I;") =";FRE(I);"bytes":NEXT I
210    PRINT:PRINT "                  Remain Disc [A:] =";DSKF("a:");"bytes"
220    PRINT:PRINT "                  Date - Time = ";DATE$;" - ";TIME$
230    IF INKEY$=CHR$(&H1B) THEN RETURN ELSE 230
240 *TEXTEDIT           ' ｶﾝﾀﾝﾅ ｱｽｷｰ ﾃｷｽﾄ ｴﾃﾞｨﾀ ( ﾊﾟﾗﾒｰﾀ ﾌｧｲﾙ ﾖｳ )
     ⋮                                          (250行～1030行省略)
```

〔リスト2.5〕メニューを選択して実行するプログラム(の開発途上品)のリスト②

```
      :
1040 *MOUSESIM            ' ｼﾞｯｹﾝﾖｳ ﾏｳｽ ｲﾝﾀｰﾌｪｰｽ ﾃｽﾄ
1050    CLS 2:LOCATE 12,1:PRINT "MOUSE Simulation Controller":GOSUB *SUBTITLE
1060    ON COM GOSUB *SIOINT
1070   *MOUSELOOP                 ' ﾃﾞｰﾀ ｦ ﾁｪｯｸｼﾃ ﾋｮｳｼﾞ ｽﾙ ﾙｰﾌﾟ
1080     C$=INKEY$:IF C$="" THEN *MOUSELOOP
1090     IF C$=CHR$(&H1B) THEN RETURN
1100     IF ASC(C$)=30 THEN MD$="up":GOSUB *EX:GOTO *MOUSELOOP
1110     IF ASC(C$)=31 THEN MD$="down":GOSUB *EX:GOTO *MOUSELOOP
1120     IF ASC(C$)=29 THEN MD$="left":GOSUB *EX:GOTO *MOUSELOOP
1130     IF ASC(C$)=28 THEN MD$="right":GOSUB *EX:GOTO *MOUSELOOP
1140     IF ASC(C$)=8 THEN MD$="rightsw":GOSUB *EX:GOTO *MOUSELOOP
1150     IF ASC(C$)=18 THEN MD$="leftsw":GOSUB *EX:GOTO *MOUSELOOP
1160     PRINT "-";:GOTO *MOUSELOOP
1170   *EX:PRINT "<"+MD$+">";:PRINT #1,MD$+CHR$(&HA);:RETURN
1180 *SIOSIM                ' ｼﾞｯｹﾝﾖｳ RS232C ｲﾝﾀｰﾌｪｰｽ ﾃｽﾄ
1190    CLS 2:LOCATE 12,1:PRINT "RS232C Direct Transmitter":GOSUB *SUBTITLE
1200    ON COM GOSUB *SIOINT
1210   *SIOLOOP                   ' ﾃﾞｰﾀ ｦ ﾋｮｳｼﾞ ｽﾙ ﾙｰﾌﾟ
1220     C$=INKEY$:IF C$="" THEN *SIOLOOP
1230     IF C$=CHR$(&H1B) THEN RETURN
1240     IF C$=CHR$(&HD) THEN C$=CHR$(&HA)
1250     PRINT C$;:PRINT #1,C$;:GOTO *SIOLOOP
1260   *SIOINT                    ' RS232C ﾜﾘｺﾐ ｼｮﾘ ﾙｰﾁﾝ
1270     IF LOC(1)=0 THEN RETURN
1280     CD$=INPUT$(1,#1):IF CD$=CHR$(&HD) THEN CD$=CHR$(&HA)
1290     PRINT CD$;:GOTO *SIOINT
1300   *SIOPASS                   ' RS232C ﾊﾟｽ !
1310     IF LOC(1)=0 THEN RETURN
1320     CD$=INPUT$(LOC(1),#1):RETURN
1330 *TRANS                 ' ｼﾞｯｹﾝ ﾎﾟｰﾄﾞ ﾖｳ ﾄｸｼｭ ｿｳｼﾝ ﾌｫｰﾏｯﾄ ｲﾝﾀｰﾌｪｰｽ ﾃｽﾄ
1340    CLS 2:LOCATE 12,1:PRINT "ISDN Direct Transmitter":GOSUB *SUBTITLE
1350   *TRANSLOOP                 ' ﾃﾞｰﾀ ｦ ﾁｪｯｸｼﾃ ﾋｮｳｼﾞ ｽﾙ ﾙｰﾌﾟ
1360     C$=INKEY$:IF C$="" THEN *TRANSLOOP
1370     IF C$=CHR$(&H1B) THEN RETURN
1380     IF C$=":" THEN GOSUB *CX:GOTO *TRANSLOOP
1390     IF C$=CHR$(13) OR C$=" " THEN C$=":":GOSUB *CX:GOTO *TRANSLOOP
1400     IF ASC(C$)>&H2F AND ASC(C$)<&H3A THEN GOSUB *CX:GOTO *TRANSLOOP
1410     IF ASC(C$)>&H40 AND ASC(C$)<&H47 THEN GOSUB *CX:GOTO *TRANSLOOP
1420     IF ASC(C$)>&H60 AND ASC(C$)<&H67 THEN *DX
1430     PRINT "-";:GOTO *TRANSLOOP
1440   *DX:C$=CHR$(ASC(C$)-&H20):GOSUB *CX:GOTO *TRANSLOOP
1450   *CX:PRINT C$;:PRINT #1,C$;:RETURN
1460 *FINISH                ' ｼｭｳﾘｮｳ ( for debug )
1470    CONSOLE 0,25,1:CLS 2:LIST:END
1480 *UPMOVE                ' ﾒﾆｭｰ ｶｰｿﾙ ｳｴ ｲﾄﾞｳ ﾙｰﾁﾝ
1490    MASKNEW=MASKNEW-1:IF MASKNEW=0 THEN MASKNEW=KINDS
1500    GOSUB *CURSORWRITE:RETURN
1510 *DOWNMOVE              ' ﾒﾆｭｰ ｶｰｿﾙ ｼﾀ ｲﾄﾞｳ ﾙｰﾁﾝ
1520    MASKNEW=MASKNEW+1:IF MASKNEW>KINDS THEN MASKNEW=1
1530    GOSUB *CURSORWRITE:RETURN
1540   *CURSORWRITE                ' ｶｰｿﾙ ﾏｰｸ ｦ ｶｸ ﾙｰﾁﾝ ( ﾅｶﾞｻｶﾞ ﾉﾋﾞﾁﾁﾞﾐ ｽﾙ ! )
1550     COLOR@ (P1-1,MASKOLD+P2)-(LEN(B$(MASKOLD))+P1,MASKOLD+P2),0
1560     COLOR@ (P1-1,MASKNEW+P2)-(LEN(B$(MASKNEW))+P1,MASKNEW+P2),4
1570     MASKOLD=MASKNEW:RETURN
1580 *DISPINIT              ' ｼｮｷ ｶﾞﾒﾝ ｦ ｶｸ
1590    CLS 2:CONSOLE 0,25:LOCATE 12,1:PRINT "   T E S T    P A N E L   "
1600    GOSUB *SUBTITLF
1610    ON COM GOSUB *SIOPASS
1620    LOCATE 4,3:PRINT "=== MAIN MENU ===":P1=13:P2=3
1630    FOR I=1 TO KINDS:LOCATE P1,I+P2:PRINT B$(I):NEXT I
1640    GOSUB *CURSORWRITE:RETURN
1650 *PORTINIT              ' RS232C ﾎﾟｰﾄ ｦ ｼｮｷｶ ｽﾙ
1660    OPEN "com1:n81nn" AS#1:RETURN
1670 *PARAINIT              ' ﾊﾟﾗﾒｰﾀｰ ｼｮｷｶ
1680    WIDTH 80,25:CONSOLE ,,0
1690    KINDS=20:DIM B$(KINDS)
1700     RESTORE *DATA.MAIN.MENU:FOR I=1 TO KINDS:READ B$(I):NEXT I
1710    TEXTS=50:DIM T$(TEXTS)
1720    TONE=100:DIM T(TONE,5):PATT=1
1730    MASKOLD=1:MASKNEW=1:RETURN
1740 *DATA.MAIN.MENU        ' ﾒﾆｭｰ ｺｳﾓｸ ﾘｽﾃｨﾝｸﾞ ｴﾘｱ !
1750   DATA " (1) Direct ISDN Transmitter (for debug)"
1760   DATA " (2) Direct RS232C Transmitter (for debug)"
1770   DATA " (3) [Mouse] Simulation (for debug)"
1780   DATA " (4) Remain Memory Bytes Display"
1790   DATA " (5) Universal Text Editor (ASCII File)"
1800   DATA " (6) < not defined >"
1810   DATA " (7) < not defined >"
```

```
1820    DATA " (8) "
1830    DATA " (9) "
1840    DATA "(10) "
1850    DATA "(11) "
1860    DATA "(12) "
1870    DATA "(13) "
1880    DATA "(14) "
1890    DATA "(15) "
1900    DATA "(16) "
1910    DATA "(17) "
1920    DATA "(18) "
1930    DATA "(19) "
1940    DATA "(20) "
1950 *SUBTITLE
1960     LOCATE 50,1:PRINT "produced by Y.Nagashima"
1970     COLOR@ (2,1)-(77,1),4:CONSOLE 5,17:LOCATE 0,5:RETURN
1980 *SUBTITLF
1990     LOCATE 50,1:PRINT "produced by Y.Nagashima"
2000     COLOR@ (2,1)-(77,1),4:RETURN
```

機械語ルーチンによる高速化

さて，さきの例のようなシステムをいくつか開発してみると，最初に問題となってくるのは，Basic の最大の欠点である**処理スピードの遅さ**でしょう．たとえば温度の表示にしても，たんに PRINT 文のキャラクタ表示ならともかく，

「遠くからでも見えるように，画面一杯の大きな数字で電光掲示板のようにしたい」

「視覚的にアピールする温度計の絵を描きたい」

といった要求の場合を考えてみましょう(**図2.4**)．もし Basic のグラフィック命令を組み合わせて図形をペイントすると，この速度は著しく遅いため，メイン・ルーチン自体の処理能力を大幅に低下させて，もたもたした非常に見苦しい画面となってしまいます．

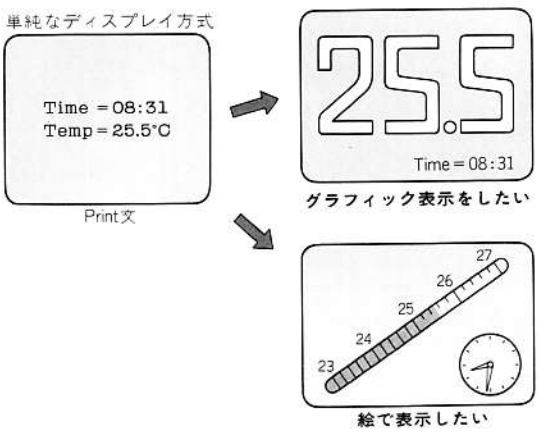
ここで効いてくるのが，Basic 中の**機械語ルーチン**の使用です．これは，インタプリタで逐一翻訳しながらの実行でなく，あらかじめアセンブラなどで書かれている機械語ルーチンをメモリにロードしておき，その処理に限って高速の機械語処理を行うものです(**リスト2.6**)．これには2種類あり，

① 簡単な処理を関数の形式で呼び出す

② 一連の処理をサブルーチンとして呼び出す

というものに分かれます．いずれにしても，この手法

〔図2.4〕表示の仕方によってイメージが変わる！



〔リスト2.6〕
機械語ルーチンの活用で高速化する
(サブルーチンとして呼びだす例)

```
1000 'save "sample",a
1010    DEF SEG=&H8000              ' 機械語ルーチン領域の確保
1020    BLOAD "b:subrt.bin",0       ' 機械語ルーチンをLoad
1030    SUBRT=0                     ' 機械語ルーチン先頭Address
1040    TEMP=&HE0D0                 ' Port Address
1050 '##### Main Loop #####
1060    X1=INP(TEMP)                ' Data Input
1070    X2=INP(TEMP)
1080    IF(X1<>X2) THEN GOTO 1060   ' Normal Data ?
1090    CALL SUBRT(X1)              ' 機械語ルーチンをCall
1100    GOTO 1050
```

を本格的に取り入れるには，パソコンに使われているCPUのアセンブリ言語と，パソコンのハードウェアについての理解が必要となります．もっとも，すでに完成されている処理ルーチンを流用したり，雑誌や解説書のサンプルにしたがって機械語をデータ文として打ち込む，という場合には，機械語ルーチンの実現が容易に可能で，その部分は相当に高速化されます．

Basicコンパイラによる高速化

使用するパソコンに**Basicコンパイラ**というツールがサポートされている場合には，別の方法でBasicシステムを高速化できます．この場合は，パソコンの**DOS***という環境を理解・活用する必要がありますが，ここではDOSじたいの説明は省略して，コンパイラにおける開発の手順を示すことにします．

まず，あらかじめ通常のBasicインタプリタで開発・デバッグして完成したプログラムを

```
save "test.bas", a
```

のように，テキスト形式(ASCII形式，文字形式ともいう)としてセーブします．これにより，Basicプログラムは圧縮された特別な形式でなく，コンパイラが参照できる標準的なファイルとなります．その後，DOSにもどり，Basicコンパイラという実行形式のツールを起動し，Basicプログラムをソース・プログラムとして指定します．エラーがなければオブジェクト・ファイルが作成され，さらにリンカというツールによって実行形式のプログラムへと変換してやれば，DOS上の実行形式の高速なプログラムとして完成します(**リスト2.7**)．ここで，プログラムによっては，実行の際に同じドライブ上に，Basicコンパイラ用の実行モジュール・ファイルをおくことが必要な場合もあるので，注意します．

ただしこの場合，慎重に対応しなければならない注意点もいくつかあります．まず，すべてのBasicコマンドが1対1に，完全に機械語処理に変換されるわけではありません．この点についてはマニュアルによって，インタプリタとコンパイラの相違点を十分に理解しておく必要があります．また，インタプリタでは実行されないでいた文に，構造上のバグが潜んでいて，インタプリタでは問題なく実行できるプログラムがコンパイルできない，という種類のトラブルもよく経験します．さらに，システムへの不法アクセスを生むバグによるエラーというのは，インタプリタなら

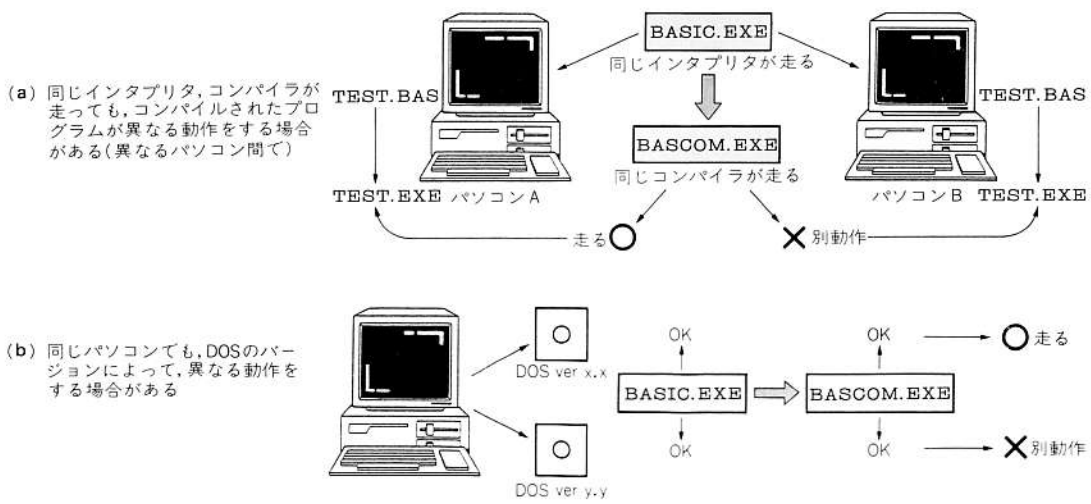
```
"I/O Error"
```

というメッセージでストップするだけなのに，ちゃんとコンパイルされた実行形式プログラムが暴走する場合には，キーボードで止められない場合も多く，泣く泣くリセットすることになります．さらに，DOSのバージョンやパソコンの機種による相性によって，インタプリタでは表面化しなかった動作の相違を多く経験しています(**図2.5**)．このように，Basicコンパイラは手軽な印象のわりに，実際にはかなりの注意が必要で，ある程度の経験と慣れが要求されるようです．

〔リスト2.7〕Basicコンパイラ活用の手順

```
>basic

load "sample"
Ok
list
1000   'save "sample",a
1010     PRINT:PRINT " Start Time = ";TIME$
1020     FOR I=1 TO 10000
1030         J=SIN(I)
1040     NEXT I
1050     PRINT:PRINT "Finish Time = ";TIME$:PRINT
1060     IF INKEY$="" THEN 1060
Ok
run

 Start Time = 18:30:05      ) Basicインタプリタ
Finish Time = 18:30:59      ) 実行時間＝54秒

Ok
system

>basicc                          ← コンパイラの起動

Personal Computer BASIC Compiler Version 1.0

Source Filename [.BAS]: sample
Object Filename [SAMPLE.OBJ]:
Source Listing [NUL.LST]:

20138 Bytes Available
19812 Bytes Free

    0 Warning Error(s)    } コンパイル正常終了
    0 Severe  Error(s)    }

>link                            ← リンカの起動

Personal Computer Linker Version 2.4

Object Modules [.OBJ]: sample
Run File [SAMPLE.EXE]:
Source Listing [NUL.MAP]:
Libraries [.LIB]:

>sample

 Start Time = 18:36:19      ) Basicコンパイラ
Finish Time = 18:36:29      ) 実行時間＝10秒

>
```

〔図2.5〕Basic コンパイラの注意点



DOS 環境やバッチ処理を活用する

コンパイラを使うかどうかは別にしても，DOS コマンドや**バッチ・プログラム**などの DOS のシステムが使いこなせてくると，全体の操作性をかなり向上させることができます．たとえば，パソコンの電源をオンにしたら自動的にシステムの初期設定を行い，必要なファイルを必要なディレクトリから呼び込み，所定の作業環境に入ったうえで，Basic を呼んでプログラムの実行を自動的に開始する，というようなシステムが簡単に構築できます(**リスト2.8**)．

また，いくつかのメニューからプログラムを選択して実行するとか，プログラムが終了すると必要なファイルに所定の処理を行ってから電源が自動的に切れるとか，ある条件に該当するファイルだけを検索して処理する，といったことも，DOS に用意されているツールを組み合わせることによって簡単に実現できます．最近では，エスケープ・シーケンスを豊富に使ったり，仮パラメータや環境文字列を何度も書きかえるなどの，バッチ・プログラムの究極ワザを競ったような本を多く見かけますが，アイデアを自分のシステムのヒントにする，という意味では，それなりに結構おもしろいと思います．

さらに，DOS の環境を使いこなす，という意味では，専用デバイス・ドライバを自作して登録したり，システム常駐ユーティリティを自作したりする楽しみもあります．筆者の経験では，このように DOS の環境をカスタマイズしていくと，パソコンに使われている，という感じから，**パソコンを使ってやっている**，という感じになって，より快適に作業が進む気がします．

〔リスト2.8〕システムを自動実行するためのバッチ処理の例

```
>type a:¥autoexec.bat

echo off
echo     システム初期設定開始.....
cd b:¥test1
copy a:¥sys¥*.exe b:
copy a:¥sys¥*.com b:
copy a:¥dat¥*.bas b:
copy a:¥dat¥*.dat b:
b:
basic sample
a:
dir b:¥dat¥*.* /w

>
```

なぜ C 言語なのか

Basic のシステムでスピードのつぎに問題となるのが，ソフトウェアの**構造化の限界**です．実際にプログラムしてみるとわかりますが，**Basic ではいくらサブルーチンに分割しても，1000 ステップ程度を越えると全体の見通しが非常に悪くなります．**また，メニュー形式で処理を場合分けするときも，メニューの項目と階層が何十段にも増えると，プログラムがかなり複雑になります．さらに，変数の制限やデバッグ・修正の

効率も下がって，ある程度以上の規模では，マイコン・システムのソフトウェア環境として，非常に使いにくい状態になります．たとえば，新規の仕事として，

「そこそこの機能のワープロ・ソフトを，
Basicで開発しなさい」

といわれたら，おそらく誰もが使用言語の再検討を提案せざるを得ないでしょう．便利にみえるインタプリタのスクリーン・エディタも，プログラムが大規模になると，もっとも単純なエディタのEDLINにも含まれている簡単な編集機能すらないために，かえって不便な場合も出てきます．

そこで登場するのが，パソコンのDOS上の高級言語コンパイラです．現在もっともポピュラな**C言語**などは，あまりに種類が多くて選ぶのに困るほどです．Cの流行の原因には，**構造化プログラミングの記述に適している**ことや，**比較的CPUの細かい処理を記述できる**点もありますが，最近多くなった，**使いやすいDOS上のエディタを使って開発できる**，という理由もあると思います．モジュール分割した各ルーチンをエディタで同時にオープンして，比較しながら製作・編集・修正できる，というのは効率を高めるための重要な要因なのです．

C言語ではファイル関係もよくサポートされているので，ディレクトリ構造まで含めた高度なファイル管理も可能です．また，たいていのCコンパイラでは，DOSのバッチ処理・DOSコマンド・実行形式プログラムの呼出しや，DOS内のファンクション・コール呼出しといった機能をサポートしており，Cで実現できる機能には非常に大きなものがあります．言いかえると，よほど高速の処理が必要でないかぎり，パソコンで出来ることのほとんどを開発できる環境といえます(さきのBasicの例をCで記述したものを，**リスト2.9**に示します)．筆者の場合，個人的には必ずしもCを信奉してはいませんが，気分転換に簡単なMS-DOS*ユーティリティを書いたり，メディア変換/データ変換のためのツールをCで書くことは，もはや空気のように日常的になっています．

C言語のもう一つのメリットは，各処理ルーチンをモジュール化したライブラリとして，将来も使用できたり，すでにある各ルーチンのソースを手直しして自分のルーチンとして活用できるという点です．コンパクトなライブラリ・モジュールとしてコンパイルされているルーチンを流用するのは，アセンブラでも同様のことですが，プログラムが小さなルーチン(関数)に分割されているC言語では，そのような各関数のソースを適当に自分用に訂正しつつ取り込む，という「**ソフトウェア部品**」的なメリットがあります．このように，プログラムが複雑・巨大になればなるほど，C言語の構造化のメリットは鮮明になってきます．またCは高級言語ですから，理論的にはどんなCPUのアセンブラにも変換できる，というのも大きな魅力です．

Cプログラムの使用例

Cを使った小さなツールの例を，いくつかのサンプルで紹介しましょう．もちろんこれまでには，マウス・ドライバやグラフィック・ドライバを含むような，巨大なソフトを製作したこともありますが，ここではむしろ，身の回りに置いてあるようなツールを扱うことにします．

●まず使用例から

リスト2.10にあるのが，ここで紹介するツールの使用例です．まずは(1)のように，データ用のサンプルとしてtest.binという，0から63まで順に並んだだけの64バイトのファイルを用意します．(2)のように，ダンプ内容も見ておきます．ここで(3)にある，bin2ascというツールが第1のもので，名前のとおり，バイナリ・ファイルをASCIIファイルに変換します．これは，ちょうど(2)のダンプ・リストの1文字1文字を，ASCII形式で見たようなものです．変換後は(4)のように，test.ascという128バイトのファイルができて，(5)のダンプ・リストのように，たしかに文字形式

になっています．このファイルであれば，(6)のようにtypeコマンドによっても表示できるのがわかります．さて，もう一つのツールは(7)のtail-addというものです．これは(8)のように1バイト増えるような処理を行うもので，(9)のダンプ・リストからわかるように，ファイルの末尾に1バイトのファイル終了マークを付加します．これとペアになるのが，(11)のtail-cutというツールで，(13)で見ると最後の1バイトがカットされているのがわかります．もう一つのツールは(14)のasc2binというもので，その名の通り，ASCII形式のファイルを2バイトずつ読みこんで，バイナリ・ファイルへと変換します．こうしていったんASCII形式に変換してからバイナリ形式に戻したtest.newと，元のtest.binとを，(17)のようにDOSのcompユーティリティで比較します．一致するがファイルの終わりがない，というメッセージが出ました．そこで両方のファイルに再度tail-addをほどこすと，(20)のようにOKが出ました．内容はもちろん，(21)のようなファイルに

〔リスト2.9〕
例1をC言語で書いてみる(リスト2.1のBasicプログラムと完全コンパチではない)

```
#include  <stdio.h>
#define   temp_port     0xe0d0
#define   power_port    0xe0d2
extern    unsigned _rax,_rcx,_rdx;

main(){
        char c;
        int fd,x1,x2;
        int limit=50;
        long sum=0L;
        int i=0;
        int j,k=0;
        unsigned old,t;
        float mean;

        _outb(0xff,power_port);
        fd=fopen("c:test.dat","w");   ←― ファイルのオープン
        old=time_check();
        for(;;){
                for(j=0;j++;j<100) {k++;}
                c=csts();
                if(c!=0){
                        fclose(fd);
                        exit();
                }
                x1=_inb(temp_port);   } データの2度取込み
                x2=_inb(temp_port);   }
                if(x1==x2){
                        sum=sum+(long)x1;
                        i++;
                        mean=(float)sum/(float)i;   ←― 平均値の計算
                        t=time_check();
                        if(old!=t){
                                old=t;
                                printf("¥nTime = %2d:%2d",t/256,t%256);
                                printf(" , Temp = %f",mean);
                                i=0;
                                sum=0L;
                        }
                        if((i%50)==0){
                                putw(x1,fd);
                        }
                        if(x1>limit){
                                putchar(0x07);   }
                                putchar(0x07);   }
                                putchar(0x07);   } beep (ブザーをならす)
                                putchar(0x07);   }
                                putchar(0x07);   }
                                _outb(0x00,power_port);
                                fclose(fd);
                                exit();
                        }
                }
        }
}

time_check(){
        _rax=0x2c00;   }
        _doint(0x21);  } 時刻データを取り出すDOSコール
        return(_rcx);  }
}
```

〔リスト2.10〕リスト2.11の使用例

```
(1) C>
    C>dir test.*

    TEST     BIN        64

(2) C>dump test.bin

    000000  00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  ........ ........
    000010  10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F  ........ ........
    000020  20 21 22 23 24 25 26 27 - 28 29 2A 2B 2C 2D 2E 2F   !"#$%&' ()*+,-./
    000030  30 31 32 33 34 35 36 37 - 38 39 3A 3B 3C 3D 3E 3F  01234567 89:;<=>?

(3) C>bin2asc

      <<< BIN to ASC >>>  Data Conversion Program  :  Produced by Y.Nagashima

            Source [Binary] File Name = test.bin


            Destination [ASCII] File Name = test.asc


                              ----- Conversion is completely finished ...

(4) C>dir test.*
    TEST     ASC       128
    TEST     BIN        64

(5) C>dump test.asc

    000000  30 30 30 31 30 32 30 33 - 30 34 30 35 30 36 30 37  00010203 04050607
    000010  30 38 30 39 30 41 30 42 - 30 43 30 44 30 45 30 46  08090A0B 0C0D0E0F
    000020  31 30 31 31 31 32 31 33 - 31 34 31 35 31 36 31 37  10111213 14151617
    000030  31 38 31 39 31 41 31 42 - 31 43 31 44 31 45 31 46  18191A1B 1C1D1E1F
    000040  32 30 32 31 32 32 32 33 - 32 34 32 35 32 36 32 37  20212223 24252627
    000050  32 38 32 39 32 41 32 42 - 32 43 32 44 32 45 32 46  28292A2B 2C2D2E2F
    000060  33 30 33 31 33 32 33 33 - 33 34 33 35 33 36 33 37  30313233 34353637
    000070  33 38 33 39 33 41 33 42 - 33 43 33 44 33 45 33 46  38393A3B 3C3D3E3F

(6) C>type test.asc
    000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F202122232425
    262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F

(7) C>tail-add test.asc

(8) C>dir test.*

    TEST     ASC       129
    TEST     BIN        64

(9) C>dump test.asc

    000000  30 30 30 31 30 32 30 33 - 30 34 30 35 30 36 30 37  00010203 04050607
    000010  30 38 30 39 30 41 30 42 - 30 43 30 44 30 45 30 46  08090A0B 0C0D0E0F
    000020  31 30 31 31 31 32 31 33 - 31 34 31 35 31 36 31 37  10111213 14151617
    000030  31 38 31 39 31 41 31 42 - 31 43 31 44 31 45 31 46  18191A1B 1C1D1E1F
    000040  32 30 32 31 32 32 32 33 - 32 34 32 35 32 36 32 37  20212223 24252627
    000050  32 38 32 39 32 41 32 42 - 32 43 32 44 32 45 32 46  28292A2B 2C2D2E2F
    000060  33 30 33 31 33 32 33 33 - 33 34 33 35 33 36 33 37  30313233 34353637
    000070  33 38 33 39 33 41 33 42 - 33 43 33 44 33 45 33 46  38393A3B 3C3D3E3F
    000080  1A

(10) C>type test.asc
    000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F202122232425
    262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F

(11) C>tail-cut test.asc
```

```
C>dir test.*

(12) TEST      ASC        128
     TEST      BIN         64

C>dump test.asc

000000  30 30 30 31 30 32 30 33 - 30 34 30 35 30 36 30 37  00010203 04050607
000010  30 38 30 39 30 41 30 42 - 30 43 30 44 30 45 30 46  08090A0B 0C0D0E0F
(13) 000020  31 30 31 31 31 32 31 33 - 31 34 31 35 31 36 31 37  10111213 14151617
000030  31 38 31 39 31 41 31 42 - 31 43 31 44 31 45 31 46  18191A1B 1C1D1E1F
000040  32 30 32 31 32 32 32 33 - 32 34 32 35 32 36 32 37  20212223 24252627
000050  32 38 32 39 32 41 32 42 - 32 43 32 44 32 45 32 46  28292A2B 2C2D2E2F
000060  33 30 33 31 33 32 33 33 - 33 34 33 35 33 36 33 37  30313233 34353637
000070  33 38 33 39 33 41 33 42 - 33 43 33 44 33 45 33 46  38393A3B 3C3D3E3F

C>asc2bin test.asc test.new

(14)  <<< ASC to BIN >>>  Data Conversion Program  :  Produced by Y.Nagashima

                          ----- Conversion is completely finished ...

C>dir test.*

TEST      ASC        128
(15) TEST      BIN         64
TEST      NEW         64

C>dump test.new

000000  00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  ........ ........
(16) 000010  10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F  ........ ........
000020  20 21 22 23 24 25 26 27 - 28 29 2A 2B 2C 2D 2E 2F   !"#$%&' ()*+,-./
000030  30 31 32 33 34 35 36 37 - 38 39 3A 3B 3C 3D 3E 3F  01234567 89:;<=>?

C>comp test.bin test.new

C:TEST.BIN と C:TEST.NEW
(17) ファイルの終りが見つかりませんでした
ファイルの内容は一致しています
別のファイルを比較しますか＜Y／N＞？ n

(18) C>tail-add test.bin

(19) C>tail-add test.new

C>comp test.bin test.new

C:TEST.BIN と C:TEST.NEW
(20) ファイルの内容は一致しています
別のファイルを比較しますか＜Y／N＞？ n

C>dump test.new

000000  00 01 02 03 04 05 06 07 - 08 09 0A 0B 0C 0D 0E 0F  ........ ........
(21) 000010  10 11 12 13 14 15 16 17 - 18 19 1A 1B 1C 1D 1E 1F  ........ ........
000020  20 21 22 23 24 25 26 27 - 28 29 2A 2B 2C 2D 2E 2F   !"#$%&' ()*+,-./
000030  30 31 32 33 34 35 36 37 - 38 39 3A 3B 3C 3D 3E 3F  01234567 89:;<=>?
000040  1A

C>dir test.*

TEST      ASC        128
(22) TEST      BIN         65
TEST      NEW         65

C>
```

なっているというわけです.

●各ツールの内容は

Cプログラム専門の人から見るとお粗末かもしれませんが,筆者のCソース・リストは**リスト2.11**のようなものになります.1本目のbin2ascと,2本目のasc2binとはペアのもので,ほとんどそっくりにできています.ASCII文字コードを変換する方法としては,

① 変換専用のライブラリ関数を使う
② 算術的に変換の計算をする
③ 変換テーブルを参照する

などが考えられますが,ここではもっとも単純に③のテーブル変換を使用しています.また,3本目のtail-addと4本目のtail-cutも,みごとにペアとなって書かれています.tail-addにおいては,どんなファイルにも無条件に1バイトの(1Ah)という値を加えます.

〔リスト2.11〕Cツールのソース・リスト①

```
>type bin2asc.c ―――――――――――――――――― ツール(1) bin2asc ――――――――――――――――――
#include <stdio.h>

main(argc,argv)
        int argc;  char *argv[];
{
        char c,src[32],dst[32];
        int h,l,i,s_fd,d_fd;
        puts("¥n  <<< BIN to ASC >>>  Data Conversion Program");
        puts("   :  Produced by Y.Nagashima");
        if(argc==1){
                puts("¥n¥n¥tSource [Binary] File Name = ");
                gets(src);                      /* ソース ファイル ヲ シテイ シナカッタ ﾊﾞｱｲ */
        }
        else{
                strcpy(src,argv[1]);            /* アタエラレタ ファイル ネーム */
        }
        s_fd=open(src,0);                       /* ソノ ファイル ヲ イッタン サガｼﾃ ミル */
        if(s_fd==-1){
                puts("¥n¥n¥t¥tSource File does not exist ...¥n");
                exit();                         /* ナカッタラ ｼｭｳﾘｮｳ ! */
        }
        if(argc!=3){
                puts("¥n¥n¥tDestination [ASCII] File Name = ");
                gets(dst);                      /* ﾀｰｹﾞｯﾄ ファイル ヲ シテイ シナカッタ ﾊﾞｱｲ */
        }
        else{
                strcpy(dst,argv[2]);            /* アタエラレタ ファイル ネーム */
        }
        d_fd=creat(dst);                        /* ﾑｼﾞｮｳｹﾝ ﾆ ｵｰﾌﾟﾝ */
        while(1){                               /* ﾃﾞｰﾀ ﾍﾝｶﾝ ﾉ ﾙｰﾌﾟ */
                i=getc(s_fd);                   /* 1ﾊﾞｲﾄ ﾄﾘｺﾑ */
                if(i==-1){break;}               /* ファイル ｶﾞ ｵﾜｯﾀ ? */
                c=i/16;                         /* ｼﾞｮｳｲ 4ﾋﾞｯﾄ ヲ */
                putc(trans(c),d_fd);            /* ﾓｼﾞ ﾆ ﾍﾝｶﾝ ｼﾃ ファイル ﾆ ｶｸ */
                c=i%16;                         /* ｶｲ 4ﾋﾞｯﾄ ヲ */
                putc(trans(c),d_fd);            /* ﾓｼﾞ ﾆ ﾍﾝｶﾝ ｼﾃ ファイル ﾆ ｶｸ */
        }
        close(s_fd);                            /* ファイル ヲ ﾄｼﾞﾃ */
        close(d_fd);                            /* ﾌﾟｼﾞ ｵｼﾏｲ ! */
        puts("¥n¥n¥t¥t¥t ----- Conversion is completely finished ...¥n");
}

trans(p1)
        char p1;
{
        switch(p1){                             /* 4ﾋﾞｯﾄ ﾃﾞｰﾀ ヲ ﾓｼﾞ ﾆ ﾍﾝｶﾝ ｽﾙ */
                case 0:  return('0');
                case 1:  return('1');
                case 2:  return('2');
                case 3:  return('3');
                case 4:  return('4');
                case 5:  return('5');
                case 6:  return('6');
                case 7:  return('7');
                case 8:  return('8');
```

対象のファイルの大きさがわからないので，リスト上にデータのバッファを置かず，ダミー・ファイルを設けて右から左へとたれ流しにしています．ドライブの指定を省略してカレント・ドライブとしていますが，固定的なシステムであれば，RAMディスクやハード・ディスクを指定すると高速になります．1バイトをあとで加えるのは簡単ですが，tail-cutのほうの，1バイトをけずるところが，唯一考えさせられたところでしょうか．これはリストのように，ちょっとした置き換えで解決しています．

●ファイル選択用サブプログラム

Cプログラムのもう一つのサンプルは，**リスト2.12** (p.180)のようなツールで，大きなプログラム中から呼ぶために作った，ファイル選択用プログラムです．冒頭のドキュメント・ファイルの内容のとおり，DOS上で

```
                case 9:  return('9');
                case 10: return('A');
                case 11: return('B');
                case 12: return('C');
                case 13: return('D');
                case 14: return('E');
                case 15: return('F');
        }
}
```

>type asc2bin.c ―――――――――――――――― ツール(2) asc2bin ――――――――――――――――

```
#include <stdio.h>

main(argc,argv)
        int argc;  char *argv[];
{
        char c,src[32],dst[32];
        int h,l,i,s_fd,d_fd;
        puts("¥n  <<< ASC to BIN >>>  Data Conversion Program");
        puts("   :  Produced by Y.Nagashima");
        if(argc==1){
                puts("¥n¥n¥tSource [ASCII] File Name = ");
                gets(src);                      /* ソース ファイル ヲ シテイ シナカッタ ﾊﾞｱｲ */
        }
        else{
                strcpy(src,argv[1]);            /* アタエラレタ ファイル ネーム */
        }
        s_fd=open(src,0);                       /* ソノ ファイル ヲ イッタン サガｼﾃ ミル */
        if(s_fd==-1){
                puts("¥n¥n¥t¥tSource File does not exist ...¥n");
                exit();                         /* ナカッタラ ｼｭｳﾘｮｳ ! */
        }
        if(argc!=3){
                puts("¥n¥n¥tDestination [Binary] File Name = ");
                gets(dst);                      /* ﾀｰｹﾞｯﾄ ﾌｧｲﾙ ヲ ｼﾃｲ ｼﾅｶｯﾀ ﾊﾞｱｲ */
        }
        else{
                strcpy(dst,argv[2]);            /* アタエラレタ ファイル ネーム */
        }
        d_fd=creat(dst);                        /* ﾑｼﾞｮｳｹﾝ ﾆ ｵｰﾌﾟﾝ */
        while(1){                               /* ﾃﾞｰﾀ ﾍﾝｶﾝ ﾉ ﾙｰﾌﾟ */
                h=getc(s_fd);                   /* ｼﾞｮｳｲ 1ﾊﾞｲﾄ ﾄﾘｺﾑ */
                if(h==-1){break;}               /* ファイル ｶﾞ ｵﾜｯﾀ ? */
                l=getc(s_fd);                   /* ｶｲ 1ﾊﾞｲﾄ ﾄﾘｺﾑ */
                if(l==-1){break;}               /* ファイル ｶﾞ ｵﾜｯﾀ ? */
                c=16*trans(h)+trans(l);         /* ｱﾘｾﾃ 1ﾊﾞｲﾄ ﾃﾞｰﾀ ﾄ ｽﾙ */
                putc(c,d_fd);                   /* ｺﾚﾖ ﾌｧｲﾙ ﾆ ｶｸ */
        }
        close(s_fd);                            /* ファイル ヲ ﾄｼﾞﾃ */
        close(d_fd);                            /* ﾌﾞｼﾞ ｵｼﾏｲ ! */
        puts("¥n¥n¥t¥t¥t ----- Conversion is completely finished ...¥n");
}

trans(p1)
        char p1;
{
        switch(p1){                             /* ﾓｼﾞ ﾃﾞｰﾀ ヲ 4ﾋﾞｯﾄ ﾃﾞｰﾀ ﾆ ﾍﾝｶﾝ ｽﾙ */
                case '0':  return(0);
                case '1':  return(1);
                case '2':  return(2);
```

手入力でDIRをとって(ワイルド・カード含む)，そこからファイル名を選ぶことに相当する作業をメインのソフトを走らせるなかで実行してしまうためのものです．

動作のポイントの第1は，このようなメインとのインターフェース部分で，〈選択すべきファイルの条件〉と，〈選択されたファイル名を格納すべきファイルの指定〉とを与えることで，このツール自体はどのようなメインからも使えるようにしてあります．メイン側では，

```
exec("file-sel", "a : *.asc  d : work.dum");
```
（↑このツール名　↑選択候補　↑選択されたファイル名を入れるファイル）

などと呼んでやれば，このツールによってファイルが選択されるというものです．

第2のポイントとしては，ディレクトリ一覧をつくるのに，いちいちファイル検索を繰り返さずに，子プロセスとしてcommand.comを起動して，そこでDIRの出力をバッファ・ファイルにパイプ出力し，つぎに

〔リスト2.11〕Cツールのソース・リスト②

```
                case '3':  return(3);
                case '4':  return(4);
                case '5':  return(5);
                case '6':  return(6);
                case '7':  return(7);
                case '8':  return(8);
                case '9':  return(9);
                case 'A':  return(10);
                case 'B':  return(11);
                case 'C':  return(12);
                case 'D':  return(13);
                case 'E':  return(14);
                case 'F':  return(15);
                case 'a':  return(10);
                case 'b':  return(11);
                case 'c':  return(12);
                case 'd':  return(13);
                case 'e':  return(14);
                case 'f':  return(15);
        }
}
```

```
>type tail-add.c ―――――――――――――――――――― ツール(3) tail-add ――――――――――――――――――――
#include <stdio.h>

main(argc,argv)
        int argc;  char *argv[];
{
        char c,src[32],dst[32];
        int i,s_fd,d_fd;
        if(argc==1){
                puts("¥n¥tSource File Name = ");
                gets(src);                      /* ソース ファイル ヲ シテイ シナカッタ ﾊﾞｱｲ */
        }
        else{
                strcpy(src,argv[1]);            /* アタエラレタ ファイル ネーム */
        }
        s_fd=open(src,0);                       /* ソノ ファイル ヲ イッタン サカﾞシテ ミル */
        if(s_fd==-1){
                puts("¥n¥n¥t¥tSource File does not exist ...¥n");
                exit();                         /* ナカッタラ シュウリョウ ! */
        }
        strcpy(dst,"@@@.@@@");
        d_fd=creat(dst);                        /* サキﾞョウ ヨウ ﾀﾞﾐｰ ファイル ヲ サクセイ */
        while(1){                               /* ﾃﾞｰﾀ ショリ ノ ﾙｰﾌﾟ */
                i=getc(s_fd);                   /* 1ﾊﾞｲﾄ トリコム */
                if(i==-1){break;}               /* ファイル ｶﾞ オワッタ ? */
                c=(char)i;                      /* コレヲ モウ 1 カイ */
                putc(c,d_fd);                   /* ファイル ニ カク */
        }
        putc(0x1a,d_fd);                        /* サイゴﾞ ニ [1A] ヲ クワエル ! */
        close(s_fd);                            /* ユコテﾞ イッタン */
        close(d_fd);                            /* ファイル ヲ トジﾞテ */
        s_fd=open(dst,0);                       /* テンソウノ ムキ ヲ カエテ */
        d_fd=open(src,1);                       /* フタタビﾞ オープﾟン */
        while(1){                               /* ﾃﾞｰﾀ ショリ ノ ﾙｰﾌﾟ */
                i=getc(s_fd);                   /* 1ﾊﾞｲﾄ トリコム */
```

このバッファ・ファイルを文字列の集合として扱う，というところです．DOSによってはDIRリストのフォーマットが多少異なるかもしれませんが，一部の修正で汎用性があると思います．

そして第3のポイントは，画面への出力をMS-DOSの標準エスケープ・シーケンスを使用したところで，アトリビュートの部分を変更するだけで，たいがいのMS-DOSパソコンで使えるソフトとなっています．機種ごとの対応は，上下左右の矢印キーに割り当ててあるコードを指定するところの変更ぐらいのものでしょう．

このルーチンを骨格とした，同様のツールはいくつもあり，このようにEXEファイル単位で共通処理を外に出すことで，大きなシステムのソフトの見通しもよくなるようです．このようなDOSと密着したソフト技術は，まさにC言語のメリットをいかした発想で

```
                if(i==-1){break;}       /* ﾌｧｲﾙ ｶﾞ ｵﾜｯﾀ ? */
                c=(char)i;              /* ｺﾚﾖ ﾓｳ 1 ｶｲ */
                putc(c,d_fd);           /* ﾌｧｲﾙ ﾆ ｶｸ */
        }
        close(s_fd);                    /* ﾌｧｲﾙ ﾖ ﾄｼﾞﾃ */
        close(d_fd);                    /* ﾌﾞｼﾞ ｵｼﾏｲ ! */
        unlink(dst);                    /* ﾀﾂ ﾄﾘ ｱﾄ ﾖ ﾆｺﾞｻｽﾞ.. */
}
```

>type tail-cut.c ——— ツール(4) tail-cut ———

```
#include <stdio.h>

main(argc,argv)
        int argc;  char *argv[];
{
        char c,src[32],dst[32];
        int i,j,k,s_fd,d_fd;
        if(argc==1){
                puts("¥n¥tSource File Name = ");
                gets(src);              /* ｿｰｽ ﾌｧｲﾙ ﾖ ｼﾃｲ ｼﾅｶｯﾀ ﾊﾞｱｲ */
        }
        else{
                strcpy(src,argv[1]);    /* ｱﾀｴﾗﾚﾀ ﾌｧｲﾙ ﾈｰﾑ */
        }
        s_fd=open(src,0);               /* ｿﾉ ﾌｧｲﾙ ﾖ ｲｯﾀﾝ ｻｶﾞｼﾃ ﾐﾙ */
        if(s_fd==-1){
                puts("¥n¥n¥t¥tSource File does not exist ...¥n");
                exit();                 /* ﾅｶｯﾀﾗ ｼｭｳﾘｮｳ ! */
        }
        strcpy(dst,"@@@.@@@");
        d_fd=creat(dst);                /* ｻｷﾞｮｳ ﾖｳ ﾀﾞﾐｰ ﾌｧｲﾙ ﾖ ｻｸｾｲ */
        while(1){                       /* ﾃﾞｰﾀ ｼｮﾘ ﾉ ﾙｰﾌﾟ */
                i=getc(s_fd);           /* 1ﾊﾞｲﾄ ﾄﾘｺﾑ */
                if(i==-1){break;}       /* ﾌｧｲﾙ ｶﾞ ｵﾜｯﾀ ? */
                c=(char)i;              /* ｺﾚﾖ ﾓｳ 1 ｶｲ */
                putc(c,d_fd);           /* ﾌｧｲﾙ ﾆ ｶｸ */
        }
        close(s_fd);                    /* ｺｺﾃﾞ ｲｯﾀﾝ */
        close(d_fd);                    /* ﾌｧｲﾙ ﾖ ﾄｼﾞﾃ */
        unlink(src);                    /* ｺﾜｲｹﾄﾞ... ｿｰｽ ﾖ ｹｼﾃ */
        s_fd=open(dst,0);               /* ﾃﾝｿｳﾉ ﾑｷ ﾖ ｶｴﾃ */
        d_fd=creat(src,1);              /* ﾌﾀﾀﾋﾞ ｵｰﾌﾟﾝ */
        i=getc(s_fd);                   /* ﾏｽﾞ ｻｷﾆ 1ﾊﾞｲﾄ ﾄﾘｺﾝﾃﾞ ｵｸ ! */
        while(1){                       /* ﾃﾞｰﾀ ｼｮﾘ ﾉ ﾙｰﾌﾟ */
                j=getc(s_fd);           /* ｻﾗﾆ 1ﾊﾞｲﾄ ﾄﾘｺﾒﾙｶﾅ ? */
                if(j==-1){break;}       /* ﾅｶｯﾀﾗ ... ｵｼﾏｲ ! */
                c=(char)i;              /* ｱｯﾀﾗ ... ｻｯｷ ﾉ ﾃﾞｰﾀ ﾖ */
                putc(c,d_fd);           /* ﾌｧｲﾙ ﾆ ｶｲﾃ */
                i=j;                    /* ｲﾏ ﾉ ﾃﾞｰﾀ ﾖ ｻｷﾞ ﾆ ﾂｶｳ ! */
        }
        close(s_fd);                    /* ﾌｧｲﾙ ﾖ ﾄｼﾞﾃ */
        close(d_fd);                    /* ﾌﾞｼﾞ ｵｼﾏｲ ! */
        unlink(dst);                    /* ﾀﾂ ﾄﾘ ｱﾄ ﾖ ﾆｺﾞｻｽﾞ.. */
}
```

〔リスト2.12〕ファイル選択サブプログラムのリスト

```
>type file-sel.doc

   [FILE-SEL.EXE] : File Select Menu Sub-Program

       argv<1> = "*:*****.***" <-- Objects (<*.*> is OK)

       argv<2> = "*:*****.***" <-- Target File Name

            !!! Using RAM-DISK = [D:] !!!

               (ex) If argv<1>="a:*.src", argv<2>="d:dummy.buf", so
                    You may use this as follows :
                            " file-sel a:*.src d:dummy.buf " !!
                    Then [a:*.src] are displayed, and you can select.
                    Selected File Name is written to [d:dummy.buf].
```

機能説明ドキュメント・ファイル

```
>type file-sel.c

typedef  int  FILE;                              /* [stdio.h] */
#define  stdin  0                                /*       |   */
#define  stdout 1                                /*       |   */
#define  stderr 2                                /*       |   */
#define  NULL   0                                /*       |   */
#define  TRUE   1                                /*       |   */
#define  FALSE  0                                /*       |   */
#define  EOF    (-1)                             /*       |   */
#define  ERR    (-1)                             /* [stdio.h] */
#define  ESC    0x1b
#define  CR     0x0a0d
#define  UP     1111                             /* ｺｺﾉ ｽｳｼﾞﾊ */
#define  DOWN   2222                             /* ｲｸﾂ ﾃﾞﾓ   */
#define  RIGHT  3333                             /* ｲｲﾝﾃﾞｽ... */
#define  LEFT   4444                             /* ﾃｷﾄｰﾆ     */
#define  RET    5555                             /* ｴﾗﾍﾞﾏｽ... */

char   file_name[64],name_buff[80][10],message[100];
char   dst_file_name[16],target[32];
int    fdes,f_count,disp_x,disp_y,key_data,point,old_point;

main(argc,argv)
        int argc;  char *argv[];
{
        strcpy(target,argv[1]);                  /* ｻｶﾞｽ ﾌｧｲﾙ ﾉ ｼﾃｲ */
        strcpy(dst_file_name,argv[2]);           /* ｴﾗﾍﾞﾚﾇ ﾌｧｲﾙ ﾒｲ ﾉ ｲｷｻｷ */
        strcpy(file_name,"/c dir ");
        strcat(file_name,argv[1]);
        strcat(file_name," > d:dirbuff.zzz");
        exec("command.com",file_name);           /* [DIR] ｦ ｼﾞｯｺｳ ｽﾙ */
        dir_setting();                           /*  DIR ﾘｽﾄ ﾋｮｳｼﾞ */
        data_file_select();                      /* ｾﾝﾀｸ ｼﾃ ﾄｳﾛｸ ｽﾙ */
}

dir_setting(){
        int i;
        char char_buff[10];
        point=old_point=f_count=0;
        strcpy(file_name,"d:dirbuff.zzz");       /* ﾃﾞｨﾚｸﾄﾘ･ﾘｽﾄ ﾉ ﾌｧｲﾙ ｦ */
        fdes=fopen(file_name,"r");               /* ﾏｽﾞ ｵｰﾌﾟﾝ ｽﾙ */
        if(fdes==0) return(0);                   /* (ｶﾅﾗｽﾞ ｱﾙ ﾊｽﾞ...) */
        while(1){
                if(fgets(message,100,fdes)==0) break;
                if(isalpha(message[0])==1){      /* ｼﾞｯｻｲﾆ ﾃﾞｰﾀ ﾉ ｱﾙ ｷﾞｮｳ */
                        for(i=0;i<8;i++) char_buff[i]=message[i];
                        char_buff[8]='¥0';       /* ﾌｧｲﾙ ﾒｲ ｦ ﾄﾘｺﾑ */
                        strcpy(name_buff[f_count++],char_buff);
                }
                if(f_count>77) break;            /* ｻｲﾀﾞｲ 77 ﾌｧｲﾙ */
        }
        close(fdes);
        unlink(file_name);                       /* ﾋｸﾖｳﾅｲ ﾊﾞｯﾌｧ ﾆ ｷｴﾃ ﾓﾗｳ */
        cursor_off();                            /* ﾕｺﾃﾞ ｶｰｿﾙ ｦ ｹｼﾃ */
        list_display();                          /* ﾌｧｲﾙ ｲﾁﾗﾝ ｦ ﾋｮｳｼﾞ */
}

list_display(){
        int i;
        screen_clear();                          /* ﾏｽﾞ ｶﾞﾒﾝ ｦ ｸﾘｱ ｼﾃ */
        for(i=0;i<f_count;i++){                  /* ﾌｧｲﾙ ﾉ ｶｽﾞ ﾀﾞｹ ﾙｰﾌﾟ */
                if(i==0){                        /* ｻｲｼｮ ﾊ 1ﾍﾞﾝﾒ ﾆ ﾏｰｸ */
                        cursor_move(11*(i%7)+1,2*(i/7)+2);
                        cursor_reverse();        /* ﾏｰｸ ｦ ﾂｹﾙ ! */
                        putchar(' ');
```

```
                        puts(name_buff[i]);     /* ﾌｧｲﾙ ﾒｲ ﾋｮｳｼﾞ */
                        putchar(' ');
                        cursor_normal();        /* ﾏｰｸ ﾊ ｺｺﾏﾃﾞ */
                }
                else{                           /* ｿﾚ ｲｶﾞｲ ﾉ ﾌｧｲﾙ */
                        cursor_move(11*(i%7)+2,2*(i/7)+2);
                        puts(name_buff[i]);     /* ﾌｧｲﾙ ﾒｲ ﾋｮｳｼﾞ */
                }
        }
        cursor_move(2,2);                       /* ﾒｲﾝ ﾆ ﾓﾄﾞﾘﾏｽ... */
}

data_file_select(){
        int i;
        char char_buff[32];
        while(1){
                key_input_check();              /* ｷｰ ﾖﾙｼﾞﾙｼ ｦ ﾓﾆﾀ ｼﾃ... */
                if(key_data==RET){
                        break;                  /* [RET]ｷｰ ﾅﾗ ｲﾖｲﾖ ｶｷｺﾐ !*/
                }
                else if(key_data!=0){           /* ｿｳﾃﾞ ﾅｹﾚﾊﾞ */
                        if(mark_move()==1) mark_display();   /* ﾏｰｸ ｦ ｲﾄﾞｳ */
                }
        }
        target[2]=0;
        strcat(target,name_buff[point]);
        for(i=0;i<10;i++){
                char_buff[i]=target[i];
                if(char_buff[i]==' ') char_buff[i]=0;
        }
        char_buff[10]='¥0';
        fdes=fopen(dst_file_name,"w"); /* ﾀｰｹﾞｯﾄ ﾌｧｲﾙ ﾆ */
        fputs(char_buff,fdes);  /* ｴﾗﾊﾞﾚﾀ ﾌｧｲﾙ ﾈｰﾑ ｦ ｶｲﾃ */
        putw(CR,fdes);
        close(fdes);   /* ｺﾚﾃﾞ OK ! */
        cursor_on();   /* ｶｰｿﾙ ｦ ﾓﾄﾞｼﾃ ｵｸ */
}

mark_move(){            /* ｶｰｿﾙ ﾏｰｸ ﾉ ｲﾄﾞｳ ﾊﾝﾃｲ */
        int pp;
        pp=point;
        switch(key_data){
         case UP:               /* ｳｴ */
                 if(pp>6) pp=pp-7;/* ｻｲｼﾞｮｳﾚｳ ﾅﾗﾊﾞ*/
                 else return(0);            /* ｿﾉﾏﾏ */
                 break;
         case DOWN:     /* ｼﾀ */
                 if(pp+7>=f_count) return(0);
                 else pp=pp+7;  /* ｻｲｶﾙﾚｳ ﾅﾗﾊﾞ*/
                 break;                     /* ｿﾉﾏﾏ */
         case RIGHT:        /* ﾐｷﾞ */
                 if((pp%7)==6) pp=pp-6;
                 else pp++;  /* ｲｹﾅｹﾚﾊﾞ ｿﾉﾏﾏ */
                 if(pp>=f_count) return(0);
                 break;
         case LEFT:         /* ﾋﾀﾞﾘ */
                 if((pp%7)==0) pp=pp+6;
                 else pp--;  /* ｲｹﾅｹﾚﾊﾞ ｿﾉﾏﾏ */
                 if(pp>=f_count) return(0);
                 break;
        }
        point=pp;
        return(1); /* ｾｲｼﾞｮｳ ｼｭｳﾘｮｳ ｺｰﾄﾞ */
}

mark_display(){     /* ﾏｰｸ ﾉ ｲﾄﾞｳ ｦ ﾋｮｳｼﾞ */
        int i,j;
        i=point;
        j=old_point;
        cursor_move(11*(j%7)+1,2*(j/7)+2);
        putchar(' ');
        puts(name_buff[j]);/* ﾏｰｸ ｦ ｹｼﾃ ﾌﾂｳﾆ ｶｳ */
        putchar(' ');
        cursor_move(11*(i%7)+1,2*(i/7)+2);
        cursor_reverse();   /* ﾏｰｸ ｽﾀｰﾄ */
        putchar(' ');
        puts(name_buff[i]); /* ﾌｧｲﾙ ﾈｰﾑ */
        putchar(' ');
        cursor_normal();    /* ﾏｰｸ ｦ ﾓﾄﾞｽ */
        cursor_move(11*(i%7)+2,2*(i/7)+2);
        old_point=i;
}
```

```
key_input_check(){   /* ｷｰ ｼﾞｮｳﾀｲ ﾉ ﾓﾆﾀｰ */
        char c;
        key_data=0;
        c=ci();
        if(c==0x0d) key_data=RET; /* [CR] ? */
        switch(c){  /* ｺｺｦ ｶｴﾙﾄ ﾍﾞﾂﾉ ﾊﾟｿｺﾝﾓ OK ! */
                case 0x48: key_data=UP;     break;
                case 0x50: key_data=DOWN;   break;
                case 0x4d: key_data=RIGHT; break;
                case 0x4b: key_data=LEFT;   break;
        }
}
cursor_reverse(){
        printf("%c[7m",ESC); /* Attrib = 7 */
}

cursor_normal(){
        printf("%c[0m",ESC); /* Attrib = 0 */
}

cursor_on(){
        printf("%c[>5l",ESC);/* Cursor Display */
}

cursor_off(){
        printf("%c[>5h",ESC);/* Cursor Not*/
}                                       /*Display */

cursor_move(p1,p2)
        int p1,p2;
{
        printf("%c[%d;%dH",ESC,p2,p1);/* Move to*/
}                                               /*<y,x> */

screen_clear(){
        printf("%c[2J",ESC);    /* Screen Clear */
}
```

あり，いい意味でのアマチュア精神の活躍の場となるのです．

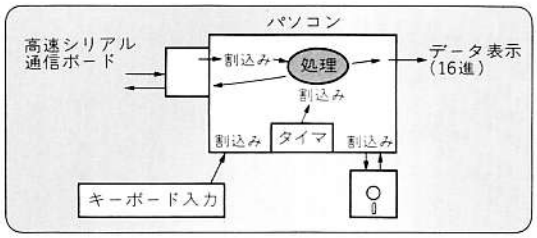
割込みとアセンブラ

さて，C言語での開発に限界がでてくるのは，おもに割込み処理のスピードによるものです．たとえば，かなり昔のことになりますが，筆者の経験した，つぎのようなシステムについて考えてみましょう（**リスト2.13**）．

例2

数10 kbps程度のシリアル通信ポートをもち，通信データを刻々と画面にモニタしながら，キーボードに割り当てられたいくつかのコマンドに対応したディスク・アクセス，ポート入出力処理を行う．データのプロトコル上，取りこぼしは許されない．

〔例2〕割込み処理が複数あるパソコン応用システムの例



このようなシステムでは，割込みが多重に発生し，Cではかなりきつい条件となってきます．とくにこのとき使ったパソコンは今では幻のパソコンとなってしまった変り種のもので，キーボードの情報を検出する方法

〔リスト2.13〕割込み処理をアセンブラで記述したCプログラムの例①

```
>type main.c  《Cのメイン・プログラム》

#include <STDIO.H>

        extern  int     INIT();     /* subr */
        extern  int     MONITOR();  /* subr */
        extern  int     FINISH();   /* subr */
        extern  int     KEY_SCAN(); /* bios */
        extern  int     PLOT();     /* disp */
        extern  int     WRITE();    /* disp */
        extern  int     DISC();     /* file */

        extern  int     XDATA,YDATA;/* common*/
        extern  int     i,j,scan;  /*parameter */

main(){
        initial();    /* system initialize */
        for(;;){
                key_check();
                data_display();
        }
}

initial(){
        INIT();         /* assembler initial */
        for(i=1;i<9;i++){
                XDATA=5;
                YDATA=11*i+12;
                j=0x60+i;
                PLOT(j); /* panel graphics */
        }                /* (省略) */
        for(i=0;i<160;i++){
                XDATA=i;
                WRITE(0);/* display initial set */
        }
        DISC(0);         /* file system initial */
}

key_check(){
        scan++;
        if(scan<100){
                return; /* scan wait */
        }
        scan=0;
        i=KEY_SCAN();
        if(i==0){               /* non event */
                return;
        }
        else if(i==0x1e){ /* A */
                FINISH(); /* ﾌﾟﾛｸﾞﾗﾑ終了 */
        }
        else if(i==0x30){       /* B */
                disc_service(); /* (省略) */
        }
}
data_display(){
        XDATA=MONITOR();
        if(XDATA==0xffff){
                return;         /* non data */
        }
        else{
                WRITE(1);       /* data display */
        }                       /* (省略) */
}
```

```
>type subr.a  《アセンブラのサブルーチン①》

        cgroup  group   code
        dgroup  group   data,xdata,ydata

        data    dseg    para
                dispff  rw 1000
                top_f   rw 1
                pot_f   rw 1
        xdata   dseg    para    common
                x_data  rw 1
        ydata   dseg    para    common
                y_data  rw 1

        code    cseg    public
                public  init            ; main
                public  monitor         ; main
                public  finish          ; main
                extrn   get_data:near   ; bios
                extrn   enable_nmi:near ; bios
                extrn   disable_nmi:near ; bios
                extrn   set_flag:near   ; bios
                extrn   reset_flag:near ; bios
                extrn   dos_return:near ; bios
                extrn   mode_set:near   ; disp

        irct    equ     020h
        irmask  equ     00000100b
```

が特殊性だったり，割込みコントローラを直接制御して各種割込みの優先度を変更する必要があったりしました．

さらに，当時はこのパソコン用のC言語じたいが少なくて，あるCP/M版のCを制限付きでMS-DOS版にアレンジした，というCコンパイラを使用したのですが，制限どころかバグまであるという，相当な悪条件下での開発となりました．まずは定石どおり，割込みのベクタ・アドレスを設定するDOSファンクション・コールを使い，シリアルの割込み処理ルーチンをCとアセンブラで，ディスク関係をCの関数呼出しで，キーボード関係をBIOSコールで処理し，これらのルーチンをC中のメイン・ルーチンから呼ぶ，という構造にしました．また，ハードの制約からディスク処理とシリアル割込みの共存は不可能であるのを確認して，メイン処理を排他的なメニュー構造にしました．その上で，ある程度のプログラムができた段階でデバッグしてみると，シリアル割込みがそこそこの頻度で起こる場合まで問題ないように見えて，シリアル・データの密度を上げながらキーボードのイベントを同時に発生させると，ごくまれに誤動作し，ときにはデッドロックしてしまう，という現象に悩まされました．

割込み処理のアセンブラ・ルーチンから呼ばれるシリアル・データを解釈するCのルーチンがボトルネックのような気がして，デバッガで解析してみるうちに，Cコンパイラで変換された機械語プログラムを見て，その予想外の姿に驚いてしまいました．あまりに冗長なのです．標準化のためなのでしょうが，機械語とはいっても，DOSのシステム・コールや安全確実な処理アルゴリズムに変換したり，汎用性のために過度に厳


```
init: ;-----------------------------------------
        push    ds
        mov     ax,cs
        mov     ds,ax
        mov     ax,cs:offset hard_int
        mov     dx,ax
        mov     ax,250ah
        int     21h       ; vector address set
        pop     ds
        mov     al,09h
        call    mode_set ; display mode set
        mov     al,irmask
        call    set_flag ; hard int. flag enable
        sti
        ret

hard_int: ;-------------------------------------
        push    ds
        push    ss
        push    es
        push    bp
        push    di
        push    si
        push    dx
        push    cx
        push    bx
        push    ax
        call    disable_nmi
        call    get_data
        call    int_sequence
        mov     al,00100000b
        out     irct,al
        call    enable_nmi
        pop     ax
        pop     bx
        pop     cx
        pop     dx
        pop     si
        pop     di
        pop     bp
        pop     es
        pop     ss
        pop     ds
        sti
        iret

monitor: ;--------------------------------------
        mov     ax,top_f
        cmp     pot_f,ax
        jnz     mon001
        mov     ax,0ffffh     ; buffer empty
        ret
mon001: mov     bx,ds:offset dispff
        mov     si,pot_f
        add     si,bx
        mov     al,ds:[si]    ; data get from fifo
        inc     pot_f
        cmp     pot_f,1000
        jnz     mon002
        mov     pot_f,0
mon002: mov     ah,0
        ret

int_sequence: ;---------------------------------
        mov     bx,ds:offset dispff
        mov     di,top_f
        add     di,bx
        mov     ds:[di],al      ; data set to fifo
        inc     top_f
        cmp     top_f,1000
        jnz     intend
        mov     top_f,0
intend: ret
finish: ;---------------------------------------
        push    ax
        mov     al,irmask
        call    reset_flag
        pop     ax
        mov     al,3
        call    mode_set   ; display mode recover
        call    dos_return ; quit
        ret

        end


>type bios.a   《アセンブラのサブルーチン②》

        cgroup  group   code
        dgroup  group   data,xdata,ydata

        data    dseg    para
                flag    rb 1
        xdata   dseg    para    common
                x_data  rw 1
```

重なレジスタ退避を行っているために，C言語から変換されたプログラムは，直接アセンブラで書いた場合に比べてかなりの回り道をしているのです(**リスト2.14**)．

本格的にCやアセンブラをやるのは初めてだったので，この機会に徹底的にやってみようと決意しました．デバッガによってあちこちを詳しく解析し，冗長なDOSコール，BIOS*コールをつぎつぎに改良して自前のアセンブラとして組み，Cルーチンと置き換えて比較実験を繰り返してみました．他と競合するディスプレイBIOSを解析して画面処理をほとんど自前のものに変更し，キーボードBIOSを書き換え，割込み順位を変えて実験し，最終的にはディスク処理もアセンブラにして，結局，最初はC言語だったプログラムが，全部アセンブラになってしまいました．もちろん高速処理も完全となり，それまで相手まかせであったDOS・BIOSのファンクション・コールも自分のものとなって，動作の細部にまで自信がもてました(ついでにDOSまで勉強できてしまいました．p.186 Appendix 2.1「**体験的プログラム解析のすすめ**」参照)．

これは非常に極端な例でしたが，C言語とアセンブラをどのようなバランスで使いこなすか(**コラム2.4**)，というのはプログラマの重要なセンスであると思います．筆者の場合，実験用のツールとしてはCを多く使いますが，少し本格的な仕事をさせたいときには，最初からマクロ・アセンブラで書く場合もよくあります．アセンブラはやはり高速で，デバッガに慣れればCよりも細かいデバッグが容易です(p.194 Appendix 2.2「**MASMのマクロ使用例**」参照)．MS-DOS用のマクロ・アセンブラ(MASM)の場合，とくにCPUのア

〔リスト2.13〕割込み処理をアセンブラで記述したCプログラムの例②

```
        ydata   dseg    para    common
                y_data  rw 1

        code    cseg    public
                public  key_scan
                public  get_data
                public  enable_nmi
                public  disable_nmi
                public  set_flag
                public  reset_flag
                public  dos_return

        buff_head       equ     0001ah
        buff_tail       equ     0001ch
        irct            equ     00020h
        buff_start      equ     00080h
        buff_end        equ     00082h
        nmi_port        equ     000a0h
        data_port       equ     0e0d0h
        status_port     equ     0e0d2h

key_scan: ;-------------------------------------------
        ; (This Routine is nearly equal to KBD-BIOS !)
        cli
        push    es
        mov     bx,40h
        mov     es,bx
        mov     si,buff_head
        mov     bx,es:[si]
        mov     di,buff_tail
        cmp     bx,es:[di]
        jnz     ks1
        mov     ax,0            ; non event = [00]
        jmp     ks3
ks1:    mov     ax,es:[bx]
        inc     bx
        inc     bx
        mov     si,buff_end
        cmp     bx,es:[si]
        jne     ks2
        mov     si,buff_start
        mov     bx,es:[si]
ks2:    mov     di,buff_head
        mov     es:[di],bx
        mov     al,ah           ; scan code = [al]
        mov     ah,0
ks3:    pop     es
        sti
        ret

get_data: ;-------------------------------
        mov     dx,status_port
loop1:  in      al,dx
        and     al,00000001b
        jnz     loop1
        mov     dx,data_port
        in      al,dx       ; get data = [al]
        ret

enable_nmi: ;-----------------------------
        mov     al,flag
        out     irct+1,al
        in      al,nmi_port
        mov     al,80h
        out     nmi_port,al
        ret

disable_nmi: ;----------------------------
        mov     al,10h
        out     nmi_port,al
        in      al,irct+1
        mov     flag,al
        mov     al,0
        out     irct+1,al
        ret

set_flag: ;-------------------------------
        mov     ah,al
        in      al,irct+1
        not     ah
        and     al,ah
        out     irct+1,al
        ret

reset_flag: ;-----------------------------
        mov     ah,al
        in      al,irct+1
        or      al,ah
        out     irct+1,al
        ret

dos_return: ;-----------------------------
        mov     ah,4ch
        int     21h         ; quit
        ret

        end
```

(他のいくつかのアセンブラ・サブルーチンは省略)

〔リスト2.14〕
C言語プログラム
の展開例

```
>type sample.c

#include  <stdio.h>

main(){
        puts("Hello World !");
}

>symdeb sample.exe

2AE7:0003 55            PUSH    BP
2AE7:0004 8BEC          MOV     BP,SP
2AE7:0006 B80600        MOV     AX,0006
2AE7:0009 50            PUSH    AX
2AE7:000A E82B01        CALL    0138
2AE7:000D 8BE5          MOV     SP,BP
2AE7:000F 5D            POP     BP
2AE7:0010 C3            RET

2AE7:0138 58            POP     AX
2AE7:0139 5B            POP     BX
2AE7:013A 53            PUSH    BX
2AE7:013B 50            PUSH    AX
2AE7:013C 8A07          MOV     AL,[BX]
2AE7:013E 0AC0          OR      AL,AL
2AE7:0140 740C          JZ      014E
2AE7:0142 53            PUSH    BX
2AE7:0143 50            PUSH    AX
2AE7:0144 E8DCFF        CALL    0123
2AE7:0147 83C402        ADD     SP,+02
2AE7:014A 5B            POP     BX
2AE7:014B 43            INC     BX
2AE7:014C EBEE          JMP     013C
2AE7:014E C3            RET

2AE7:0123 58            POP     AX
2AE7:0124 5A            POP     DX
2AE7:0125 52            PUSH    DX
2AE7:0126 50            PUSH    AX
2AE7:0127 80FA0A        CMP     DL,0A
2AE7:012A 7507          JNZ     0133
2AE7:012C B20D          MOV     DL,0D
2AE7:012E E80200        CALL    0133
2AE7:0131 B20A          MOV     DL,0A
2AE7:0133 B402          MOV     AH,02
2AE7:0135 CD21          INT     21
2AE7:0137 C3            RET

2AE7:01B3   48 65 6C 6C 6F 20 57 6F 72 6C 64 20 21 00  [Hello World !.]
```

Cのソース・プログラム

コンパイルされたプログラムの
逆アセンブル・リストの一部

本当に肝心な部分はこれだけ！

コラム 2.4

C言語とアセンブラのトレードオフ

どのようなバランスでC言語とアセンブラを使い分け，使いこなすのか，という記事は，あらゆる本でよく見かけます．もちろん筆者も「これしかない！」と断言できる意見は述べられません．限られた経験則でいえば，つぎのように考えることにしています．

(1) 高速な部分，リアルタイム処理はアセンブラで．
(2) コンソール対応，グラフィック・ディスプレイはCで(マン-マシン・インターフェースは細かく凝りたい，そしてスピードはそこそこ)．
(3) 仕様変更のありそうな部分はCで．
(4) アセンブラのルーチンは細分化したモジュールとしてCから呼ぶ．
(5) 割込み対応はアセンブラで．
(6) デバッグを{十分に / 細かく}行うときはアセンブラで．
(7) 開発期間が限られているときは，なるべくオールCかオール・アセンブラで(**インターフェースがバグの元**)．
(8) 時間が許せば，別々にアセンブル/コンパイルしたモジュールをリンクするのでなく，なるべく別々にエディットしたソースをアセンブル/コンパイル時にインクルードして，1本プログラムとして扱う．

などです．

ーキテクチャとC言語の変数体系が非常に似ていて，マクロ・アセンブラをうまく使って書いていると，Cのプログラムの便利さとあまり変わらない気がしてきます(この例のときに使ったCコンパイラはもう絶版になりましたが，Cというのはどこかしら不安だ，という心理がまだ残っているからかもしれません)．

C言語のレベルはともかく，DOSやBIOSのファンクション・コールをアセンブラで書こうとすれば，自然とパソコンの中心であるCPUのハードに肉薄することになります．また，**割込みやRS-232-Cやディスク I/Oなどのルーチンをアセンブラで書くためには，CPU周辺LSIの詳細を理解する必要があり**，このへんまでくると，パソコンとはいっても，マイコン・システムの技術としても相当なレベルといえるでしょう．

Appendix 2.1

体験的プログラム解析のすすめ

初めてマクロ・アセンブラ(MASM)のマニュアルを読んだときは，正直いって何もわかりませんでした．PC-9801版も多少そうですが，IBMのマニュアルなど，ユーザにプログラムの自作を断念させるための文章の見本のようです．筆者がアセンブラを身につけることができたのは，MS-DOSのデバッガ(IBM PCならdebug.com，PC-9801ならsymdeb.exe)によってプログラムを解析しているうちに，本当の理解が得られたためと思っています．

日本のパッケージ・ソフトには，コピー・プロテクトのかかっているものが非常に多いのですが，筆者は自己向上のための課題として，プログラムを解析してコピー・プロテクトを破ることを趣味の一つとしています．もちろんソフトは正規購入でユーザ登録するのですが，パソコン自身の故障によって大切なディスクを破壊された経験が10回以上はあるので，マスタ・ディスクをそのまま使う，などということは考えられないのです．世の中にはコピー・ツールとかいう，頭を使わないでプロテクトを破るソフトもあるそうですが，せっかくの楽しみを知らないでコピー・ツールを使う人の気が知れません．以下，筆者の三つの「楽しい経験」を紹介します．

第1の例は，DIRコマンドで表示され，COPYコマンドでコピーできたのでコピー・フリーのソフトなのだ，と思って実行させたら走らない，というタイプのものでした(**リストA**)．

デバッガに入ってプログラムを逆アセンブル表示してみると，冒頭で2回[8D40]のルーチンをコールしています．ブレーク・ポイントをこのあとに設定してgoさせてもそこまでいかないので，どうやら[8D40]のルーチンでマスタ・ディスクであるかどうかを判定しているようです．

そこで[8D40]以下を表示してみました．じつはこのときは，まだ〈INT **〉というようなDOS割込み・BIOS割込みというのを知らなかったので，ここでアセンブラのマニュアルとBIOSマニュアルをもちだし，じっくり調べてみました．すると，ディスク用のBIOSコール[INT 13h]の前に設定するパラメータで，トラック・ナンバ=80というのが目をひきました．MS-DOSの2DDフォーマットではトラックは0から79であり，80というのは初めてだったのです．

このチェック・ルーチンの動作をまとめると，MS-DOSフォーマットの定義外である80トラック目の第1セクタに[FFFF]が書かれていればマスタ・ディスクとして正常にもどり，プログラムに進めるというものでした．普通のMS-DOSフォーマットのディスクにコピーしても，ここが書き込まれていないと，無限ループになって実行できないのです．MS-DOSのシステムのようで，じつはそのちょっと外側にソフト的なマークをつけていたのです．

ここまでわかれば，このソフトの，実行できるコピーを作る方法はもうはっきりしました．マスタ・ディスクと同じマークをつけるソフトを作るのは，このときには難しかったので，ここではチェック・ルーチンをパスするように書き換えてみました．

このような，MS-DOSでコピーできても実行できないものの変形として，ある非常に有名なワープロ・ソフトのインストール・プログラムもありました．ハード・ディスクにインストールすると，マスタ・ディスクのどこか一部のところが変わってしまい，2回目のインストールができない，というものです．このときには，かなりの長さの「ソフト的なマーク」を同様に解析して，このマークを書くためのプログラム，というものを製作してプロテクトを破ってみました．同じソフトハウスの，こ

〔リストA〕
「第1のプログラム」
解析リスト

```
>debug sample1.exe

        <<< Program 先頭部分 >>

-u 500,538

2AC1:0500 E83D88        CALL    8D40        ##  <-- Check Routine !!!

2AC1:0503 B88A08        MOV     AX,088A
2AC1:0506 8ED0          MOV     SS,AX           <-- SS Set
2AC1:0508 BC9001        MOV     SP,0190         <-- SP Set
2AC1:050B B8C403        MOV     AX,03C4
2AC1:050E 8ED8          MOV     DS,AX           <-- DS Set

2AC1:0510 16            PUSH    SS
2AC1:0511 54            PUSH    SP
2AC1:0512 1E            PUSH    DS

2AC1:0513 E82A88        CALL    8D40        ##  <-- Check Routine !!!

2AC1:0516 1F            POP     DS
2AC1:0517 5C            POP     SP
2AC1:0518 17            POP     SS

2AC1:0519 B81900        MOV     AX,0019
2AC1:051C CD10          INT     10              <-- Display BIOS

2AC1:051E B88105        MOV     AX,0581
2AC1:0521 BB0808        MOV     BX,0808
2AC1:0524 CD10          INT     10              <-- Display BIOS

2AC1:0526 B80307        MOV     AX,0703
2AC1:0529 CD16          INT     16              <-- Keyboard BIOS

2AC1:052B B800B8        MOV     AX,B800
2AC1:052E 8EC0          MOV     ES,AX
2AC1:0530 B80380        MOV     AX,8003
2AC1:0533 CD1A          INT     1A              <-- 時刻 BIOS

2AC1:0535 E80300        CALL    053B            <-- Parameters Set

2AC1:0538 E99500        JMP     05D0            --> Program Start !!


        <<< Master Disc Check Routine >>>

-u 8d40,8d6c

2AC1:8D40 B8AB33        MOV     AX,33AB
2AC1:8D43 8ED8          MOV     DS,AX           <-- DS Set
2AC1:8D45 1E            PUSH    DS
2AC1:8D46 07            POP     ES              <-- ES Set

2AC1:8D47 B400          MOV     AH,00           <-- FDD Reset
2AC1:8D49 B240          MOV     DL,40           <-- 80Track
2AC1:8D4B CD13          INT     13              <-- Disc BIOS

2AC1:8D4D BB0402        MOV     BX,0204     ##  <-- Buffer Address
2AC1:8D50 B402          MOV     AH,02       ##  <-- Sector Read
2AC1:8D52 B001          MOV     AL,01       ##  <-- Sector Count = 1
2AC1:8D54 B550          MOV     CH,50       ##  <-- Track NO.= 80 !!!
2AC1:8D56 B101          MOV     CL,01       ##  <-- Sector NO.= 1~~
2AC1:8D58 B600          MOV     DH,00       ##  <-- Head NO.= 0
2AC1:8D5A B240          MOV     DL,40       ##  <-- Drive NO.= 0 (2DD)
2AC1:8D5C CD13          INT     13          ##  <-- Disk BIOS

2AC1:8D5E 72ED          JB      8D4D            <-- Error = Try Again !

2AC1:8D60 BE0402        MOV     SI,0204         <-- Buffer for Check
2AC1:8D63 8B04          MOV     AX,[SI]
2AC1:8D65 3DFFFF        CMP     AX,FFFF         <-- Data = [FFFF] ?
2AC1:8D68 7402          JZ      8D6C
2AC1:8D6A EBE1          JMP     8D4D            <-- Error = 無限Loop !!!
2AC1:8D6C C3            RET                     <-- OK = Return

-q

>
```

れも有名な図形プロセッサ・ソフトのプロテクトもまったく同様であり，マークのごく一部が違っているだけでした．

第2の例は，コピーどころかDIRで表示することもできない，それもMS-DOSのシステムとしては正常なディスクとしても認めない，というソフトです．

DOSコールの機能呼出し(36h)で見ると，「全クラスタ数」が異常であり，さらに(4Eh, 47h, 44h)とことごとくエラーで，どうもこれはMS-DOSではない，と考えざるを得ないことになりました．ディスクのアクセスはBIOSコールか，それもダメならFDCを直接プログラムで操作することになります．

ここでいろいろなマニュアルをひろげ，パソコンは立上り時にまず「**ブート・ローダ**」という短いソフトをROM BIOSによって読み込み，つぎにこのブート・ローダでプログラム本体を読み込むのだ，というメカニズムを初めて知りました．ブート・ローダが変われば，同じ8086のソフトなら，MS-DOSでなくてもいいのです．

そこでまず，8086のリセット・ベクタ先からdebugによって逆アセンブル・リストを解析し，このパソコンがリセット時に何をしているかを調べました(**リストB**)．CPUとはさすがに速いもので，リストにあげた20くらいの仕事をサッとこなして，ようやくブート・ローダを読み込んでいました．

そこではマニュアルにあったトラック0，セクタ1というのを読んで，メーカ名のIDチェックを行い，ようやくブート・プログラムへとジャンプしています．以上の

〔リストB〕「第2のプログラム」解析リスト

```
<<< Power ON Reset Routine Sequence >>>

     1. Disable NMI , Disable INTs , Clear Memory Mapping
     2. Sound Chip OFF , VIDEO OFF , FDD Motor OFF
     3. CPU Test
     4. 8255 Initialize
     5. Video Gate Array Initialize
     6. RAM Mapping
     7. RAM Test
     8. Interrupts Initialize
     9. 8259 Initialize
    10. 8253 Initialize
    11. CRT Check
    12. Keyboard Initialize
    13. VRAM Test / Initial Screen Display
    14. Kanji ROM Test
    15. Memory Size Check / Display
    16. Keyboard Test
    17. Cassette Interface Test
    18. RS-232-C Port 8250 Initialize
    19. FDD Test
    20. Goto BOOT LOADER !!!   #####

<<< Boot Strap Loader Sequence >>>

     1. Head = [0] , Drive = [0]
     2. Reset FDD -- 4 Times
     3. Track = [0] , Sector = [1]
     4. Read Data <INT 13h>
     5. Data Format Check
     6. If Error --> Error Routine
     7. Jump to Boot Program Location !!!   #####

<<< Target Disk Boot Program >>>

-u e63:0 a5

0E63:0000 00            JMP     0013            <-- Jump to Check Routine

0E63:0013 8CC8          MOV     AX,CS           <-- AX = CS
0E63:0015 8ED8          MOV     DS,AX           <-- DS = CS
0E63:0017 8EC0          MOV     ES,AX           <-- ES = CS
0E63:0019 8ED0          MOV     SS,AX           <-- SS = CS
0E63:001B BC007C        MOV     SP,7C00         <-- Stack Pointer Set

0E63:001E B240          MOV     DL,40           <-- Drive=[0] , 80 Track
0E63:0020 B400          MOV     AH,00           <-- FDD Reset
0E63:0022 CD13          INT     13              <-- Disc BIOS

0E63:0024 C41E077C      LES     BX,[7C07]       <-- Destination Address Set
0E63:0028 BEA67C        MOV     SI,7CA6         <-- Offset Set
```

ROM-BIOS中のReset以降の処理シーケンス

ROM-BIOS中のブート・ローダ読込みシーケンス

準備で，やっとターゲットのディスクにとりつくことができました．まずはブート・プログラム・エリアをdebugのディスク読込みコマンドによって取り込んで，例によって逆アセンブル・リストにしてみました．

すると，異常なチェック・ルーチンにいきなり突き当たりました．結果からいえば，プログラムは全体が四つの部分に分かれていて，4パートの読込みブロックがありました．そして，

第1ブロック…512バイト/セクタ，8セクタ1回取込み

第2ブロック…512バイト/セクタ，8セクタ1回取込み

第3ブロック…1024バイト/セクタ，1回3セクタずつ，2トラックずつ飛び飛びに11回取込み

第4ブロック…256バイト/セクタ，1回5セクタずつ，2トラックずつ飛び飛びに10回取込み

というようにプログラムを読み込んでいるのです．これはプログラムの読込みそのものがバラバラなので，プロテクトのチェック・ルーチンをパスする，という方法では通用しません．

マスタ・ディスクそのもののフォーマットが普通ではないので，これをそのまま再現するのは非常に困難でした．そこで，マスタ・ディスクを㊚とすると，㊚を読み込みながら，通常のMS-DOSフォーマットされたディスクに㊊のプログラムを書き込む第1のプログラムと，㊊から別の㊊へコピーする第2のプログラムを製作し

```
0E63:002B E86900        CALL    0097            <-- Vector [1Eh] Set
0E63:002E B90102        MOV     CX,0201         <-- Track=[2],Sector=[1]
0E63:0031 BA4000        MOV     DX,0040         <-- Head=[0],Drive=[0],2DD
0E63:0034 B008          MOV     AL,08           <-- Sector Count = 8
0E63:0036 E85400        CALL    008D            <-- Sector Read <BIOS>
0E63:0039 81C30010      ADD     BX,1000         <-- Destination Address Set
0E63:003D 80C502        ADD     CH,02           <-- Track NO. = (+2)
0E63:0040 B008          MOV     AL,08           <-- Sector Count = 8
0E63:0042 E84800        CALL    008D            <-- Sector Read <BIOS>

0E63:0045 FF1E077C      CALL    FAR [7C07]      <-- Upper Half Screen Display

0E63:0049 C41E0B7C      LES     BX,[7C0B]       <-- Destination Address Set
0E63:004D BEAD7C        MOV     SI,7CAD         <-- Offset Set
0E63:0050 E84400        CALL    0097            <-- Vector [1Eh] Set
0E63:0053 B90615        MOV     CX,1506         <-- Track=[21],Sector=[6]
0E63:0056 BA4001        MOV     DX,0140         <-- Head=[1],Drive=[0],2DD
0E63:0059 B003          MOV     AL,03           <-- Sector Count = 3
0E63:005B E82F00        CALL    008D            <-- Sector Read <BIOS>
0E63:005E 81C3000C      ADD     BX,0C00         <-- Destination Address Set
0E63:0062 80ED02        SUB     CH,02           <-- Track NO. = (-2)
0E63:0065 79F2          JNS     0059            <-- Loop(21,19,17,.....,3,1)

0E63:0067 FF1E0B7C      CALL    FAR [7C0B]      <-- Lower Half Screen Display

0E63:006B C41E0F7C      LES     BX,[7C0F]       <-- Destination Address Set
0E63:006F BEB47C        MOV     SI,7CB4         <-- Offset Set
0E63:0072 E82200        CALL    0097            <-- Vector [1Eh] Set
0E63:0075 B90113        MOV     CX,1301         <-- Track=[19],Sector=[1]
0E63:0078 BA4001        MOV     DX,0140         <-- Head=[1],Drive=[0],2DD
0E63:007B B005          MOV     AL,05           <-- Sector Count = 5
0E63:007D E80D00        CALL    008D            <-- Sector Read <BIOS>
0E63:0080 81C30005      ADD     BX,0500         <-- Destination Address Set
0E63:0084 80ED02        SUB     CH,02           <-- Track NO. = (-2)
0E63:0087 79F2          JNS     007B            <-- Loop(19,17,15,.....,3,1)

0E63:0089 FF2E0F7C      JMP     FAR [7C0F]      --> OK. Program Start !!!

0E63:008D B402          MOV     AH,02           <-- Sector Read
0E63:008F CD13          INT     13              <-- Disc BIOS
0E63:0091 7201          JB      0094            <-- Error ?
0E63:0093 C3            RET                     <-- OK = Return
0E63:0094 E97CFF        JMP     0013            <-- Error = 無限Loop

0E63:0097 1E            PUSH    DS              <-- DS Keep
0E63:0098 33C0          XOR     AX,AX           <-- AX=[0000]
0E63:009A 8ED8          MOV     DS,AX           <-- Set to DS : Vector Area
0E63:009C 89367800      MOV     [0078],SI       <-- Vector Address
0E63:00A0 8C0E7A00      MOV     [007A],CS       <-- Vector Segment
0E63:00A4 1F            POP     DS              <-- DS Recover
0E63:00A5 C3            RET
```

てみました．㋒は全セクタが512バイト/セクタにされているだけで，四つのブロックはそのままとしました．

こうして完成した㋒のコピーが正常に動いたときは，とても有名なこのゲーム・ソフト自体からよりも，はるかに大きな充足感が得られたのはいうまでもありません．

第3の例は，ちょっとひとひねりしてあって面白いと思ったプロテクトです．MS-DOSのシステムにはしっかりのっていて，DIRもCOPYもDISKCOPYも受け付けるのに，マスタ・ディスクでないと止まるタイプのものです．例によってデバッガに入り，pコマンドとtコマンドで少しずつ走らせ，デッドロックするサブルーチンを見つけだし，その中に入っていきました(**リストD**)．すると〈INT 41h〉という見慣れないソフト割込みがありました．こんなアドレスはDOSにもBIOSにもなく，マニュアルにはユーザ領域とあります．そこでベクタ・アドレス・エリアからプログラム・アドレスを求め(ここでは〈41h〉×4=〈104h〉，よってセグメント=0AB9，アドレス=0081)そこからリストにしてみると，あるある，いかにも暗号チェック・ルーチンぽい部分がありました．ソ

〔リストC〕「第2のプログラム」コピー用プログラム(㋐→㋒，㋒→㋒)

```
;*-------------------------------------------------------------*
;*      Master --> Child Copy Program                          *
;*-------------------------------------------------------------*
        cgroup  group   code

gdata segment
        dta     db 10000 dup (0aah)
gdata ends

code segment public 'code'
        assume  cs:code,ds:gdata,es:extra,ss:stack

main:;========== Main Routine ==========
        mov     ax,stack
        mov     ss,ax
        mov     ax,offset ss:ss_top
        mov     sp,ax
        mov     ax,extra
        mov     es,ax
        call    gds_set
        mov     ah,0
        mov     dl,01000000b
        int     13h  ; Disk <A> System Reset
        mov     ah,0
        mov     dl,01000000b
        int     13h  ; Disk <B> System Reset
        call    gds_set
        mov     ax,ds
        mov     es,ax
        mov     bx,offset dta
        mov     cx,0001h
        mov     dx,0040h
        mov     ax,0201h
        int     13h  ; Boot Load from <A>
        call    changer
        mov     dx,0041h
        mov     ax,0301h
        int     13h  ; Boot Save to <B>
        mov     si,offset bank1
        call    disc_para_set
        mov     cx,0201h
        mov     dx,0040h
        mov     ax,0208h
        int     13h  ; 1st Load from <A>
        mov     dx,0041h
        mov     ax,0308h
        int     13h  ; 1st Save to <B>
        mov     cx,0401h
        mov     dx,0040h
        mov     ax,0208h
        int     13h  ; 2nd Load from <A>
        mov     dx,0041h
        mov     ax,0308h
        int     13h  ; 2nd Save to <B>
        mov     ch,15h
loop_001:
        mov     si,offset bank2
        call    disc_para_set
        mov     cl,06h
        mov     dx,0140h
        mov     ax,0203h
        int     13h ; 3rd Load from <A>
        mov     si,offset bank1
        call    disc_para_set
        mov     cl,03h
        mov     dx,0141h
        mov     ax,0306h
        int     13h ; 3rd Save to <B>
        sub     ch,2
        jns     loop_001
        mov     cx,1301h
loop_002:
        mov     si,offset bank3
        call    disc_para_set
        mov     dx,0140h
        mov     ax,0205h
        int     13h                 ; 4th Load from <A>
        mov     si,offset bank1
        call    disc_para_set
        mov     dx,0041h
        mov     ax,0305h
        int     13h                 ; 4th Save to <B>
        sub     ch,2
        jns     loop_002
        mov     ah,4ch
        int     21h                 ; DOS Return
bank1:
        db      0efh,003h,025h,002h,009h,02ah,0ffh
bank2:
        db      0efh,003h,025h,003h,009h,02ah,0ffh
bank3:
        db      0efh,003h,025h,001h,009h,02ah,0ffh

changer:;========== Data Change Routine ==========
        call    gds_set
        push    bx
        mov     bx,97h
        mov     dta[bx],0c3h    ; Return
        mov     bx,54h
        mov     dta[bx],03h     ; Sector No. = 3
        mov     bx,5ah
        mov     dta[bx],06h     ; Sector Count = 6
        mov     bx,7ah
        mov     dta[bx],0       ; Head = 0
        pop     bx
        ret

disc_para_set:;========== Disc Parameter ==========
        push    es
        mov     ax,0
        mov     es,ax
```

フト割込み中なので，ディスクBIOSをfar callしているところが面白いと思いました．

このルーチンは，じつは細かく解析していません．おおまかに4回，おそらくマスタ・ディスクのMS-DOSエリア外に書かれたパスワードを読み込み，それをさらに，非常に複雑な暗号化ルーチンに通して最終的なチェックをしています．これにはほとんどあきらめかけました．が，ふと気づいてみると，セグメント・アドレスが異常に低いのです．これは普通のEXEファイルの置かれるセグメント・アドレスとは遠く離れていて，さらに調べてみると，特殊なデバイス・ドライバ〈***.sys〉が目につきました．結論をいえば，マスタ・ディスクの〈config.sys〉に登録されたこのデバイス・ドライバが，この〈INT 41h〉のルーチンをシステムにロードし，そのベクタをセットしているのでした．こうなればターゲットのEXEプログラムではなく，デバイス・ドライバのほうに少し手を加えるだけで，コピーしたシステム・ディスクでも正常に立ち上がるようになりました．コピー・プロテクトといえばソフト本体のどこかにチェックがある，という先入観のウラをかいた，いいアイデアと感心しました．

```
        mov     di,0078h
        mov     es:[di],si
        mov     di,007ah
        mov     es:[di],cs
        pop     es
        ret
gds_set:;========== Data Segment Set ==========
        push    ax
        mov     ax,gdata
        mov     ds,ax
        pop     ax
        ret

code    ends

extra segment
        es_area db 1024 dup (?)
extra ends

stack segment stack
        ss_area db 2048 dup (?)
        ss_top  label word
stack ends

        end     main
;*-----------------------------------------------*
;*      Child --> Child Copy Program             *
;*-----------------------------------------------*
        cgroup  group   code

gdata segment
        dta     db 10000 dup (0aah)
gdata ends

code segment public 'code'
        assume  cs:code,ds:gdata,es:extra,ss:stack

child:;========== Main Routine ==========
        mov     ax,stack
        mov     ss,ax
        mov     ax,offset ss:ss_top
        mov     sp,ax
        mov     ax,extra
        mov     es,ax
        call    gds_set
        mov     ah,0
        mov     dl,01000000b
        int     13h     ; Disk <A> System Reset
        mov     ah,0
        mov     dl,01000000b
        int     13h     ; Disk <B> System Reset
        call    gds_set
        mov     ax,ds
        mov     es,ax
        mov     bx,offset dta
        mov     cx,0001h
        mov     dx,0040h
        mov     ax,0201h
        int     13h     ; Boot Load from <A>
        mov     dx,0041h
        mov     ax,0301h
        int     13h         ; Boot Save to <B>
        mov     cx,0201h
        mov     dx,0040h
        mov     ax,0208h
        int     13h         ; 1st Load from <A>
        mov     dx,0041h
        mov     ax,0308h
        int     13h         ; 1st Save to <B>
        mov     cx,0401h
        mov     dx,0040h
        mov     ax,0208h
        int     13h         ; 2nd Load from <A>
        mov     dx,0041h
        mov     ax,0308h
        int     13h         ; 2nd Save to <B>
        mov     cx,1503h
loop_001:
        mov     dx,0140h
        mov     ax,0206h
        int     13h         ; 3rd Load from <A>
        mov     dx,0141h
        mov     ax,0306h
        int     13h         ; 3rd Save to <B>
        sub     ch,2
        jns     loop_001
        mov     cx,1301h
loop_002:
        mov     dx,0040h
        mov     ax,0205h
        int     13h         ; 4th Load from <A>
        mov     dx,0041h
        mov     ax,0305h
        int     13h         ; 4th Save to <B>
        sub     ch,2
        jns     loop_002
        mov     ah,4ch
        int     21h         ; DOS Return

gds_set:;========== Data Segment Set ==========
        push    ax
        mov     ax,gdata
        mov     ds,ax
        pop     ax
        ret

code    ends

extra segment
        es_area db 1024 dup (?)
extra ends

stack segment stack
        ss_area db 2048 dup (?)
        ss_top  label word
stack ends

        end     child
```

以上のように，プロテクト解析はじつにスリリングなゲームであり，よい意味でのハッカー気分を十分に楽しませてくれるものです．そして，自分のプログラミングに役立つヒントも多く教えてくれるものです．まだまだ筆者のやっているレベルは初心者のものですが，これからもなるべくコピー・プロテクトされたソフトを選んで購入し，おおいに「勉強」させてもらおうと思っています．

〔リストD〕「第３のプログラム」解析リスト

```
>symdeb sample3.exe

Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]

-u 151f:6d97

151F:6D97 06              PUSH    ES
151F:6D98 8CC8            MOV     AX,CS
151F:6D9A 8EC0            MOV     ES,AX           <-- ES = CS
151F:6D9C BAD9E5          MOV     DX,E5D9         <-- Parameter Set
151F:6D9F 81EACD67        SUB     DX,67CD         <-- ?
151F:6DA3 B8CB67          MOV     AX,67CB         <-- Parameter Set
151F:6DA6 CD41            INT     41              <-- Software Interrupt !!!
151F:6DA8 07              POP     ES

-d 0:100

0000:0100  81 00 B9 0A 81 00 B9 0A-5A 5D 60 00 5A 5D 60 00  ..9...9.Z]`.Z]`.
0000:0110  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0120  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0130  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0140  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0150  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0160  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.
0000:0170  5A 5D 60 00 5A 5D 60 00-5A 5D 60 00 5A 5D 60 00  Z]`.Z]`.Z]`.Z]`.

-u ab9:81 1e8

0AB9:0081 1E              PUSH    DS
0AB9:0082 06              PUSH    ES
0AB9:0083 53              PUSH    BX
0AB9:0084 51              PUSH    CX
0AB9:0085 56              PUSH    SI
0AB9:0086 57              PUSH    DI
0AB9:0087 55              PUSH    BP

0AB9:0088 8CCB            MOV     BX,CS
0AB9:008A 8EDB            MOV     DS,BX           <-- DS = CS
0AB9:008C 803E800000      CMP     Byte Ptr [0080],00
0AB9:0091 8C06E901        MOV     [01E9],ES
0AB9:0095 A3EB01          MOV     [01EB],AX
0AB9:0098 8916ED01        MOV     [01ED],DX
0AB9:009C 8EC3            MOV     ES,BX           <-- 1st Check ?
0AB9:009E 7403            JZ      00A3            <-- YES = Disc Read
0AB9:00A0 E92101          JMP     01C4            <-- NO = Pass !

0AB9:00A3 FB              STI
0AB9:00A4 B419            MOV     AH,19           <-- Current Drive
0AB9:00A6 CD21            INT     21              <-- DOS Call

0AB9:00A8 B690            MOV     DH,90           <-- [A:]
0AB9:00AA 0AC6            OR      AL,DH
0AB9:00AC A2F501          MOV     [01F5],AL       <-- Parameter Set
0AB9:00AF 8A26F201        MOV     AH,[01F2]       <-- Command = [Read]
0AB9:00B3 BB1502          MOV     BX,0215
0AB9:00B6 B504            MOV     CH,04
0AB9:00B8 8A0EF401        MOV     CL,[01F4]
0AB9:00BC B600            MOV     DH,00
0AB9:00BE 8A16F301        MOV     DL,[01F3]
0AB9:00C2 BD8502          MOV     BP,0285
0AB9:00C5 E81A01          CALL    01E2            <-- Disc BIOS : Read

0AB9:00C8 7377            JNB     0141            <-- Error !
0AB9:00CA 80FCB0          CMP     AH,B0           <-- Password ?
0AB9:00CD 7572            JNZ     0141            <-- Error !
```

ポイントの部分
(INT 41h発見！)

INT 41hのベクタ・アドレス
(セグメント＝0AB9)
(オフセット＝0081)

そこから逆アセンブル表示

```
0AB9:00CF B82200          MOV     AX,0022
0AB9:00D2 A3EF01          MOV     [01EF],AX
0AB9:00D5 BBF801          MOV     BX,01F8
0AB9:00D8 E88F00          CALL    016A            <-- 暗号 Check Routine ?
0AB9:00DB A0F501          MOV     AL,[01F5]       <-- 2nd Password Access
0AB9:00DE BB1000          MOV     BX,0010
0AB9:00E1 8A16F301        MOV     DL,[01F3]
                           :
0AB9:0141 E99600          JMP     01DA            <-- Error Routine

0AB9:0144 8A26F101        MOV     AH,[01F1]       <-- 4th Password Access
0AB9:0148 BB1000          MOV     BX,0010
0AB9:014B B504            MOV     CH,04
0AB9:014D A0F501          MOV     AL,[01F5]
0AB9:0150 B600            MOV     DH,00
0AB9:0152 BD8502          MOV     BP,0285
0AB9:0155 8A0EF401        MOV     CL,[01F4]
0AB9:0159 E88600          CALL    01E2            <-- Disc BIOS : Read

0AB9:015C B80100          MOV     AX,0001
0AB9:015F A3EF01          MOV     [01EF],AX
0AB9:0162 BB2A02          MOV     BX,022A
0AB9:0165 E80200          CALL    016A            <-- 暗号 Check Routine ?
0AB9:0168 EB5A            JMP     01C4            --> Normal Exit !
0AB9:016A 891EF601        MOV     [01F6],BX
0AB9:016E BB5302          MOV     BX,0253
0AB9:0171 BE8502          MOV     SI,0285
0AB9:0174 BD0000          MOV     BP,0000
0AB9:0177 8B0EEF01        MOV     CX,[01EF]       暗号 Check Routine
                           :                      (詳しい動作は未解析)
0AB9:01BA 8BF8            MOV     DI,AX
0AB9:01BC BE5302          MOV     SI,0253
0AB9:01BF F3              REPZ
0AB9:01C0 A6              CMPSB
0AB9:01C1 7516            JNZ     01D9            --> ERROR : 暴走
0AB9:01C3 C3              RET

0AB9:01C4 8E06E901        MOV     ES,[01E9]       <-- Normal Finish
0AB9:01C8 8B1EEB01        MOV     BX,[01EB]
0AB9:01CC A1ED01          MOV     AX,[01ED]
0AB9:01CF 268907          MOV     ES:[BX],AX
0AB9:01D2 C606800001      MOV     Byte Ptr [0080],01
0AB9:01D7 EB01            JMP     01DA            <-- Check OK !!!

0AB9:01D9 58              POP     AX              <-- 異常時

0AB9:01DA 5D              POP     BP              <-- 正常時
0AB9:01DB 5F              POP     DI
0AB9:01DC 5E              POP     SI
0AB9:01DD 59              POP     CX
0AB9:01DE 5B              POP     BX
0AB9:01DF 07              POP     ES
0AB9:01E0 1F              POP     DS
0AB9:01E1 CF              IRET

0AB9:01E2 9C              PUSHF
0AB9:01E3 9A821A80FD      CALL    FD80:1A82       <-- DISC BIOS !!
0AB9:01E8 C3              RET

-q

>
```

MASMのマクロ使用例

```
;=======================================================;
;       Macro Sample : NUM-Input Program for BATCH      ;
;=======================================================;

code    segment
        assume  cs:code,ds:code
        org     100h

;------ マクロの定義 ----------------------------------------

putchar macro   char                    ; 1 Character Display
        mov     dl,char
        mov     ah,02h
        int     21h
        endm

put_c   macro                           ; [CL] Display
        mov     dl,cl
        mov     ah,02h
        int     21h
        endm

puts    macro   message                 ; Strings Display
        mov     dx,offset message+100h
        mov     ah,09h
        int     21h
        endm

console macro                           ; 1 Data Input Waiting --> [CL]
        mov     ah,08h
        int     21h
        mov     cl,al
        endm

check   macro   data_low,data_high,jump_address
        mov     al,cl
        cmp     al,data_low
        jb      jump_address
        cmp     al,data_high
        ja      jump_address
        endm

asc_to_bin  macro                       ; [CL] --> [AL]
        mov     al,cl
        sub     al,30h
        endm

exit_a  macro                           ; Exit with [AL]
        mov     ah,4ch
        int     21h
        endm

exit    macro   code                    ; Exit with Code
        mov     ax,4c00h+code
        int     21h
        endm

        cr      equ     0ah
        lf      equ     0dh

;------ Program 本体は こんなに簡単 ! ---------------------------------

start:
        putchar  cr                     ; 改行
        putchar  lf                     ; 行頭
        puts     request_message        ; 入力の要求
        console                         ; 数字入力待ち
        put_c                           ; Echo Back
        check    '1','9',start          ; 1から9の間かな?
        putchar  cr                     ; OKなら
        putchar  lf                     ; 改行して
        asc_to_bin                      ; Data Conversion 結果の
        exit_a                          ; 値を返す

request_message:
        db      '          1－9の間から選択して下さい．．．$'

code    ends
        end     start
```

MASMのマクロを活用した例として，バッチ・プログラムで使えるツールを書いてみました．

バッチからメニューを表示してこのツールを呼び，数字キーによる選択を待ちます．選択結果を

```
if errorlevel ＊＊……
```

によって利用して，必要な処理にジャンプすればいいわけです．

プログラム本体部分にCPUニモニックが一つもない，という極端な例ですが，このようにつぎつぎとマクロを作っていくと，一種の**オリジナル言語**のようになって楽しくなります．

3. 基礎レベル：汎用ボード・マイコン

基礎レベルでは，STD，VME あるいは 98 バス・ボードなど，市販の標準バス・ボードをベースにしてシステムを構成する例を軸に話を展開する．これら市販ボードを組み合わせてシステムを開発する場合の要点は，「目的の仕事をハードとソフトの領域にどう分けるか」，「ハード環境を動作チェックする際のノウハウ」，「EPROM に焼くソフトをどう作るか」，それに「システムの信頼性の検討」などである．ここでは，とくに，ソフトの開発(マイコン開発支援装置，パソコン ICE，デバッグ・モニタ…)と信頼性設計(瞬断対策，ノイズ対策，ソフトのデバッグ手法，自己診断プログラム…)について詳しく述べる．（編集部）

前章では目的のシステム自体がパソコンを含んでいましたが，これでは経済的とはいえません．そこで，つぎの段階では**パソコンを開発ツールと位置づける**ことにして，もう少し現実的に，基板の状態のボード・マイコンという製品について検討します．これは，8，16 ビットのポピュラな CPU と周辺 LSI を搭載した，コンパクトな汎用システムです．基板内部の動作が保証され，目的の仕事に応じて適切なボードを組合せ可能なので，コスト・パフォーマンスと信頼性の高いシステムを容易に開発できるものです．

標準バス・標準インターフェース

パソコンの CPU が，ユーザの知らないうちに，グラフィック関係や DOS のシステム・サービスなどの部分で，ユーザの使用しない過剰な処理を多く行っているのに対して，ボード・マイコンでは，CPU に対して，必要な仕事だけをプログラムします．このため，とくに Basic のような汎用言語を用いて 16 ビット・パソコンで組んだ仕事の大部分を，意外にも 8 ビット CPU ボードのシステムで実現できるようです．ここでは，前章のパソコンで実現したものに似た例題として，

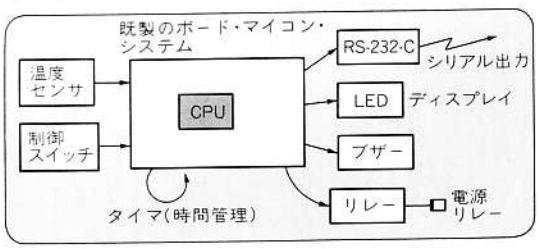
例 3

外部の温度センサによって温度を計測し，刻々とそのデータを LED に表示しながら RS-232-C* からも出力する．温度が設定値を超えたら，ブザーが鳴って電源リレーを切る．

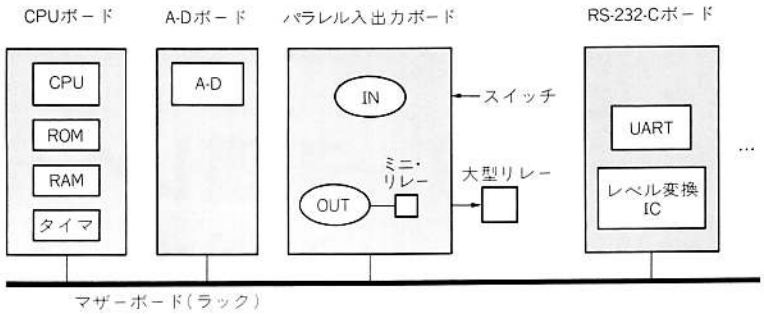
というようなシステムを考えてみましょう．

まずは中心となるボード・マイコンを選択します(図3.1)．この例のような処理量なら，あとで多少の追加仕様があっても 8 ビット CPU で十分と思われますが，その処理量の大まかな判断は，ある程度の経験によってつかめてきます(8 ビット CPU ボードと 16 ビット CPU ボードの価格はかなり差があるので，よほどの要求がないと，簡単なシステムに 16 ビット CPU ボードは使えないなあ，というむしろ経済的な直感が先に立つようですが…)．そして，CPU ボードだけでなく，温度センサからの出力アナログ信号を取り込む

〔例 3〕既製のボード・マイコンによる温度計測システム



〔図3.1〕
複数のボードによって
システムを組む



A-D 変換ボードや，パラレル入出力ボード，RS-232-C ボードなどが必要となるので，拡張性のあるボード・セットであるほうが好都合です．このような基板の信号バスには，〈STD〉や〈VME〉といった名前の**標準バス**があります(**図3.2**)．この標準バスを採用するメリットとしては，CPU の細かいタイミング信号まで外部から制御できること，同等規格の他社のボードとの互換性があること，などがあり，多少コスト高でも標準バスのものを使用したほうが，結果的には安心できるようです．

また，基板のバス信号規格とは別に，〈RS-232-C〉，〈GPIB〉*，〈SCSI〉*といった名前の，信号を共通の規格で通信するための**標準インターフェース**の規約もあります(**図3.3**)．これは，複数のディジタル機器の間でのデータ通信や，パソコンとの接続などに活用されるばかりでなく，システム内部に複数の CPU 基板をもった場合の内部制御信号としても，デバッグが容易であるために重宝する規格です．今回の例では，RS-232-C を使用してもっとも簡単なシリアル信号を出力し，パソコン側のターミナル・ソフトによって表示させることを想定しています．

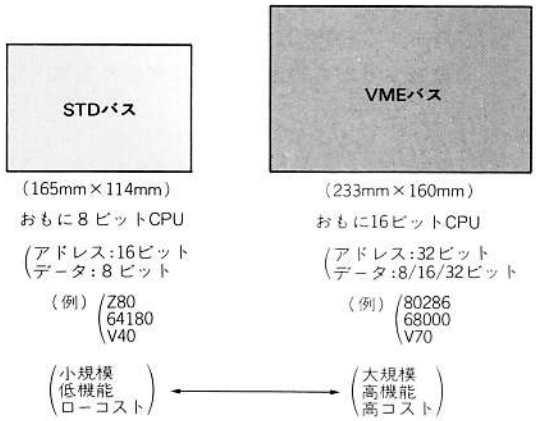
システム設計(市販ボード利用の場合)

パソコンでは基本的なハードウェアがほぼ定まっていますが，ボード・マイコンのシステムではハード自体の構成が自由なので，まず，**目的となる仕事をハードとソフトの領域に分ける**検討が必要になります．CPU のソフトで何をして，その周辺としてハードが何をするのか，という役割分担と，具体的にシステムの各部分での信号形態もここで考えます．**例 3** の場合，まず温度センサの出力信号はアナログの値で，このままでは CPU の処理の対象とならないので，A-D 変換ボードは必須となります．

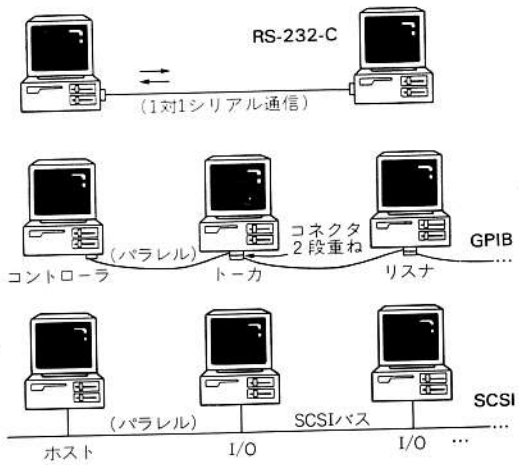
また，LED*表示やリレー制御はディジタル信号なので，たんに出力ポートがあればすみます．

RS-232-C の出力レベルは通常の CPU の TTL*/

〔図3.2〕標準バスの例



〔図3.3〕標準インターフェースの例



CMOS*レベルとは異なるので，専用のインターフェースICが必要です．一般に，CPU周辺としてRS-232-Cを用いたシリアル通信を担当するUART*というLSIがあり，これとインターフェースICとをセットにして「RS-232-Cボード」という拡張ボードが作られています．これを使えば，CPUはシステム・バス上でUARTとやりとりするだけでRS-232-Cを介した通信が行えるわけです．

CPUのソフトの仕事としては，A-Dボードから温度データを得て，補正した温度データをLEDボードに出力して表示し，一方で，UARTを介してRS-232-Cに出力し，さらに温度データによってはブザーやリレーを制御する，という内容となります．

ここでの入力情報は温度センサからの出力データという，それほど高速の処理の必要もないものなので，CPUのソフトのスピードで十分ですが，もし，音声信号や映像信号などの超高速処理であれば，A-D*コンバータも非常に高速なものが要求されるばかりでなく，CPUがソフト的に処理するスピードでは遅くて使えない，という領域になります．このような場合は，

① A-D変換したデータをバッファ・メモリにため込むような専用のハードウェア基板に対し，CPUはDMA転送のような高速処理でインターフェースする．

② A-D変換したデータを，さらにホストCPUが必要なデータ形式にまで処理・圧縮するようなハードウェア，もしくはオン・ボードの専用CPUをもつインテリジェントな拡張ボードを用いてホストCPUはこれとインターフェースする．

という対応も必要になります．

また，ボード上のCPUソフトをどの言語で開発するか，という選択も検討します．64180やV30あたりの有名なCPUならアセンブラばかりでなく，最近はCコンパイラなども多く出ています．

Cの中にも，まずアセンブラのソース・プログラムの形式に変換するようなCもあり，いろいろに使い分けできそうです．ここでは，第4章の「専用CPUボード」へのつながりを考えて，伝統的なアセンブラで製作することを想定しました．これは，アセンブラで作れる人が，ときにはCによって楽をして，いろいろなメリットを狙うような余裕をもつのに対して，Cでしか作れない人がいざアセンブラをやる必要にせまられても，なかなか余裕が出てこない，という筆者の(周囲の)経験からの提案です．

さらに，ソフトとハードの境界領域のもう一つの要素として，**システムの信頼性**についての検討も必要になりますが，ここでは深入りせず，この章の最後に，まとめて検討することにします．

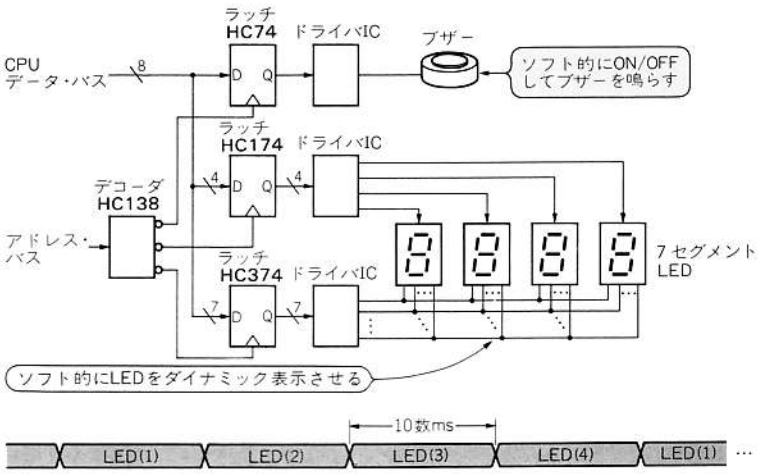
パソコンの場合，ケース後面の拡張用外部端子から内側は，マニュアル通りのブラックボックスとして扱えば十分でしたが，ボード・マイコンの場合はちがいます．個々の基板のマニュアルしかありませんから，システムを設計する場合には，その組合せに関するハードウェア上の理解が必要になります．

まず，ラックのマザーボードを介して接続されるバス信号の仕様を検討して，必要なボードを決定します．この例題の場合には，RS-232-Cは出力のみで，データの量もそれほどではないので，割込みを使わなくても十分動作します．しかしせっかく送信・受信できるUARTなので，のちのちホストなりコンソールからの制御を受信するような拡張の可能性も考えて，RS-232-Cとしては，割込み信号まで対応したバス規格のものにします．タイマ割込みはCPUボード上のタイマから直接使います．また，センサと接続するA-Dボードや，スイッチ・リレー用の入出力ボードは標準品としても，LED表示やブザー出力用として適当なボードがなければ，共通規格のユニバーサル基板上に自作する，とかの対応も必要です(**図3.4**)．

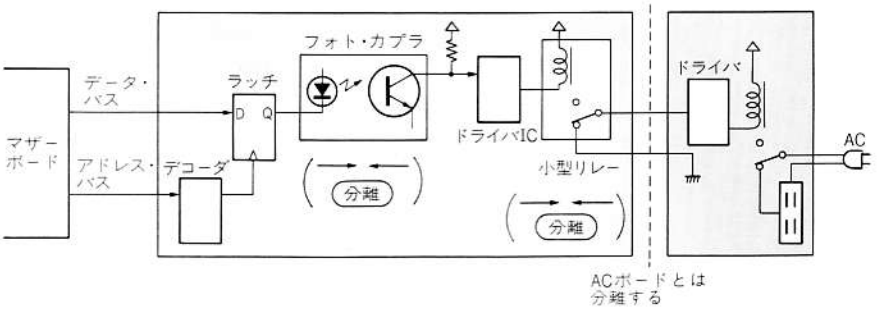
ボードを決定すれば，あとはアナログ的な配慮を考えましょう．ここでは温度センサ自体についてはあまり詳しく触れずに，たんにデータ補正テーブルを作って参照するのみとします．A-Dボードの電源ラインをディジタルと分離するのも重要です．また，リレー制御の出力ボードはパソコンのときと同様に，フォト・カプラでアイソレートした小型リレーから，実際の電源用大型リレーへの2段構成が安全でしょう(**図3.5**)．

以上のようなシステム設計が終われば，ラックを含めてボード類を入手し，自作ボード，電源，センサ，リレー，ブザー，ケースなどの部品を含めた全体のハードウェアを試作して，ようやくつぎのソフトの段階へと進むことになります．実際には電源供給の確認，バス信号・各種信号接続の確認，コネクタ・ケーブルの接続，リセット系・クロック系の処理などの，ハードウェアのノウハウが必要となります(p.208の**Appendix 3.1「新CPU基板が動くまで」**参照)．また，使用する測定器も，テスタだけでなくシンクロ，ロジアナ，

〔図3.4〕
自作基板の回路例
(LED/ブザー・ボード)



〔図3.5〕
AC電源をコントロール
するときにはアイソレー
ションが重要

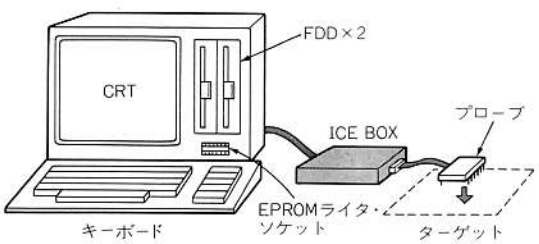


スペアナといった，一応「プロの道具」(**コラム3.1**)が登場することになります．

開発支援装置

ハードの環境が整えば，いよいよボード上のCPUのプログラム開発がつぎのステップです．ボードにはCPUのプログラムROMのソケットがあるので，最終的にはここにプログラムを書き込んだEPROM*を挿入するわけです．ところがEPROMのデータは機械語で，簡単には開発できませんから，なんらかの**開発ツール**が必要になります．ここでは開発環境の歴史にしたがって，まず，専用のハードウェアとして製品になっている「**マイコン開発支援装置**」という高額のツールを使う場合を例にとってみます．

〔図3.6〕古典的マイコン開発ツール



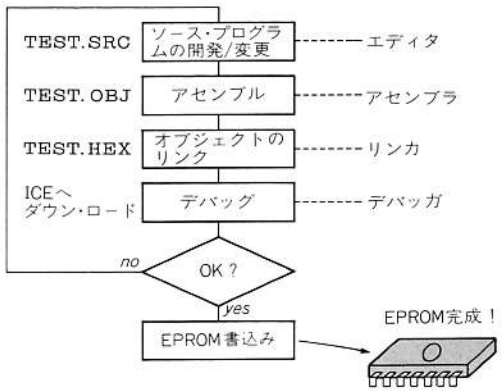
この開発ツールは一般に，2ドライブ程度のディスク・ドライブ，CRT*，キーボードをもつ，パソコンと同様の外観の装置です(**図3.6**)．しかし専用のシステム・ソフトしか走らないので，むしろ現在の机上型ワープロに近い専用機といえます．開発の手順としては，プログラム開発のためのエディタによってソース・プログラムをアセンブリ言語で書き，つぎにアセンブラやリンカによって機械語形式のファイルに変換します．これを内蔵のROMライタ・ソケットによってEPROM化したり，あるいはデバッガに移って実行したりします(**図3.7**)．

開発ツールのデバッガは，パソコンで行うようなソ

フト的デバッグだけでなく，実際にCPUのソケットに挿入して実行を代行するICE*をもっていて，非常に強力にデバッグを支援します．特定のアドレスやポートへのアクセスでブレークしたり，ステップ動作でレジスタの変化を見たり，一定時間の動作をトレースしたりする機能によって，CPUの微妙な動作をほとんど追跡することができ，プログラムのデバッグはここで完璧になります．あとは最終版のプログラムをEPROMに焼いて完成，となります．

ここの例題のケースでも，まずは起動からリレー制御，スイッチ検出，終了のメイン・ルーチンを製作・デバッグします(**リスト3.1**)．さらにLED表示ルーチン，A-Dボードの温度検出ルーチンを加えます(**リスト3.2**)．大まかな処理が単純な逐次処理として動作確認できれば，つぎにタイマ割込みルーチンをジョイントしながら全体をデバッグします．RS-232-Cは，まず相手のパソコン側の「受信-表示モニタ」を製作します(**リスト3.3**)．つぎに，タイマ割込みから一定時間をカウントするルーチンによってフラグが立てられると，シリアル通信LSIにデータを送ってRS-232-C出力にする動作を確認し，最後にLEDと転送データの一致を確認します(**リスト3.4**)．以上のような流れで，エディタでの修正とデバッグのループを繰り返せば，簡単にプログラムは完成レベルに収束していきます．

〔図3.7〕開発ツールによるマイコン開発の手順



このような開発ツールのメリットは，全体の作業を1台の中にすべてまとめてあるので，非常に操作性が良い点にあります．また，ICEとアセンブラを取り替えると別のCPUにも使えたり，DOSは必要機能に限定したオリジナルなので，ICEへのオブジェクト・ロードなどの内部転送が高速です．一方で，低レベルの専用エディタの使いにくさを我慢したり，ICEのないCPUには使えない，という欠点もあります．また，高額のツールなので，CPUの世代交代のスピードに追従しにくい点も，現実に設備として導入する場面では問題となるようです．

コラム3.1

プロの測定器とは

マイコン技術者，あるいは(ディジタル)電子回路技術者として，どのような測定器をツールとして活用できればいいのでしょうか．じつは，筆者もそれほど自信がないのですが，ざっと思いつくままあげてみると，オシロスコープ，ディジタル・テスタ，ストレージ・スコープ，ロジック・アナライザ，スペクトル・アナライザ，デバッガ(ICE)，周波数カウンタ，発振器などがあります．

このうち，ほとんど使ったことがないストレージやロジアナやカウンタについては，コメントすべき内容もありません．また，テスタを使うのは，基板を作った最初の導通チェッカ程度のもので，スペアナや発振器を使うのも，特殊な状況の計測のときにかぎります．こうしてみると，ほとんどの場合，シンクロとデバッガがあれば，筆者の場合にはこと足りているようです．

電子回路の開発を始めたころは，ハード/ソフト・デバッグの際に，あらゆる信号の状態を全部把握しておきたくて，ロジアナをそこらじゅうに接続したり，オシロは多チャネルのものを，ICEのクリップもあちこちに，と，セッティングに多くの時間をかけました．それが，先輩の作業を真似ながら経験を積み，次第に慣れてくると，手元に4チャネルのシンクロがなければ2チャネルでもいいや，ICEはプローブだけ挿せばいいや，と手抜きになっていくのです．つまり，実際に有効なチェック・ポイントが「見えて」くるわけで，うまく言葉になりませんが，これがノウハウというものなのでしょう．

そしてさらに，デバッグの際にうまく手抜きできるような回路設計をするテクニックや，必要ならば簡単な計測ツールは作ってしまう姿勢が，本当のプロの計測環境というものなのかな，と思います．

〔リスト3.1〕
ラベル定義/初期化/メイン・ルーチンなどの例
(一部ルーチンは省略)

```
0000                    name    sample

4000            work            equ     04000h  ; RAM Top Address
7FFF            stack           equ     07fffh  ; Stack Pointer Address
8000            uart            equ     08000h  ; UART for RS232C
9000            f_f             equ     09000h  ; Int. Clear F/F
A000            timer           equ     0a000h  ; 1msec Timer
B000            pia             equ     0b000h  ; Relay,Buzzer,LED Port
C000            sw              equ     0c000h  ; ON/OFF SW Port
D000            sensor          equ     0d000h  ; Temp. Sensor Port

4000            uart_flg        equ     work+0  ; UART Interrupt Flag
4001            timer_flg       equ     work+1  ; Timer Interrupt Flag
4002            temp_data       equ     work+2  ; Temp. Data Buffer
4003            sec_event       equ     work+3  ; 1Sec Event Flag

0000                    org     0000h           ; Reset Vector
0000            start:
0000 31FF7F             ld      sp,stack        ; Stack Pointer Set
0003 F3                 di                      ; Disable Interrupt
0004 CD0001             call    initialize      ; System Initialize
0007 C33301             jp      main_loop       ; Let's go !

0100                    org     0100h
0100            initialize:
0100 210040             ld      hl,work
0103            ram_clear:
0103 3600               ld      (hl),0          ; RAM Area Clear
0105 2C                 inc     l
0106 C20301             jp      nz,ram_clear    ; 256 Bytes
0109 2103A0             ld      hl,timer+3      ; Timer Initialize
010C 3636               ld      (hl),00110110b  ; Command Set
010E 2100A0             ld      hl,timer        ; Timer #0 Data Set
0111 3664               ld      (hl),100
0113 3628               ld      (hl),40
0115 2103B0             ld      hl,pia+3        ; PIA Initialize
0118 3682               ld      (hl),10000010b  ; Command Set
011A 2100B0             ld      hl,pia          ; Port #0 Data Set
011D 3601               ld      (hl),00000001b  ; Relay ON
011F 23                 inc     hl              ; Port #1 Data Set
0120 3600               ld      (hl),00000000b  ; Buzzer OFF
0122 23                 inc     hl              ; Port #2 Data Set
0123 36FF               ld      (hl),11111111b  ; LED OFF
0125 210180             ld      hl,uart+1       ; UART Initialize
0128 364E               ld      (hl),01001110b  ; Mode Set
012A 3611               ld      (hl),00010001b  ; Command Set
012C 210090             ld      hl,f_f          ; Int. F/F Clear
012F 3600               ld      (hl),0
0131 FB                 ei                      ; Enable Interrupt
0132 C9                 ret
0133            main_loop:
0133 CD4501             call    sw_check        ; SW Status Check
0136 CD6A01             call    led_display     ; LED Output
0139 CD7901             call    temp_input      ; Sensor Check
013C CD8F01             call    uart_send       ; RS232C Transmit
013F CD9C01             call    timer_check     ; 1Sec / Temp.Check
0142 C33301             jp      main_loop

0145            sw_check:
0145 2100C0             ld      hl,sw           ; SW Port Address
0148 7E                 ld      a,(hl)          ; Data Input
0149 E601               and     00000001b       ; SW Event ? (Active Low)
014B C0                 ret     nz              ;  NO --> Return
014C            power_off:
014C 2100B0             ld      hl,pia          ; Relay Port Address Set
014F 3600               ld      (hl),00000000b  ; Relay OFF
0151 23                 inc     hl
0152 23                 inc     hl              ; LED Port Address Set
0153 36FF               ld      (hl),00000000b  ; LED All ON !
0155            sw_loop1:
0155 2100C0             ld      hl,sw           ; SW Port Address Set
0158 7E                 ld      a,(hl)          ; Data Input
0159 E601               and     00000001b       ; SW OFF ?
015B CA5501             jp      z,sw_loop1      ; Waiting
015E            sw_loop2:
015E 2100C0             ld      hl,sw           ; SW Port Address Set
0161 7E                 ld      a,(hl)          ; Data Input
0162 E601               and     00000001b       ; More SW Event ?
0164 C25E01             jp      nz,sw_loop2     ; Waiting
0167 C30000             jp      start           ; Return to Start
```

〔リスト3.2〕
LED 表示/温度
入力ルーチンの例

```
016A            led_display:
016A 3A0340             ld      a,(sec_event)     ; 1Sec Event Flag Check
016D E601               and     00000001b         ; Event ?
016F C8                 ret     z                 ;  NO --> Return
0170 3A0240             ld      a,(temp_data)     ; Get Temp Data
0173 EEFF               xor     0ffh              ; All Bits Reverse
0175 3202B0             ld      (pia+2),a         ; LED Display
0178 C9                 ret

0179            temp_input:
0179 2100D0             ld      hl,sensor         ; Temp. Sensor Address Set
017C 7E                 ld      a,(hl)            ; 1st Data Input
017D 47                 ld      b,a               ; Stock
017E 2100D0             ld      hl,sensor
0181 7E                 ld      a,(hl)            ; 2nd Data Input
0182 B8                 cp      b                 ; (1st)=(2nd) ?
0183 C27901             jp      nz,temp_input     ;  NO --> Try Again !
0186 21B401             ld      hl,temp_table     ; Temp. Transform Table
0189 6F                 ld      l,a               ;    Address Set
018A 7E                 ld      a,(hl)            ; Get Correct Data
018B 320240             ld      (temp_data),a     ; Put Data to Buffer
018E C9                 ret
```

〔リスト3.3〕ホスト・パソコン側 RS-232-C 表示プログラムの例

```
#include <stdio.h>

#define ESC       0x1b /* Key [ESC] */
#define ESCAPE    1111

char ah,al,bh,bl,ch,cl,dh,dl;
char c;
int result;
unsigned ax,bx,cx,dx;
extern unsigned _rax,_rbx,_rcx,_rdx;

main(argc,argv)
        int argc;  char *argv[];
{
        cursor_off();
        screen_clear();
        while(result!=ESCAPE){
                keyboard_check();/* Main Loop */
        }
        screen_clear();
        cursor_on();
}

keyboard_check(){
        if(rs232c()==0){ /* Non RS232C Event , */
                c=csts();/* Keyboard Check !   */
                if(c==ESC) result=ESCAPE;
                else result=0;
        }
        else{       /* RS232C Data Display */
                c=rx_rs232c();
                printf(" %2x ",c);
        }
}

cursor_on(){
        printf("%c[>5l",ESC);
          /*  bios(0x1100,0,0,0,0x18);  */
}

cursor_off(){
        printf("%c[>5h",ESC);
          /*  bios(0x1200,0,0,0,0x18);  */
}
```

```
rs232c(){               /* In-Line Assemble !! */
#asm
        mov     ah,02h
        int     19h         ; RS232C BIOS
        mov     word cx_,cx ; Port Status --> CX
#
        return(cx);
}

rx_rs232c(){            /* In-Line Assemble !! */
#asm
        mov     ah,04h
        int     19h         ; RS232C BIOS
        mov     byte ch_,ch ; Get Data --> CH
#
        return(ch);
}

screen_clear(){         /* In-Line Assemble !! */
        for(bx=0;bx<32000;bx+=2){
#asm
        push    es
        mov     ax,0a800h   ; VRAM Area Segment
        mov     es,ax       ;                Address
        mov     ax,0        ; Clear Data = [0000]
        mov     bx,word bx_ ; Offset Address <-- BX
        mov     es:[bx],ax
        pop     es
#
        }
}

bios(p1,p2,p3,p4,p5)
        unsigned p1,p2,p3,p4,p5;
{                       /* DOS Call Function */
        _rax=p1; _rbx=p2; _rcx=p3; _rdx=p4;
        _doint(p5);
        return(_rax);
}
```

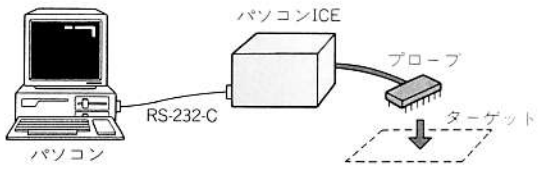
〔リスト3.4〕
割込みルーチン
/RS-232-C
ルーチンの例

```
0038                    org     0038h           ; INT Vector Address
0038 3E01               ld      a,1
003A 320040             ld      (uart_flg),a    ; Set UART Interrupt Flag
003D 210090             ld      hl,f_f
0040 3600               ld      (hl),0          ; F/F Clear
0042 ED4D               reti

0066                    org     0066h           ; NMI Vector Address
0066 3E01               ld      a,1
0068 320140             ld      (timer_flg),a   ; Set Timer Interrrupt Flag
006B ED45               retn

018F            uart_send:
018F 3A0340             ld      a,(sec_event)   ; 1Sec Event Flag Check
0192 E601               and     00000001b       ; Event ?
0194 C8                 ret     z               ;  NO --> Return
0195 3A0240             ld      a,(temp_data)   ; Get Temp Data
0198 320080             ld      (uart),a        ; Transmit to RS232C
019B C9                 ret
```

パソコンICE

最近のCPUプログラム開発環境で目立つのが，パソコンを開発ツールのホストとしたシステムです．じつは筆者も気に入っていて，条件が許せばこの環境を使う場合が多くなっています．これは，プログラム開発・アセンブル・リンクまでをホストのパソコン上で実行し，機械語プログラムをRS-232-CでICEツールへ転送するというものです．デバッグは，ホストのリモート・デバッガというツールによって，RS-232-C経由でICEを制御して実行します(**図3.8**)．この方法だとICEの部分だけが開発ツールのハードウェアとなるので，オールイン・ワン型の装置よりもはるかに低価格・コンパクトとなります．そして，さらにソフト面での，つぎの二つの大きなメリットが人気の秘密なのです(**図3.9**)．

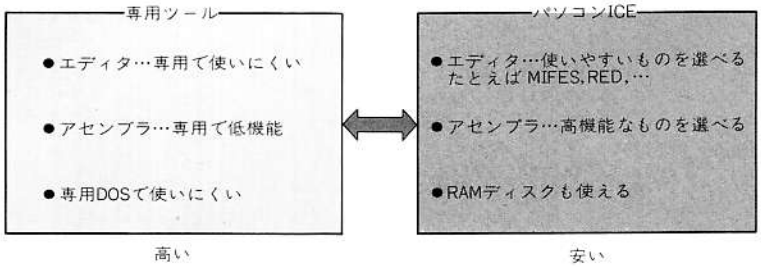
〔図3.8〕パソコンをホストにした開発システム



その第1は，**ソース・プログラムのエディタとして，自分の使い慣れた，機能の高い汎用エディタを使える**点です．高級言語でなくアセンブラだからこそ，マクロ機能やモジュール化のテクニックを駆使するので，同時に複数のファイルを編集したり，カット&ペースト機能の有効性は大きくなります．開発作業に占める時間から考えても，カーソルの移動やスクロールのスピードは重要な効率要因となるのです(ちなみにこの原稿は某有名ワープロで書いていますが，日頃使い慣れたエディタのスピードに比べて，ワープロの反応の遅さというのはストレスものです)．

そして第2に，**機能の強力な市販のアセンブラを自由に選べる**，という点です．CPUメーカ，開発ツール・メーカの専用アセンブラには制限や限界が多く，各ソフト・ハウスの提供するアセンブラとは比較になりません(**リスト3.5**)．筆者の愛用するアセンブラの場合，

〔図3.9〕
パソコンICEのメリット



〔リスト3.5〕専用アセンブラによるソース・プログラム例

```
DTCHK:                          ; DATA CHECK SEQUENCE
        LDA     DATA+1          ; RX DATA
        BPL     DTLOW
        STA     DATA+2          ; DATA STOCK
        RTS
DTLOW:                          ; DATA < 128
        STA     DATA+3          ; DATA BUFFER
        AND     #0F0H
        BEQ     D000            ; 00-0F
        CMP     #010H
        BEQ     D010            ; 10-1F
        CMP     #020H
        BEQ     D020            ; 20-2F
        CMP     #030H
        BEQ     D030            ; 30-3F
        LDA     #00H
        STA     DATA+4          ; SEND PARAMETER
        BRA     DTEXT
D000:
        LDA     #4
        STA     DATA+4          ; SEND PARAMETER
        BRA     DTEXT
D010:
        LDA     #7
        STA     DATA+4          ; SEND PARAMETER
        BRA     DTEXT
D020:
        LDA     #5
        STA     DATA+4          ; SEND PARAMETER
        BRA     DTEXT
D030:
        LDA     #6
        STA     DATA+4          ; SEND PARAMETER
DTEXT:
        JSR     TXDAT           ; RESULT DATA SEND
        RTS
```

(ラベルの文字数に制限がある)

〔リスト3.6〕汎用クロス・アセンブラによるソース・プログラム例

```
data_check_sequence:
        lda     rx_data
        bpl     data_under_128
        sta     data_stock
        rts
data_under_128:
        sta     data_buffer
        and     #0f0h
        &cp_eq  00h,rx_data_00_0f
        &cp_eq  10h,rx_data_10_1f
        &cp_eq  20h,rx_data_20_2f
        &cp_eq  30h,rx_data_30_3f
        &move   0,send_parameter
        bra     data_check_exit
rx_data_00_0f:
        &move   4,send_parameter
        bra     data_check_exit
rx_data_10_1f:
        &move   7,send_parameter
        bra     data_check_exit
rx_data_20_2f:
        &move   5,send_parameter
        bra     data_check_exit
rx_data_30_3f:
        &move   6,send_parameter
data_check_exit:
        jsr     result_data_send
        rts
```

(&で始まるコマンドはマクロ定義命令)

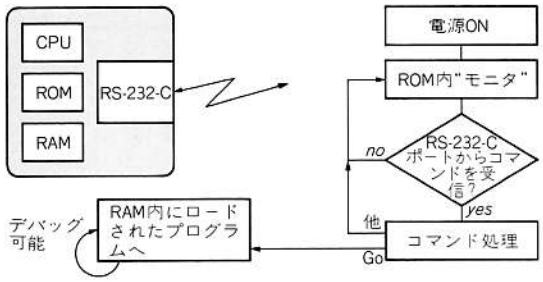
強力なマクロ機能や他CPUとのクロス化を意識した共通体系，という以上に，ラベル文字数の制限をひとつ指摘するだけで十分の理由になります．ラベルに何10文字も使えれば，アンダ・バーでつないで，各ラベル自身がその意味を完全に説明するようにプログラムできます(**リスト3.6**)．これによりほとんどの注釈行は不要となり，エディタのコピー機能によって2回目以降のキー入力も不要です．これは経験してみないとわかりにくい感覚ですが，プログラミング環境，という心理的な要因として，大きなプログラムでは完成度にも影響してくるほどのものです．

また最近は，アセンブラと互換性のある，混在可能なCコンパイラなども出ているので，選択の可能性はますます広がっています．逆に，市販のクロス・アセンブラの欠点としては，機能が豊富な分，アセンブルが非常に遅いものがあります．RAMディスクに移しても何分も返ってこない，というのは，精神衛生上あまり歓迎できません．また，プログラムが大きくなると，RS-232-Cを介しての転送の時間がやや気になってきます(p.211の**Appendix 3.2「アセンブル/ダウンロードのスピード比較」**参照)．この点，専用の通信ボードをパソコンに挿入して，専用のパラレル・バスで転送するタイプのシステムは快適です．

デバッグ・モニタ

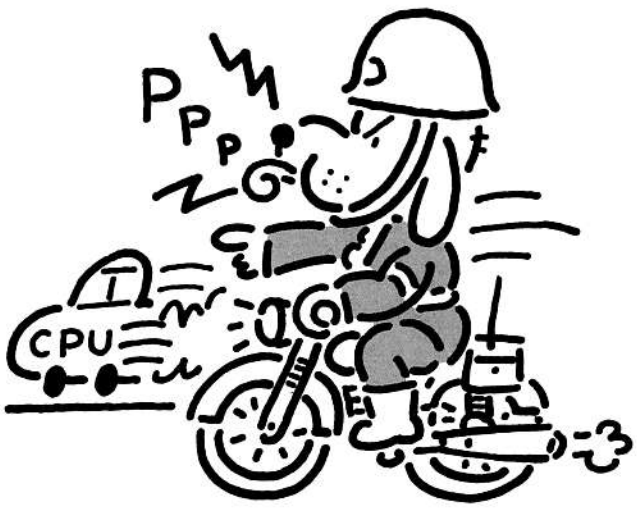
開発ツール+ICEによってCPUのプログラムを開発するタイプ以外にも，ボード・マイコンのソフト開発環境はまだあります．むしろこちらのほうが歴史は古く，特別なハードも必要としないので，アマチュア的ともいえます．これは，ボード上に完成品のROMが載っているもので，ここには**モニタ・プログラム**が入っています．このボードの電源がオンになると，CPUはモニタという名のプログラムによって走ります(**図3.10**)．モニタはRS-232-Cポートからの通信によって，外部機器からのコマンドに対応します．コマンドには，メモリやポートを読み書きしたり，特定のアドレスから実行させたり，システムの情報を聞き出すものがあり，これによってデバッグが実行できます．また，CPUのRAMエリアにプログラムを外部からロ

〔図3.10〕デバッグ・モニタROM付きボード・マイコン



ードするコマンドによって，モニタからユーザのプログラムに移行できます．これは小規模なマイコン・システムの場合には，開発ツールさえ不要な，身軽な開発環境として有効なものです．ただし，よりCPUのハードに近づいた位置にあるので，かなり機械語に近いレベルで対応する必要があります．

これ以外にも，筆者は使った経験がないので詳しく触れませんが，Basicインタプリタを ROM に載せたボード・マイコン(や1チップ・マイコン)というものがあり，インタプリタの低速さとCPUの高速さとがどう折り合うのか，興味があります．それから，拡張ディスク・コントローラ基板を使用する前提で，ROMの形でDOSを搭載したボード・マイコンもあります．おそらくこの分野は，今後はメモリ・カードのインターフェースをもち，カードによってファームウェアを自由に選択できる，というような形に発展すると思われます(**コラム3.3**)．



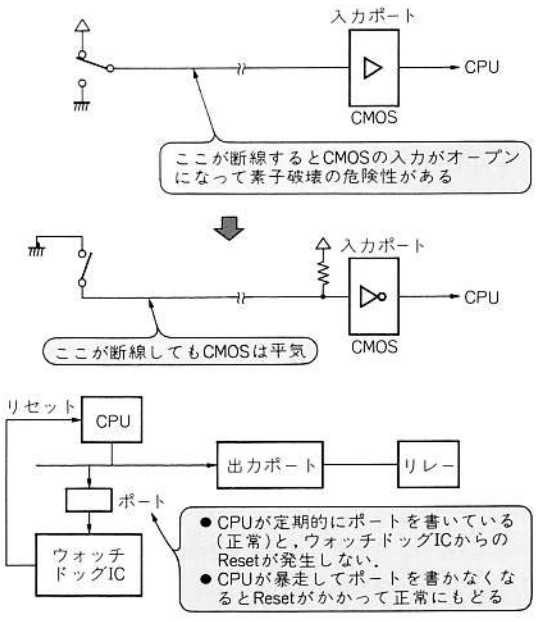
信頼性設計のポイント

ここまで述べてきたような，汎用のボード・マイコンを組み合わせたシステムと，パソコンや計測器などの量産された製品との差は，具体的にはシステムの完成度や信頼性の違いとなって，明確に現れてきます．たとえば，果物の個数をカウントするシステムの誤動作は，データのばらつきという問題ですみますが，ファンヒータや自動車の制御システムの暴走は，そのまま人命にかかわります．また，人工衛星の制御プログラムが誤動作しても，デバッグどころかリセットすらできません．あるいは，核兵器を格納する設備のコントロール・システムなどは，いったいどういう神経があれば開発できるのか，筆者には想像もつきません．

このように，信頼性の基準や重要度はケースバイケースなのですが，多少の注意で効果が上がる部分については，心がけてシステムを設計・開発したいものです．よくいわれるように，外界とインターフェースするシステムの考え方の基本は，**フェイルセーフ**の発想にあります．そして，CPUの暴走については，〈起こさない〉〈起きた場合のフォロー〉という2面から検討すべきでしょう．

フェイルセーフなシステムにするためには，CPUが

〔図3.11〕フェイルセーフなシステムの例



暴走したり，信号線の一部が断線した場合を想定して，なるべく被害が小さくなる方向にシステムが移るように考えます(**図3.11**)．リレーが入りっぱなしで過熱しないような接続とか，入力ポート信号のプルアップなどがこれにあたります．また，ウォッチドッグ・タイマICでCPUの暴走を監視したり，リセットICによって，電源変動に対して安定な初期リセットをかけます．ソフト的には，プログラムROMの未使用エリアやプログラム中のあちこちに，エラー・ルーチンへのジャンプ命令を置き，万一の暴走の際にはI/Oポートを安全に処置してから，異常の発生をディスプレイするようにします(**リスト3.7**)．

問題を起こしにくいシステムという視点では，まず，外界からの影響に対する耐性として，

① 電源の瞬間停電
② 電源の電圧低下
③ 電源ラインのノイズ
④ 静電気

などによるCPUの暴走を防ぐような対策が必要になります(**図3.12**)．これはボード・マイコンや電源のハード自体の問題ですが，システム屋にできる部分も多くあります．たとえば，スイッチング電源とレギュレータ電源とでは，上のノイズの影響が異なるので，比較して実験してみる意味があります．また，ボード・マイコンの電源よりも高い電圧の電源ユニットを使って，CPUボード上にレギュレータICを一つ載せるだけで，驚くほど瞬停，ノイズ・マージンが向上したことがあります(**図3.13**)．あるいは，バス・ラインの末

〔リスト3.7〕ソフト的な暴走対策の例

```
        org     0000h
start:
        ld      sp,stack
        di
        call    initialize
        jp      main_loop
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        org     0038h
int:
         |
         |(略)
         |
        reti
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        org     0066h
nmi:
         |
         |(略)
         |
        retn
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        org     0100h
main_loop:
        call    job_number_1
        call    job_number_2
        call    job_number_3
        jp      main_loop
        jp      error           ; Error Trap !

initialize:
         |
         |(略)
         |
        ret
        jp      error           ; Error Trap !

job_number_1:
         |
         |(略)
         |
        ret
        jp      error           ; Error Trap !
job_number_2:
         |
         |(略)
         |
        ret
        jp      error           ; Error Trap !

job_number_3:
         |
         |(略)
         |
        ret
        jp      error           ; Error Trap !

data_bank_area:
        db      '0123456789ABCDEF'
        jp      error           ; Error Trap !

error:
        call    io_off          ┐
        call    led_all_off     │
        call    wait_1sec_timer │ エラー・メッセージ表示
        call    led_all_on      │
        call    wait_1sec_timer │
        jp      error           ┘

        org     1000h
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        org     2000h
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        org     3000h
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !
        jp      error           ; Error Trap !

        end
```

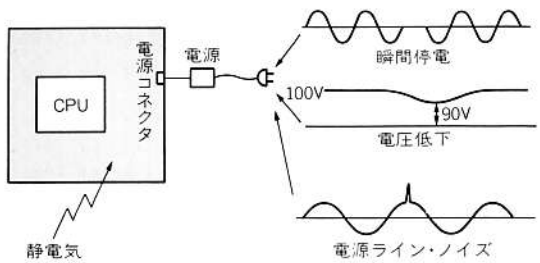
端処理とか，電源の引き回しとか，CPU ボードのパスコンの追加とか，周辺ボードから外部信号への接続の部分に，ノイズ対策用のコンデンサやフェライト・ビーズを用いる，といった簡単な対策によって，不安定だった動作が嘘のように落ち着いたこともあります．

また，逆に外部の機器に悪影響をもたらすノイズを出さない，という視点も重要です．FCC*などの規格をクリアする，という現実的な問題だけでなく，**ノイズを出すものにはノイズが入りやすい**，という明らかな事実があるからです．これらの信頼性対策は，ボード・マイコンのシステムであっても，完成度の高いレベルに限りなく近づく，という意味で，地味ながら重要なシステム設計要素であると思います．

ソフトウェアの信頼性・自己診断

事故やノイズ的な誤動作に対する信頼性設計とならんで，ソフトウェア自身の信頼性，あるいは製品全体(ユーザにとって製品のソフト的印象)の信頼性，というものの向上もまた，マイコン技術者の領域となります．

〔図3.12〕CPU を誤動作させる原因



ここでは，

①「仕様的なバグ」のないソフトウェア

② 仕様に対してバグのないソフトウェアとするためのデバッグ手法

③ 量産時に設計仕様との確認を行う〈自己診断〉プログラム

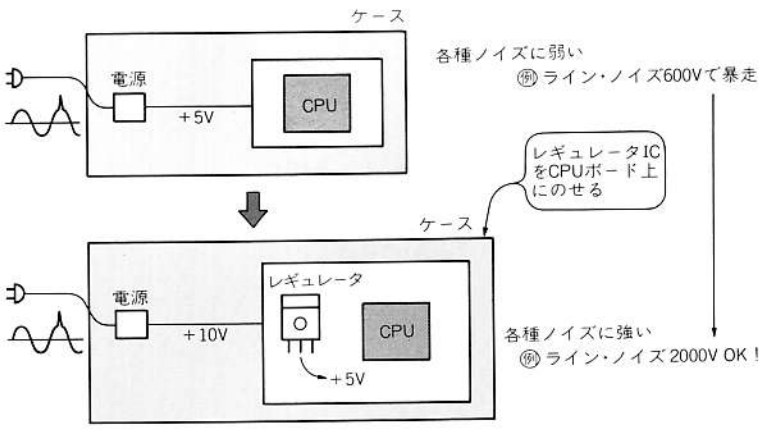
という三つの視点で，ソフト的な信頼性設計について考えてみましょう．

第1の〈**仕様的なバグ**〉というのは，たとえばプログラムの全体仕様から各ルーチンのモジュール仕様書までを開発担当者が設計して，この仕様にしたがった具体的なプログラム開発を外注する場合に，その外注先が「仕様書に対して100％完全な仕事をした」のに発生したトラブル，というものです．つまり，ソフト的なシステム設計そのもののバグのことです．

具体的な例をあげましょう．製品のパネルのあちこちのスイッチをスキャンして個々に対応する処理をするシステムで，まさかこれだけ離れた三つのスイッチを人間の手は押せるはずがない，として考えもしなかった組合せのスイッチが同時に押されると，思わぬ動作をしてしまう，というケースがありました．近くにある一群のスイッチごとにブロックに分けてスキャンして，その中では，二つ以上が押された場合も対応していたのですが，1mほど離れた別のブロックの処理はメイン・ルーチンから呼ばれるルーチンを変えていたために，非常に特殊なケースでは競合してしまったのです．このような仕様そのもののバグへの対策としては，

① 全体を構造化して，各部分(モジュール)のインターフェースも明確に仕様書化して確認する．

〔図3.13〕
ノイズ・マージンが向上した例



② 入力情報のあらゆる組合せを想定して(境界値分析など),「思わぬ操作方法」というアナをなくす.
③ 場合によってはソフト上で仕様モデルをプログラムして,入力パターンのすべての可能性に対するふるまいをシミュレーションしてみる.
④ 例外処理をあちこちに散在させないで,なるべく1本にまとめる.
⑤ 割込みがつねに多重に最悪のタイミングで起きつづけている,という条件下で通常ルーチンの動作を検討する.

などが考えられます.

第2の〈**デバッグ手法**〉というのは,ポートにCPUが書いているはずの信号が出てこない,といったあからさまなバグに対するものではありません.一応,仕様書どおりのプログラムが完成した,という時点で,「本当にどうやっても誤動作しないか」をチェックする,という完成度確認または完成度追求というものです.パソコン・ソフトなどの場合,一部にバグがあっても,登録ユーザにバージョンアップ・ソフトを送ればすんでしまうのですが,組込み機器のCPUソフトはROM化されているので,バージョンアップはできません.交換のEPROMを送ってケースを開けて交換してくれ,という話もできませんし,まして1チップCPUのマスクROMプログラムであれば,もう基板交換しかありません.最悪なら製品交換となってしまいます.

このようなデバッグのポイントの一つは,「**開発担当者がデバッグしない**」ということです.ユーザになったつもりで,いろいろでたらめに操作しているようでも,本人のクセがあり,案外ワンパターンの組合せしかチェックしていないものです.中身を知らない素人のほうが,よっぽど異常な組合せで暴走させてくれます.もう一つは,「営業」のようなマニュアルを理解できる人に,内部の事情をいっさい知らせずに,マニュアル(仕様書)だけを使って,とことん操作してもらうことです.開発担当者が自分ひとりで「明らか」と思い込んでいたチョンボや,想像もしなかった可能性を指摘してくれるものです.この意味では,ショップの店員さん,学校の先生,あるいは海外支社の外国人スタッフ,といったなるべく別種の人にいじってもらうのが,経験的にもきわめて有効です.

よくあるケースとして,この段階でおおもとの仕様書の仕様の不完全性が明確になる場合があります.この場合,小手先でごまかそうとせずに,時間の許すかぎり,上流にさかのぼって再検討するほうが,結局,信頼性向上に役立ちます.

第3の〈**自己診断プログラム**〉というのは,量産される製品の場合,すべてのハードウェアが外見上完成した段階で,チェッカにいちいちかける手間をなるべく省くものです.マイコン応用製品の場合,ハードウェアはいわばCPUの手足であるわけですから,CPUが自分で手足をあちこちとチェックする,というのは容易なことです.

生産工場での効率を考えると,なるべくチェックの手間を少なくすることは,トータル・コストにも大きく貢献します.生産ラインの出口近くのテスト部分で,いちいちチェック用基板を挿入する,とかいうのではなく,よくある方法で,「ある特別の組合せでパネル・

スイッチを押しながら電源ONする」というチェック・ルーチン・エントリ方法が便利です(いわば裏モードです)．たとえば，温度アップと温度ダウンとか，スタートとストップといった，普通は同時に押さない組合せを押しつづけながらパワーオンする，というふうにします．また，RS-232-Cコネクタに，TX DATAをそのままRX DATAに戻すコネクタを差し込みながら上のことをする，というような手もあります．CPUはこのルーチンの冒頭で，RS-232-Cからある暗号パターンを送信し，そのまま受信されるのを確認することで，このシリアル・ポートのハードウェア確認もしてしまえるわけです．

CPUはこのチェック・ルーチンでは，

① RAMのチェック
- RAMがささっているか
- データの読み書きができるか(ところどころのアドレスで，いくつかのデータのパターンで実行)

② ROMのチェック
- 特定アドレスのバージョン番号を確認

③ 周辺LSIのチェック
- できるものは割込み対応なども

④ キー・スキャン，ディスプレイ・デバイスのチェック

などを実行し，OKなら所定のLEDを所定パターンで点滅するなどして知らせます．1秒もあれば相当のチェックができます．

製品がフィールドに出てしまってからのクレームに対応する診断方法，というのも重要です．最近の日本では，とくに数万円以下の電化製品はたんなる消耗品になっているので，故障してもまた買えばいい，という異常な時世になっています．これはメーカにとっては思うつぼでもあるのですが，やはりシステム屋としては，フィールドでのクレームに対して，いちいちシンクロをもって駆けつけなくてもすむような手だてを組み込んでおきたいものです．

RS-232-C端子のついた装置のようなものであれば，正規のモードのほかに，ある特殊(裏)モードで立ち上げると，RS-232-Cからパソコンにつないで，じつは内部をテストできるモニタが走る，などというのは美しいと思います．

また，ROMソケットにEPROMを挿入している製品であれば，テスト・プログラムの入ったEPROMをサービスマン用に作っておく，という方法もあります．

昔の電化製品はテレビでもステレオでも，電気屋さんがテスタとオシロで修理できたのですが，ディジタル化された現在では，どこにプローブをあてても，まったく様子がつかめません．高額な装置や輸出されるものでは，現地デバッグ用の治具としてテスト用ボードを製作することもあります．ボードから出たコネクタを製品の所定の場所に接続すると，オシロで波形が見られるようにD-A*変換する，といったテスト・ボードもあります．

このように，大型機ほどの本格的なものでなくても，マイコン応用システムでも自己診断機能は信頼性向上の一つの武器となるのです．

Appendix 3.1

新CPU基板が動くまで

実験用の新しいCPU基板を製作したり，購入したCPU基板を改造したとき，いざ新たに立ち上げようという瞬間はいつでもワクワクするものです．何度やっても単純なミスの一つや二つはなくならないものですが，ここではプリント基板/手配線基板/購入基板改造，のいずれにも共通した筆者の体験的「立上げプロセス」を紹介します．

1 導通テスタによるチェック

テスタの抵抗レンジか，できればディジタル・テスタの導通ブザーを使った最初のチェックは，まだ部品が一つものらない基板のうちの配線チェックです．テスタの導通ブザーはICが挿入されていると，ゼロ・オームでなくても鳴ってしまうので，いちばん最初が肝心です．プリント基板の試作屋のものであっても，たとえばデータ・バスの番号がさかさまになったり，という種類のミスがあるので，以下のような点を調べます．

(1) 電源ライン～GND

まさか，と思うこのショートは，パターンとして結ばれていなくても，手配線や手貼りのパターンでは「くっついてしまう」ことがあります．

(2) データ・バス：D_0〜D_0，D_7〜D_7同士などの対応

筆者の場合，面倒くさいので全ピンは調べません．テスタの電極の一方をCPUのD_0につけて，もう一方をROMやRAMや周辺LSIのD_0につけて導通を確認する，というものです．アドレス・バスもA_0とA_7とA_{15}，とかの何箇所かで十分です．要は，信号名がひっくり返る($D_0 \longleftrightarrow D_7$など)というミスのチェックです．

(3) バス・ライン内のショート・チェック

これは大型のCADで自動作成された基板で起きたミスですが，CAD内でデータ・バスのD_3とD_4が区別されずにあちこちで「接続」されてしまっていた，というようなミスのチェックです．調べるのは簡単で，まずテスタの電極の一方をD_0につけて，もう一方をD_1〜D_7にタタタ…とすべらせて「ピー」と鳴るかどうか，と試せばいいのです．つぎはD_1，D_2，…とやっても1分間もかかりません．これは，IC各種が挿入されてからでは捜索が非常に困難なチェックです(データ・バス内の2本がショートしていると，動作中にオシロで見ると+5VとGNDの中間あたりの電位が見られるので，なれれば容易に判定できる)．

(4) メモリまわりのチェック

使用しない$\overline{OE}$端子がGNDに接続されているか，あるいはEPROMのV_{pp}端子が+5Vになっているか，さらにRAMの$\overline{WE}$とCPUの$\overline{WR}$との配線のみちすじなどもチェックしておきます．

2 IC類以外のハンダ付け

ICソケット(おそらく実験用ボードなら，ICのすべてがソケットでしょう)やパスコン，プルアップ抵抗やLEDなどの部品をとりつけます．基板が両面ぐらいまでならば，あとでミスを発見してもパターン・カットや接続ができますが，4層とかになると，ICソケットの下に隠れたパターンの修正は不可能になってきます．この点で，ソケットをハンダ付けした時点で，「サイは振られた」ことになります．

3 いったん電源をつないでチェック

IC類のない状態で，いったん電源をつなぎます．ここでのツールはテスタの電圧レンジか，オシロになります．オシロも多チャネルや高速のものの希望は限りがありませんが，筆者は2チャネルもあればなんとかできないか，とむしろ手抜きを考えるようにしています．

(1) 電源電圧のチェック

すでにショート・チェックをしているので，ここでレギュレータから煙が出ることもないでしょうが，一応，ICがない状態で4.9Vから5.1Vあたりにあることをチェックします．

(2) 電源表示用LED

のちのちICEでデバッグをする際，基板に電源がきていることが一目でわかるLEDが一つついているだけで安心します．筆者の基板には必ず入れてあります．

(3) 各ICの電源/GND端子の電圧チェック

シンクロのアース端子(ワニグチ)をGNDにしておけば，1分間ぐらいのチェックです．CPUを含む各IC/LSIのソケットの，電源ピンとGNDピンの電圧をチェックします．CMOSの石でも，さかさまでは煙が出るときがあるので，この一応のチェックは大切でしょう．

4 ROM・RAM・デコーダなどを挿入する

いきなりすべての半導体部品をつけてしまって，そこで問題があると結局あともどりするので，筆者はまずはメモリとデコーダ関係を挿入することにしています．デコーダ関係とは各種周辺LSIの$\overline{CS}$や$\overline{WR}$，$\overline{RD}$などの信号をつくっている部分で，HS-CMOSだったりPALだったりします．〈138〉を挿したのにインバータの〈04〉を忘れていて信号が出ない，などということもあるので，ここでは周辺LSIを除く「小物」をすべて挿してしまってもいいでしょう．

ここで挿入するROMにはまだプログラムが入っていないのですが，バス・ファイトのチェックのために，何もプログラムされていないEPROMをダミーとしてソケットに挿入しておきます．

5 CPUソケットにICEを挿入する

いよいよICEの登場です．再び基板の電源をオンにして，ICEを立ち上げてみます．この状態で，ターゲット・システムの電源が正常であることを示すICEのインジケータがONになっているはずです．

この段階ではメモリだけなので，以下のようなチェックをします．

(1) ICEのメモリ・エリア割付けを指定

まずはICEのメモリ・エリア割付けを変更します．ICEには内部のRAM(ふつう「システム・エリア」と呼ばれる)と，ターゲット・ボード上のエリア(ふつう「ユーザ・エリア」と呼ばれる)があり，ICEが立ち上がった直後は全メモリ空間がシステム・エリアとなっている場合が多いので，このままでは外部の基板上に信号が出てきません．そこで，割当てコマンドによって，プログラムをロードするメモリ空間(EPROMと同アドレス)の部分をシステムにしたまま，残りをユーザに指定してやります．

(2) ダンプ・コマンドでバスを見る

データ・バスにプルアップ抵抗がついているシステムであれば，なにも割り当てられていないアドレス部分をダンプさせると，"FF"という，オール"H"の表示が見られるはずです．ここでもし，D_0をGNDにショートさせれば"FE"とか，D_7をGNDにショートさせると"7F"とかになり，ICEを経てデータ・バス上の信号が確認で

きます.

(3) RAMのデータ読み書きチェック

RAMに割り当てられたメモリ・エリアをダンプしてみると，最初ならばでたらめの値が表示されます．ここに“Fill Up”コマンドで，たとえばオール“00”を書き込みます．この時点で正しく書き込めないと，ICEはエラーを伝えてきます．その場合，$\overline{WR}$信号の経路か$\overline{CS}$のところをチェックします．OKであれば，データとして“00”，“FF”，“55”，“AA”あたりの値で，再びダンプしてみてデータが正しく埋まっていれば，もう大丈夫です．ダミーで挿入したEPROMが何も悪さをしなければ，逆の意味でEPROMの$\overline{CS}$回路が正しい証拠ともなります．

(4) EPROMのダミー・データ・チェック

ROMライタには特定のエリアを特定のデータで埋める機能があるので，ここでテスト用に1回，ダミーEPROMを作ることもあります．

ICEのメモリ・エリア割付けで，ROMエリアをユーザにして，ダンプ・コマンドでデータを見ます．ROMライタで焼いたとおりのデータであれば，やがてプログラムを書き込んだときに，安心してソケットに挿入できます．

6 周辺LSIのコントロール信号のチェック

ICEのアセンブル機能かデータ変更機能を使うと，ごく簡単なプログラムはデバッガ上で直接打ち込めます．ここではデコーダ関係について，さらに周辺LSIの部分をチェックします．どのLSIについても同様で，それぞれつぎのようにします．

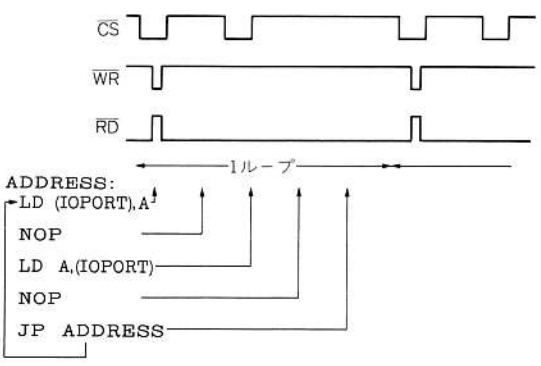
(1) ICE上のテスト・プログラム製作

ICEのメモリ・エリア割付けでシステムとした範囲に，テスト用のプログラム(**図A**)を作ります．CPUによってコードは異なりますが，

〈1行目〉I/Oにデータを書きこむ
〈2行目〉NOP
〈3行目〉I/Oからデータを読む
〈4行目〉NOP
〈5行目〉1行目に戻る

というようなものです．CPUによっては1行目と3行目が2ステップになるかもしれません．

〔図A〕ICEテスト用プログラムと動作のようす



(2) このプログラムを走らせる

これは無限ループなので，異常がなければずっと続けるはずです．そして2行目と4行目は同じNOPでも，5行目から先頭に戻る時間があるので，1行目と3行目の$\overline{CS}$の間隔は異なることになります．

ここではオシロが活躍します．オシロの第1チャネルを，I/Oの$\overline{CS}$につないで同期をとると，**図A**のように2発パルスのループとして走ります．そこで第2チャネルで$\overline{WR}$信号を見ると$\overline{CS}$の1発目の中で下がっていて，ライト命令となるのがわかります．$\overline{RD}$のほうは，プログラムのフェッチでもアクティブになるので，ここではおまけ程度です．

(3) データ・バスのチェック

ライト命令の1行目の書込みデータをいろいろ変えてやると，$\overline{CS}$がアクティブのときのデータ・バスの各ビットの値をチェックできます．このとき，“00”とか“FF”でなく，“55”や“AA”のようなとなりあったビットが反対のものを使うと，データ・バスのショートを調べられます．

(4) アドレス・バスのチェック

周辺LSIでは，アドレスのA_0やA_1を使って，内部のレジスタを指定するものがほとんどです．そこで，1行目や3行目のI/Oアドレスを変えてやると，$\overline{CS}$のときにA_0が“H”(アドレスのLSB＝1)とか，A_0とA_1の組合せなどをテストできます．

これらのチェックで，もう周辺LSIも基本的には安心して取り付けられます．

7 全部品のとりつけ

周辺LSIなどをすべてソケットに挿入し，もしフラット・パッケージで直接ハンダ付けするものがあれば，ここに来ては運を天にまかせて，とりつけてしまいます．24ピンや28ピンのミニフラットならともかく，80ピンや100ピンともなると，ミスがあっても取り去るのは非常に困難なのですが，ここまでのチェックがOKの基板であれば，もう進むしかありません．この時点で，ハードは外見上完成してしまいますから，ここからはむしろ「確認」となってほしいものです(ミスがあったら，しっかり腹をくくって戻るのみ)．

8 テスト・プログラムを走らせる

今度は，ホスト上でテスト用のプログラムをソースから組んで，アセンブル，ダウンロードによって正式に走らせます．このテストは，以後のソフト開発/デバッグ作業の流れの，試運転というわけです．

筆者がこの1回目にたいてい使うのは，周辺LSIを出

力ポートに使った，LEDの点滅ルーチンです．

8255のようなPIAはもちろん，8251のようなUARTであっても，$\overline{\text{RTS}}$端子のように，汎用の1ビット出力ポートとして使える端子があります．ここにダミーとしてLEDを接続し，ソフトのタイマによって点滅してやるわけです．プログラムはLSIイニシャライズ→LEDオン→ソフト・タイマ→LEDオフ→ソフト・タイマ→以下無限ループで点滅，というだけのものです．

このプログラムが走った状態で，一応，初期の基板チェックはOKとなります．あとはプログラムをふくらませて，割込みやそれぞれの入出力信号を動作させないと，それぞれの配線のチェックまでは終了したことになりませんが，まずはICEによってソフト開発するための土俵はできたことになるのです．

（筆者は最後に，このLED点滅テストの走行中に，データ・バスや$\overline{\text{WR}}$信号の端子などを，指で触ってまわります．そのぐらいのリークで点滅が不整脈になるようだと，まだどこかに問題があるのです．もっとキビシイひとは，ここで基板を持ち上げて，机の上に落とします．そのくらいの衝撃でも，パターンやハンダ付けに弱いところがあると，とたんに断線します．試作基板を腫れものに触るように大事に扱う，というのは，あとあといい結果を生まないのです．ここはひとつ，親獅子になったつもりで接したいものです．）

Appendix 3.2

アセンブル/ダウンロードのスピード比較

以下の**例1**～**例4**では，同一のクロス・アセンブラによって，同じような4096バイトの定数データからなるプログラムをアセンブルし，RS-232-Cによって転送する作業の時間を比較してみました．各プログラムはすべてRAMディスクに同一ファイル名で転送され，RAMディスク上のアセンブラ，リンカを経てインテルHEX形式のオブジェクト・ファイルとなり，DOSのコピー・コマンドで9600 bps，8ビット，1ストップ・ビットで転送しています．

例1は**リストA**のように，オフセット0000から連続した4096バイトのデータを配置したもので，**リストB**のように，

アセンブル＝10秒
リンク　　＝4秒
転送　　＝13秒

となりました．なお，このオブジェクト・ファイルは**リストC**のようなテキスト・ファイルです．

例2では**リストD**のように，4096バイトを256バイトずつまとめて，16ブロックの別々の位置に配置しました．この結果は**リストE**のように，

アセンブル＝11秒
リンク　　＝3秒
転送　　＝12秒

となりました．**例1**とほとんど変わらない（時刻測定に1秒の誤差があるので）のは，オブジェクト・ファイルが**リストF**のようにほぼ同じ形式となるからのようです．

例3では**リストG**のように，16バイトずつのブロックを256箇所に配置するようにしました．この結果は**リストH**のように，

アセンブル＝41秒
リンク　　＝3秒
転送　　＝13秒

となりました．**例2**と比べると，あまり絶対アドレスにサービス・ルーチンのエントリを固定して，飛び飛びのロケーションに小プログラムを置くのは，アセンブル時間の上では好ましくないようです．しかしオブジェクト・ファイルは**リストI**（アイ）のように同形式なので，RS-232-Cの転送は同じような時間です．

例4は極端な例ですが，4096バイトを**リストJ**のようにすべて別々の4096箇所に1バイトずつ置いたもので，結果は**リストK**のように，

アセンブル＝2分37秒
リンク　　＝15秒
転送　　＝1分5秒

となりました．ロケーションのマクロ展開を必死にやっているようです．オブジェクト・ファイルは**リストB**のように非常に冗長になる（1バイトのデータのために13バイトかかる）ために，こうなるとプログラム開発中にバッチで走らせている間に，ゆっくりとコーヒーを飲めることになります．

〔例１〕連続した4096バイトのデータ

```
;-------------------------------------------------------------
;       SAMPLE(1) : 4096 Byte - 1 Block
;-------------------------------------------------------------
        name    inter

        macro   %data_set                   ; 256 Bytes Block !
        defm    '0123456789ABCDEF'          ;       1
        defm    '0123456789ABCDEF'          ;       2
        defm    '0123456789ABCDEF'          ;       3
        defm    '0123456789ABCDEF'          ;       4
        defm    '0123456789ABCDEF'          ;       5
        defm    '0123456789ABCDEF'          ;       6
        defm    '0123456789ABCDEF'          ;       7
        defm    '0123456789ABCDEF'          ;       8
        defm    '0123456789ABCDEF'          ;       9
        defm    '0123456789ABCDEF'          ;      10
        defm    '0123456789ABCDEF'          ;      11
        defm    '0123456789ABCDEF'          ;      12
        defm    '0123456789ABCDEF'          ;      13
        defm    '0123456789ABCDEF'          ;      14
        defm    '0123456789ABCDEF'          ;      15
        defm    '0123456789ABCDEF'          ;      16
        endmac

        org     00000h

        %data_set       ;       1
        %data_set       ;       2
        %data_set       ;       3
        %data_set       ;       4
        %data_set       ;       5
        %data_set       ;       6
        %data_set       ;       7
        %data_set       ;       8
        %data_set       ;       9
        %data_set       ;      10
        %data_set       ;      11
        %data_set       ;      12
        %data_set       ;      13
        %data_set       ;      14
        %data_set       ;      15
        %data_set       ;      16

        end
```

（リストA）

```
F:¥[ 0:05:36]>rem                       〔１〕作業スタート！

F:¥[ 0:05:36]>copy c:sample1.zzz f:inter.s01
        1
F:¥[ 0:05:37]>ah180 inter

    Hitachi HD64180 Assembler V1.91/MD2
      (c) Copyright IAR Systems 1985


    Errors:  None           ##########
    Bytes:   4096           # inter #
    CRC:     8999           ##########

F:¥[ 0:05:47]>xlink -cz80 inter

    Micro Series Universal Linker V3.07A/MD2
      (c) Copyright IAR Systems 1987

Errors: none
Warnings: none


F:¥[ 0:05:51]>rem       *****  転送開始  *****

F:¥[ 0:05:51]>copy aout.a01 com1:
        1
F:¥[ 0:06:04]>rem       *****  転送終了  *****
```

（リストB）

```
:10000000303132333435363738394142434445464E
:10001000303132333435363738394142434445463E
:10002000303132333435363738394142434445462E
:10003000303132333435363738394142434445461E
:10004000303132333435363738394142434445460E
                    ⋮
:100FF8000303132333435363738394142434445468F
:100FFC000303132333435363738394142434445467F
:100FFD000303132333435363738394142434445466F
:100FFE000303132333435363738394142434445465F
:100FFF000303132333435363738394142434445464F
:00000001FF
```

（リストC）

〔例２〕256バイトずつ16箇所においたデータ

```
;-------------------------------------------------------------
;       SAMPLE(2) : 4096 Byte - Separate 16 Block
;-------------------------------------------------------------
        name    inter

        macro   %data_set                   ; 256 Bytes Block !
        org     ¥0¥999
        defm    '0123456789ABCDEF'          ;       1
        defm    '0123456789ABCDEF'          ;       2
        defm    '0123456789ABCDEF'          ;       3
        defm    '0123456789ABCDEF'          ;       4
        defm    '0123456789ABCDEF'          ;       5
        defm    '0123456789ABCDEF'          ;       6
        defm    '0123456789ABCDEF'          ;       7
        defm    '0123456789ABCDEF'          ;       8
        defm    '0123456789ABCDEF'          ;       9
        defm    '0123456789ABCDEF'          ;      10
        defm    '0123456789ABCDEF'          ;      11
        defm    '0123456789ABCDEF'          ;      12
        defm    '0123456789ABCDEF'          ;      13
        defm    '0123456789ABCDEF'          ;      14
        defm    '0123456789ABCDEF'          ;      15
        defm    '0123456789ABCDEF'          ;      16
        endmac

        %data_set       0       ;       1
        %data_set       1       ;       2
        %data_set       2       ;       3
        %data_set       3       ;       4
        %data_set       4       ;       5
        %data_set       5       ;       6
        %data_set       6       ;       7
        %data_set       7       ;       8
        %data_set       8       ;       9
        %data_set       9       ;      10
        %data_set      10       ;      11
        %data_set      11       ;      12
        %data_set      12       ;      13
        %data_set      13       ;      14
        %data_set      14       ;      15
        %data_set      15       ;      16

        end
```

（リストD）

```
F:¥[ 0:08:19]>rem                       〔２〕作業スタート！

F:¥[ 0:08:20]>copy c:sample2.zzz f:inter.s01
        1
F:¥[ 0:08:21]>ah180 inter

    Hitachi HD64180 Assembler V1.91/MD2
      (c) Copyright IAR Systems 1985


    Errors:  None           ##########
    Bytes:   4096           # inter #
    CRC:     2C78           ##########

F:¥[ 0:08:32]>xlink -cz80 inter

    Micro Series Universal Linker V3.07A/MD2
      (c) Copyright IAR Systems 1987

Errors: none
Warnings: none


F:¥[ 0:08:35]>rem       *****  転送開始  *****

F:¥[ 0:08:36]>copy aout.a01 com1:
        1
F:¥[ 0:08:48]>rem       *****  転送終了  *****
```

（リストE）

```
:10000000303132333435363738394142434445464E
:10001000303132333435363738394142434445463E
:10002000303132333435363738394142434445462E
:10003000303132333435363738394142434445461E
:10004000303132333435363738394142434445460E
:10005000303132333435363738394142434445546FE
:10006000303132333435363738394142434445546EE
:10007000303132333435363738394142434445546DE
:10008000303132333435363738394142434445546CE
:10009000303132333435363738394142434445546BE
:1000A000303132333435363738394142434445546AE
:1000B000303132333435363738394142434445469E
:1000C000303132333435363738394142434445468E
                    ⋮
```

（リストF）

〔例３〕16バイトずつ256箇所においたデータ

```
;-------------------------------------------------------------
;       SAMPLE(3) : 4096 Byte - Separate 256 Block
;-------------------------------------------------------------
        name    inter

        macro   %data_set               ;16 Bytes Block !
        org     ¥0+199
        defm    '0123456789ABCDEF'
        endmac

        %data_set       0
        %data_set       1
        %data_set       2
        %data_set       3
        %data_set       4
                        5

            :

        %data_set     249
        %data_set     250
        %data_set     251
        %data_set     252
        %data_set     253
        %data_set     254
        %data_set     255

        end
```

（リストG）

```
F:¥[ 0:11:04]>rem                       （３）作業スタート！

F:¥[ 0:11:05]>copy c:sample3.zzz f:inter.s01
        1
F:¥[ 0:11:06]>ah180 inter

    Hitachi HD64180 Assembler V1.91/MD2
      (c) Copyright IAR Systems 1985


    Errors:  None           #########
    Bytes:   4096           # inter #
    CRC:     6336           #########

F:¥[ 0:11:25]>xlink -cz80 inter

    Micro Series Universal Linker V3.07A/MD2
       (c) Copyright IAR Systems 1987

Errors: none
Warnings: none


F:¥[ 0:11:28]>rem       *****  転送開始  *****

F:¥[ 0:11:29]>copy aout.a01 com1:
        1
F:¥[ 0:11:42]>rem       *****  転送終了  *****
```

（リストH）

```
:1000000030313233343536373839414243444546 4E
:1000C70030313233343536373839414243444546 87
:10018E0030313233343536373839414243444546 BF
:1002550030313233343536373839414243444546 F7
:10031C0030313233343536373839414243444546 2F
:1003E30030313233343536373839414243444546 68
:1004AA0030313233343536373839414243444546 A0
:1005710030313233343536373839414243444546 D8
                        :
:10C0C800303132333435363738394142434445 46C6
:10C18F0030313233343536373839414243444546 FE
:10C2560030313233343536373839414243444546 36
:10C31D0030313233343536373839414243444546 6E
:10C3E40030313233343536373839414243444546 A7
:10C4AB0030313233343536373839414243444546 DF
:10C5720030313233343536373839414243444546 17
:10C6390030313233343536373839414243444546 4F
:00000001FF
```

（リストI）

〔例４〕4096箇所にはなればなれにおいたデータ

```
;-------------------------------------------------------------
;       SAMPLE(4) : 4096 Byte - Separate 4096 Block
;-------------------------------------------------------------
        name    inter

        macro   %data_set
        org     ¥0+256+7+0
        defb    ¥0
        org     ¥0+256+7+1
        defb    ¥0
        org     ¥0+256+7+2
        defb    ¥0
        org     ¥0+256+7+3
        defb    ¥0                      %data_set       0
        org     ¥0+256+7+4              %data_set       1
        defb    ¥0                      %data_set       2
        org     ¥0+256+7+5              %data_set       3
        defb    ¥0                      %data_set       4
        org     ¥0+256+7+6              %data_set       5
        defb    ¥0                      %data_set       6
        org     ¥0+256+7+7              %data_set       7
        defb    ¥0                      %data_set       8
        org     ¥0+256+7+8              %data_set       9
        defb    ¥0                      %data_set      10
        org     ¥0+256+7+9              %data_set      11
        defb    ¥0                      %data_set      12
        org     ¥0+256+7+10                     :
        defb    ¥0
        org     ¥0+256+7+11             %data_set     247
        defb    ¥0                      %data_set     248
        org     ¥0+256+7+12             %data_set     249
        defb    ¥0                      %data_set     250
        org     ¥0+256+7+13             %data_set     251
        defb    ¥0                      %data_set     252
        org     ¥0+256+7+14             %data_set     253
        defb    ¥0                      %data_set     254
        org     ¥0+256+7+15             %data_set     255
        defb    ¥0
        endmac                          end
```

（リストJ）

```
F:¥[ 0:15:07]>rem                       （４）作業スタート！

F:¥[ 0:15:07]>copy c:sample4.zzz f:inter.s01
        1
F:¥[ 0:15:08]>ah180 inter

    Hitachi HD64180 Assembler V1.91/MD2
      (c) Copyright IAR Systems 1985


    Errors:  None           #########
    Bytes:   4096           # inter #
    CRC:     E68C           #########

F:¥[ 0:17:45]>xlink -cz80 inter

    Micro Series Universal Linker V3.07A/MD2
       (c) Copyright IAR Systems 1987

Errors: none
Warnings: none


F:¥[ 0:18:00]>rem       *****  転送開始  *****

F:¥[ 0:18:00]>copy aout.a01 com1:
        1
F:¥[ 0:19:05]>rem       *****  転送終了  *****
```

（リストK）

```
:0100000000FF
:0100070000F8
:01000E0000F1
:0100150000EA
:01001C0000E3
:0100230000DC
:01002A0000D5
:0100310000CE
        :
:01FF4D00FFB4
:01FF5400FFAD
:01FF5B00FFA6
:01FF6200FF9F
:01FF6900FF98
:00000001FF
```

（リストL）

4．中級レベル：オリジナルCPUボード

実験室では，コンパクトで安価で柔軟性に富む標準的CPUボードを何枚か常備しておくと便利である．ここでは，そんなCPUボードを設計・開発する実例で，システム設計の各ステップを追ってみる．設計の第一歩はメモリ・マップの決定．どんな用途が発生するかわからないので自由度を最大限にもたせる．ついで，CPU周辺LSIの選択→回路の詳細設計→基板の試作ときて，ハードはほぼ終わり．ソフトの開発は，とくにICEなしの場合を詳しく述べる．さらに発展形態として，マルチCPUシステムやDSPシステムを，また，「ハードの違いを吸収してソフト的に変更が可能なシステム」を目標にセミカスタムICの導入を考えてみる．（編集部）

前章のボード・マイコンでは，パソコンを部品として使う場合に比べて安価で，かつスピードの向上したシステムを構築しました．一方その代償として，階層構造の点ではより下位に比重を移し，CPUに近づいたレベルのハード/ソフトの理解を必要とするようになりました．ここからの話はさらにその延長線上，よりローコスト・ハイスピードなマイコン・システムが目標となります．

オリジナルCPUボード

ボード・マイコンは，メーカが製品として利益を上げているのですから，当然，これを部品として使えばコスト高になります．それよりもシステムを開発する側が，自分で基板を設計・製作して，CPUやLSIなどの部品を載せたほうがコスト・メリットがあるのは明らかでしょう．ただし一般には，回路設計・ソフト開発ばかりでなく，基板のパターン設計や基板試作，あるいは部品購入・量産試作・生産移行といった各種の段階や，量産の数量ベースの要因が非常に複雑に関係してきます．実際の製品開発はこのように大変なので，ここでは，日常的に使用する**オリジナルCPUボード**（以下，ここでは専用CPUボードとも呼ぶ）として，市販のボード・マイコンより小回りのきく安価なものを設計する，という例を考えてみることにします．

筆者の経験でも，実験室で何かの簡単なツールとして，道具そのものをまず作ってしまう，ということはしばしばあります．たとえば，

「光ファイバで高速通信の実験をしたい」

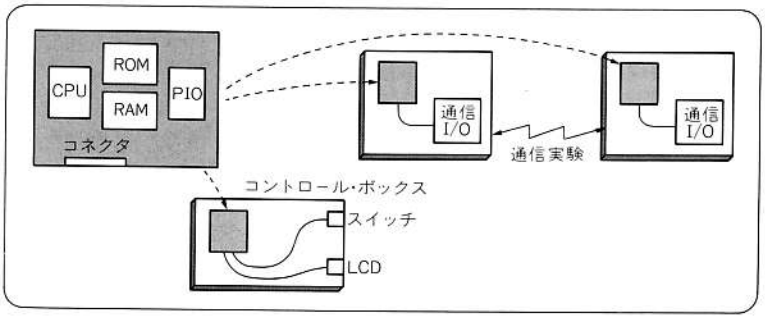
というときに，いちいち高価なプロトコル・アナライザが手元にあるとはかぎりません．そんなとき，高速シリアル通信データをバッファリングしながら，RS-232-Cと相互変換するようなCPUボードを1枚製作すれば，実験室のパソコンで十分に検証できます．また，

「複数のパソコンと複数のRGBモニタとの
接続を，リアルタイムに切り替えるような
演出のデモンストレーションをしたい」

というときにも，同じCPUボードを使って，スイッチから入出力のRGBコネクタまでを組み込んだ，小型のコントロール・ボックスを短期間に製作しました．これも，コンパクトで使い勝手のわかった，オリジナルCPUボードの機動性のいい例です．この他にも，簡単な計測ツールとしてLCD*パネルの付いた装置や，何枚も使ってLANの実験をした装置，最終的には壊れるような破壊実験に内蔵した装置など，オリジナル実験用CPUボードは，ローコストと手軽さを取柄として，実験室で大活躍しています（**例4**）．

専用CPUボードの最大のメリットは，自由にシス

〔例4〕
実験用CPUボードは
色々に活用できる



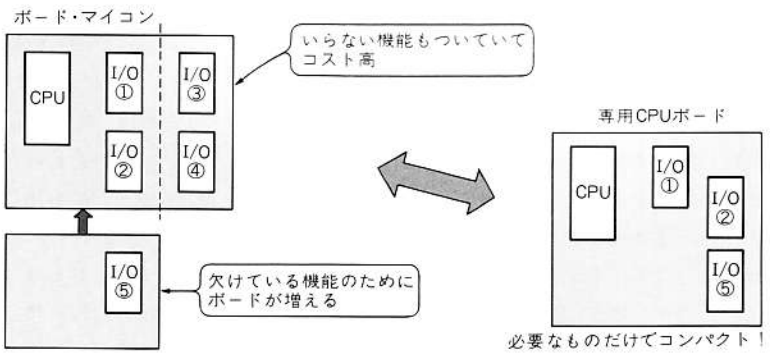
テムを設計できる点にあります．市販のボード・マイコンは最大公約数の機能を狙うために，個々の設計目標については過剰で，使わない機能や，不足して別個に増設しなければならない機能が出てきます．これが専用CPU基板では最適化でき，すっきりとしたシステムになります(**図4.1**)．また，CPUのアドレス・マッピングが自由になるのも大きな利点です．このマッピングは実験用ボードの場合，つねに共通のアドレスとなるので，その定義部分をヘッダ・ファイルとして登録しておくと，ソフト開発の際に毎回呼び出せて，とても便利な環境となります．筆者の場合，さらに共通のマクロ定義やLSI初期化・標準入出力ルーチンまでを登録してあり，何か新しく作ろうとすると，いきなり仕事本体だけを記述すれば完成する，という環境にしてあります(**リスト4.1**)．

専用CPUボードの短所もあります．市販のボード・マイコンは高価なだけあって，公開された回路図とデータ・シートの動作を保証していますが，オリジナルの場合には動作保証がありません．トラブルはすべて設計者の責任です．とくにマイコン・システムの場合，トラブルの原因がハードとソフトのどちらか不明であるような現象が多いため，ソフトだけデバッグすればよい市販ボード・マイコンとの差は歴然となります．デバッグの姿勢も，ソフトの原因を探りながら，同時にハードの可能性をつねに疑う，といった慎重な態度が必要です．もっとも，実験用ボードとしていったん検証されてしまえば，その後の使い回しは安心して活用できます．

システム構想・回路設計

さて，CPUレベルのマイコン・システム設計となると，上の階層のシステム設計とは，ポイントもテクニックもいささか変わってきます．これまでは，パソコンなりボード・マイコンなりの，外枠のハードウェアがあらかじめ決まっていました．ところがここではすべてが自由なのです(自由という名の責任の重さ！)．従来のマイコン技術では，ここから専門に回路設計をするハード屋と，そのシステムでプログラム開発するソフト屋とに分業する場合が多くありました．IC・LSIの細かいタイミングなどに設計ノウハウが必要だったためであり，また，プログラム・テクニックという名のワザによって，CPUの処理スピードがかなり左右されていたためです．しかし，**これからのマイコン・シ**

〔図4.1〕
専用CPUボードの
メリット



〔リスト4.1〕実験用CPUボードの定型プログラム例(マッピング定義/マクロ定義/LSI初期化などは共通にできる)

```
;-----------------------------------------;
;       Universal CPU Board Common Module  ;
;-----------------------------------------;
name  sample

;### I/O Address Defines ###
        bank_h          equ     08000h
        bank_l          equ     08001h
        uart_1          equ     08002h
        uart_2          equ     08004h
        led             equ     08006h
        timer           equ     08008h
        lcd             equ     0800ch
        pia_1           equ     08010h
        pia_2           equ     08014h
        program         equ     0c000h
        vector          equ     0fffah

;### Constants Defines ###
        bell            equ     07h
        tab             equ     09h
        cls             equ     0ch
        cr              equ     0dh
        esc             equ     1bh
        semi            equ     ':'
        prompt          equ     '>'

;### Macro Call Defines ###
        macro   &mov_i
                lda     #¥0
                sta     ¥1
                endmac
        macro   &mov_m
                lda     ¥0
                sta     ¥1
                endmac
        macro   &ok_ret
                bne     *+3
                rts
                endmac
        macro   &ng_ret
                beq     *+3
                rts
                endmac

;### Main Routine ###
org   program
start:
        sei
        ldx     #0
        txs
        cld
        jsr     ram_initial
        jsr     uart_initial
        jsr     lcd_initial
        cli
loop:
                        ; PROGRAM AREA !!!
        bra     loop
;### RAM Initialize ###
ram_initial:
        lda     #0
        ldx     #0
initial_loop_1:
        sta     0,x
        inx
        bne     initial_loop_1
        rts

;### UART Initialize ###
uart_initial:
        &mov_i  01000000b,uart_1+1 ; Software Reset
        &mov_i  01000000b,uart_2+1 ; Software Reset
        jsr     wait_long
; <UART #1 Initialize>
        &mov_i  11110011b,uart_1+3 ; System Mode Set
        &mov_i  00000010b,uart_1+2 ; Baud Rate Set
        &mov_i  01001110b,uart_1+1 ; Mode Set
        &mov_i  00010101b,uart_1+1 ; Command Set
; <UART #2 Initialize>
        &mov_i  11110011b,uart_2+3 ; System Mode Set
        &mov_i  00001100b,uart_2+2 ; Baud Rate Set
        &mov_i  00000001b,uart_2+1 ; Mode Set
        &mov_i  00000000b,uart_2+1 ; Command Set
        rts
```

ここにプログラム本体を入れてふくらませていく

ステム設計は，基本的には一人の技術者の領域となるでしょう．ハードの単純化が進む一方で，ソフトの小細工が不要になりつつあるからです．

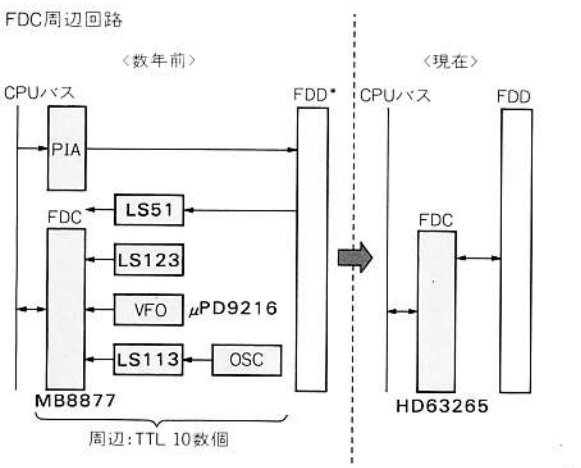
たとえばFDC*というLSIがあります．昔の本を見てみるとわかりますが，CPUがディスク・ドライブを制御しようとすると，かつては多くのIC・LSIによって，大規模かつ微妙なインターフェース回路を用いる必要がありました．これがどんどん専用LSI化され，今ではLEDを点灯させるPIOと変わらない手軽さで，回路図内の1ブロックとなってしまいました(**図4.2**)．FDCに限らず，あらゆるCPU周辺LSIが進化し続けています．

また，CPUの能力が低かった時代には，プログラミング・テクニックとして，1サイクル短く処理させるワザとか，間接的にアドレシングする方法とか，メモリを食わないプログラム手法，とかいったノウハウがありました．しかしCPUクロックが10倍ちかく高速化され，新しい豊富なレジスタ群をもち，アドレシングの拡張された命令を完備し，かつコストが変わらない現代のCPUにとって，これはプログラムを見にくくするだけのものです(**リスト4.2**)．**現代のCPUのプログラミング・テクニックといえば，むしろ見やすいように書く，多少冗長でも分かりやすく書く，**というところでしょうか．

それでは，マイコン・システムの設計の例を，具体的に考えてみましょう．ここでは実験用CPUボードということなので，まず第1段階では，CPUのマップを考えます(本当はCPUの選択からなのですが，ツールの関係ですでに決まっているものとします)．今後，どのような用途が発生するかわからないので，マップの可能性を最大限にとる必要があります(**図4.3**)．**ソフトウェアの自由度のために，メモリ空間をプログラムROMとデータRAMに分割する境界は，簡単に変更できるようにします．**場合によってはバンク・レジスタを増設してメモリ空間を拡張するので，アドレス・デコーダには予備のイネーブル入力も必要です．I/Oに

```
;### LCD Initial ###
lcd_initial:
        jsr     wait_long
        &mov_i  40h,lcd+1       ; System Set
        &mov_i  30h,lcd         ; Screen
        &mov_i  85h,lcd         ; Field(X)
        &mov_i  7,lcd           ; Field(Y)
        &mov_i  39,lcd          ; C/R
        &mov_i  48,lcd          ; TC/R
        &mov_i  3fh,lcd         ; L/F
        &mov_i  40,lcd          ; APL
        &mov_i  0,lcd           ; APH
        &mov_i  44h,lcd+1       ; Scroll
        &mov_i  0,lcd           ; Start Address #1(L)
        &mov_i  0,lcd           ;                  (H)
        &mov_i  40h,lcd         ; Line #1
        &mov_i  0,lcd           ; Start Address #2(L)
        &mov_i  10h,lcd         ;                  (H)
        &mov_i  40h,lcd         ; Line #2
        &mov_i  5ah,lcd+1       ; HDot Scroll
        &mov_i  0,lcd           ; Scroll=0
        &mov_i  5bh,lcd+1       ; Overlay
        &mov_i  01h,lcd         ; Character + Graphic
        &mov_i  58h,lcd+1       ; Display OFF
        &mov_i  14h,lcd         ; Parameter
        &mov_i  5dh,lcd+1       ; Cursor Form
        &mov_i  04h,lcd         ; X=5
        &mov_i  86h,lcd         ; Y=7
        &mov_i  46h,lcd+1       ; Cursor Write
        &mov_i  0,lcd           ; Home
        &mov_i  0,lcd           ;  Position
        &mov_i  52h,lcd+1       ; Erase
        jsr     wait_long
        &mov_i  59h,lcd+1       ; Display ON
        rts
;### IRQ Sequence ###
int:
        pha
        phx
        phy
                        ; PROGRAM AREA !!!
        ply
        plx
        pla
        rti

;### NMI Sequence ###
nmi:
        pha
        phx
        phy
                        ; PROGRAM AREA !!!
        ply
        plx
        pla
        rti

;### Waiting Timer ###
wait_long:
        ldx     #0
wait_loop_1:
        jsr     wait_short
        inx
        bne     wait_loop_1
        rts
wait_short:
        ldy     #0
wait_loop_2:
        nop
        iny
        bne     wait_loop_2
        rts

;### HEX Data Table ###
hex_table:
        db      '0123456789ABCDEF'

;### Interrupt/Reset Vector Table ###
org     vector
        dw      nmi
        dw      start
        dw      int
        end
```

〔図4.2〕集積化で周辺回路がシンプルになる



ジャンプ・テーブルのアドレスに選択的にジャンプをするルーチンの例

〔リスト4.2〕新旧のプログラミング・テクニック

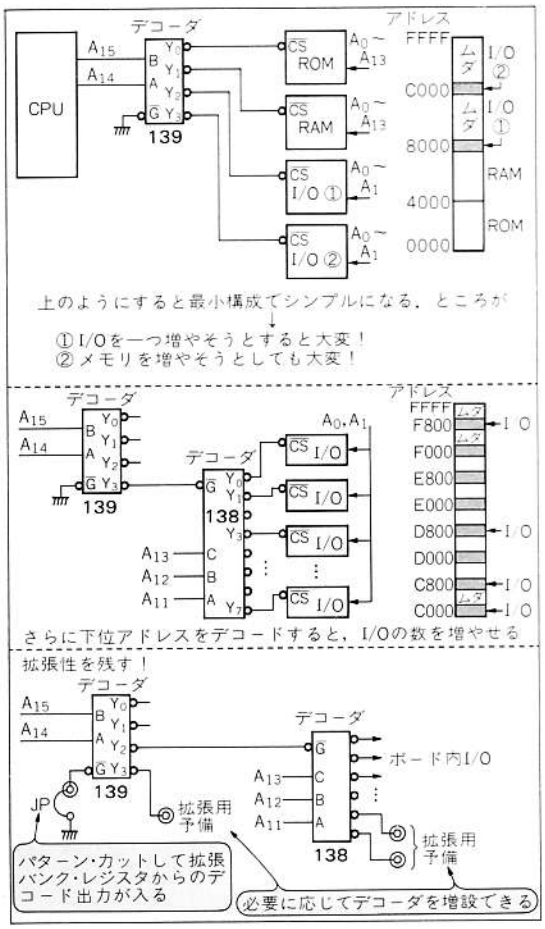
```
;--- Common Defines ----------------------------
jump_table:
        dw      routine_0       ; select = 0
        dw      routine_1       ; select = 1
        dw      routine_2       ; select = 2
        dw      routine_3       ; select = 3
        dw      routine_4       ; select = 4
        dw      routine_5       ; select = 5

;--- Old Zilog Z80 Type ------------------------
select_jump:
        add     a,a             ; select < 128
        ld      hl,jump_table   ; Table Address
        add     a,l             ; Add Offset
        ld      l,a             ;   , to L
        ld      a,(hl)  ; Routine Address to HL
        inc     hl
        ld      h,(hl)
        ld      l,a
        ex      (sp),hl ; Routine Address to SP
        ret             ; Jump to Routine

;--- New Toshiba TLCS-90 Type ------------------
select_jump:
        add     a,a             ; select < 128
        ld      hl,jump_table   ; Table Address
        ld      ix,(hl+a); Routine Address to IX
        jp      ix      ; Jump to Routine
```

ついても，拡張の可能性をなるべく多くもたせます．このようなハードの青写真は，むしろソフト経験者のほうが，夢を自由に広げることができるかもしれません．

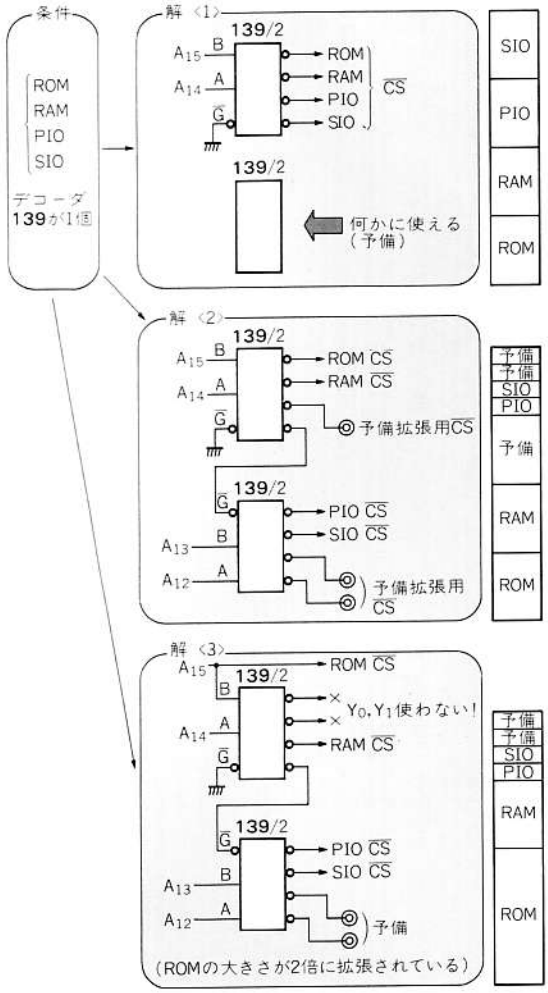
CPUとならんで，基板に搭載する**周辺LSIの選択**も重要です．一般的なパラレルI/O，シリアルI/OのLSIなどは，おそらくどんな実験用CPUボードにも載りますが，CPUはザイログ系でも周辺はインテル系にするとか，8251が四つ入ったLSIを使えばコンパクトになるとか，8255が二つ入ったLSIは別種で2社から出ているがどうするか，などを検討しながら，CPUと周辺を並べたブロック図を考えます(**図4.4**)．また，アドレス・バス，データ・バスを全部コネクタに出して，ハードの拡張性はそちらに回す，といった選択の幅もあります．いずれにしても，**CPU周辺LSIはつぎつぎに新しいものが発表されていて，基本的には後発のものは先行LSIよりも多機能・低価格なので，つねに各メーカの最新の状況を把握しておく**のは，マイコン・システム技術者にとって最低限必要なことでしょう．

アドレス・マッピングとブロック図の構想が固まれば，あとはCPUボードの回路図の詳細設計になります．周辺LSIのCPUインターフェースもずいぶん標準化・単純化されているので，よほど高速のシステムでないかぎり，チップ・セレクト，リード，ライト，リセットなどの制御信号を単純に接続しただけで動作します(**図4.5**)．I/Oの数も拡張を考慮して，ボード上のLSIだけでなく，多少の予備チップ・セレクト信号を作っておきます．デコード信号をなるべく多くして，かつデコーダのICを無駄なく美しく使うように，とい

〔図4.3〕CPUメモリ・マップの考え方



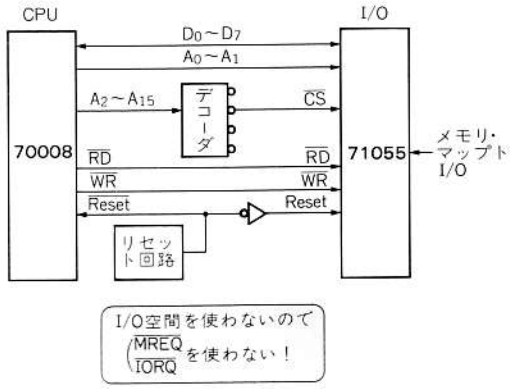
〔図4.6〕デコーダの使い方パズル



うのはパズルとして結構楽しめます(**図4.6**)．あとは，RS-232-C のレベル変換用や，CPU のリセット，ウォッチドッグ・タイマ用に，最近の新製品 IC を使ってみるとか，今回は CPU のワーク RAM に DRAM をためしに使ってみようとか，実験用ボードの場合には回路設計そのものに実験を持ち込めます．筆者の経験でも，DSP*，LCD パネル，FDD，IC カードなどの，のちに初めて製品に採用した部品のほとんどは，それまでに実験用 CPU ボードを使って「遊んで」みたものでした．

回路図ができれば，あとは基板の設計と試作になります．もしパソコン上で回路設計からアートワークまでやってしまう CAD があれば，それでもう終了します．CPU ボードの場合，A-D や D-A*周辺のノイズ対策，といったアナログ的なノウハウはあまり必要ないので，パスコンと電源・接地パターンさえ十分に引かれていれば，まず問題はないでしょう．もし，基板を実際に試作する前に，実際にブレッド・ボードを製作して確認する，という場合には，おそらくこの 1 回目がもっとも時間を要する作業となります．不思議なもので，慣れ親しんだ CPU 回路であっても，なかなか最初の通電で OK となることがありません．しかしこのブレッド・ボードのステップは，あとで述べる各種のシミュレータも出現しているので，今後のマイコン・システム開発の中では，次第に省略されていく運命にあります．

〔図4.5〕最小構成の CPU インターフェース

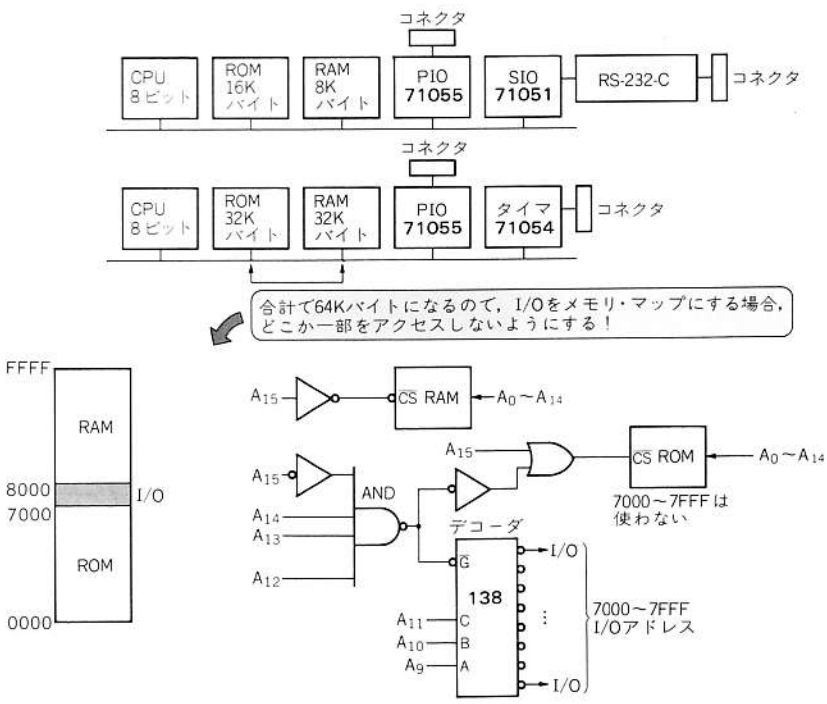


開発環境の開発

● ICE がある場合

基板の試作のどの部分までを担当するのかは，個々のケースによって異なりますが，最終段階のプリント基板製作を担当するところ(試作屋または試作部門)に試作依頼を出してしまえば，もう開発はソフトの領域

〔図4.4〕
周辺 LSI をならべてブロック図を検討し，メモリ・マップを設計する



に移ります．まずは〈ICE のある CPU〉という，非常に恵まれた環境の場合について考えましょう．これは，基本的には前章のボード・マイコンの開発と同様で，ソース・プログラムを開発→アセンブル→ ICE でデバッグ，というループの流れになります．ただ，オリジナルの回路の場合，一番最初のデバッグ時には，ハード的なミスによって CPU がまったく動かない，といった恐怖を体験する可能性もあります．このとき，ICE を使えるという条件は，言葉では伝えられないほど，デバッグを有利にするものなのです．バス衝突のようなハードウェアのミスから，部分トレース・条件ブレークなどのソフト検証まで，ICE のデバッガ機能を駆使すれば，相当に微妙なシステムのバグまで追跡して押さえることができます．とくに自作基板の場合，ソフトのデバッグ作業が同時にハードの確認作業であるだけに，ICE の示すデータへの信頼，という一点が救いなのです．最近の ICE は昔に比べてバグも減っているようなので，ポピュラなものならば安心です．ただし，**CPU メーカのオリジナル ICE の場合，とくに最新の CPU については，つねに ICE のバグを念頭に置いて作業する**必要があります．さんざん原因不明のバグを追求した末に，メーカの技術者に問い合わせると，「ICE のバージョンが古い」との回答に呆然としたこともありました．

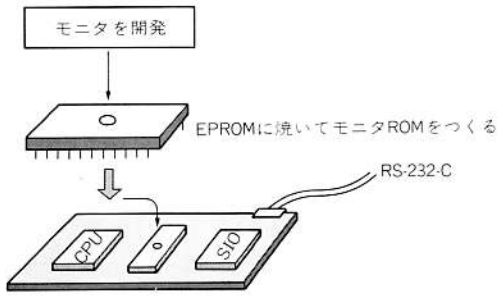
● ICE がない場合

さて，突然のプロジェクトで CPU 開発を行う，というときなど，手元に ICE が用意できない，という場合があります．また，一般に発表される前の新しい CPU を実験するときには，サンプル・チップだけで ICE が存在していない，という場合があります．このように〈ICE がない〉という条件でのマイコン・システム開発に直面すると，いよいよ技術者としての自分が試されることになります．開発環境そのものを，まず自力で開発するわけです．

ボード・マイコンの例を参考にすると，ここではモニタに相当するものをまず開発し，その環境下でさらにソフト開発を行う，という流れになるでしょう(図4.7)．最終的には全体のプログラムが ROM 化されるとして，まずはモニタ部分を確定させて EPROM 化し，つぎにモニタを経由して，RS-232-C などでプログラムを RAM 領域に転送し，モニタの下で RAM のプログラムをデバッグする，という方法です．モニタが稼働してからの開発は，前述のデバッグ・モニタ方式に準ずるので，ここではモニタ自身を開発する方法について考えてみます(p.232 の **Appendix 4.1「モニタ・プログラムの機能について」**参照)．

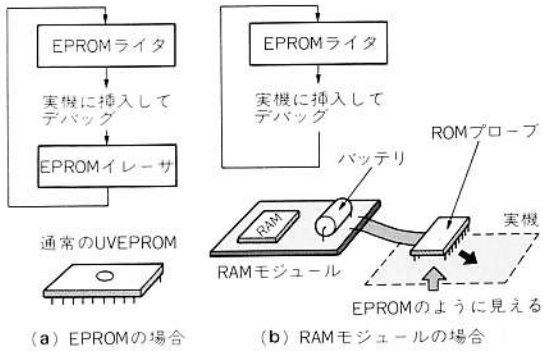
一番単純な方法は，プログラムをまず ROM ライタで EPROM に焼き，これを ROM ソケットに挿入して電源を入れてみる，という〈一発勝負・試行錯誤〉の作戦です．最近は ROM ライタで書き込める，バッテリ・バックアップされた RAM モジュールもあるので，これを毎回書き換えて試行してみる，というのも同類です(図4.8)．あるいは，ICE に似た外観をした，ROM ソケットにプローブを挿入する形式の装置で，システム内部にバッファ RAM があり，ここに RS-232-C でプログラムをダウンロードする RAM のエミュレート装置もあります．これだと，デバッグ機能を除いた汎用 ICE として，かなりの応用ができそうです(図4.9)．

〔図4.7〕モニタ ROM による CPU の開発例



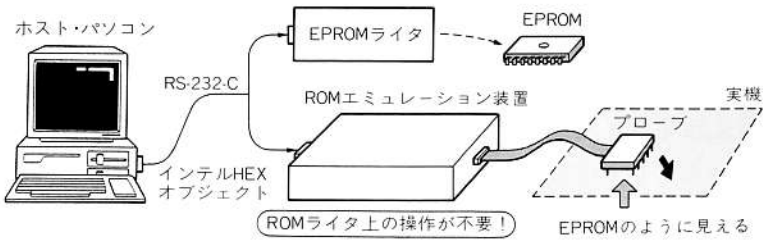
RS-232-Cによってモニタ下でソフト開発/デバッグを行う！

〔図4.8〕EPROM/RAM モジュールによる開発フロー



(a) EPROMの場合　(b) RAMモジュールの場合

〔図4.9〕
ROMエミュレータによる開発



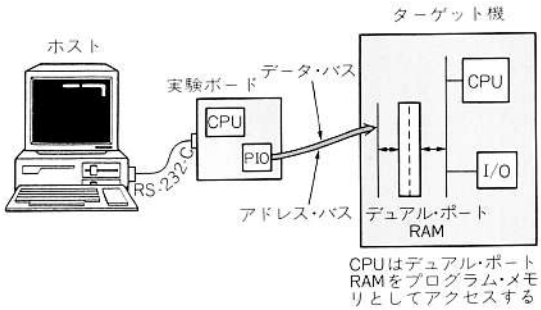
● ICEなしの開発例(1)

こんな便利な装置がなかった頃に，筆者の行った開発方法の例を二つ示しましょう．あまり美しい方法とはいえませんが，なんらかの参考にはなるかもしれません．

まず，すでに完成していたオリジナルの実験用CPUボードを使い，基板上の8251を2段重ねにして，シリアル通信ポートを1個増設しました．そして，クロック・ジェネレータとレベル変換用ICを接続して，パソコンのボーレートと合致したRS-232-Cを受信できる装置としました．このボード上のCPUプログラムとしては，受信したデータがASCIIフォーマットの文字データで，インテルHEX*形式のCPUプログラムである，と解釈するようにして，アドレスとデータに分離し，これをボード上のPIOから，それぞれケーブルを介してパラレル出力するようにしました(**図4.10**)．PIOの別の出力ビットはライト信号として，新しいアドレスとデータをセットするたびに，ライト・パルスを発生するように，“H”→“L”→“H”を繰り返します．このアドレス/データ/ライト信号が，実際のターゲットとなる開発用CPUボード上の，プログラム・メモリとして配置されたデュアル・ポートRAMの一方の側に供給されました．ターゲットCPUは，このデュアル・ポートRAMのもう一方を読み出すようにし，このCPU側はリード・オンリとして暴走を防止しました．

このようなシステムで，ホストのパソコン上でクロス・アセンブラによってソフト開発を行い，インテルHEX形式のオブジェクト・プログラムを，バッチ中に用意したDOSのコピー・コマンドによって転送しました．こうすると，ホストからプログラムを転送するたびに，ターゲット基板上のCPUをリセットすることで，新たなプログラムのデバッグを行えます．もちろん，ターゲット・システムのプログラムの中には，動作パラメータなどのデバッグ情報をLEDで外部にディスプレイしたり，モードを切り替えると何種類かのデバッグ・ルーチン，自己診断ルーチンを試行するような処理を組み込んだので，慣れてくると，まあまあ使いやすい開発環境となりました．

〔図4.10〕ICEなしの開発方法（1）



● ICEなしの開発例(2)

また別の機会には，まず，パソコンの拡張スロットに挿入する，データ・ラッチ・パラレル出力基板を自作しました．つぎに，DOS上のインテルHEXプログラム・ファイルを読み込んで，ポートからパラレルのアドレス信号，データ信号とライト信号を作って，外部に直接出力できるようなCプログラムを作りました(**リスト4.3**)．

新しいアドレスとデータをセットするたびに，ライト信号を制御してRAMへのライト・パルスとするのは，前のシステムと同様です．そして，ターゲットCPUボードの実験機の側では，今度はCPUメモリのアドレス・バス，データ・バスの全ビットをセレクタで切り替えて，パソコンからのバス・ラインと，ボード上のCPUのいずれかの制御下になるようにしました(p.223の**図4.11**)．この切替え制御信号も，ホストのパソコンの出力ポートの中に配置されていて，ホストのプログラムとしては，まず転送の冒頭でセレクタをホスト側に設定し，CPUプログラムをつぎつぎに転送し，最後にセレクタをターゲットCPU側に復帰させ

て終了する，という流れとなります．もちろん，CPUのリセット信号のポートも設けて，リセットもホストから与えました．こうすると，処理がパラレルであるために，RS-232-Cよりもかなり高速にプログラムをロードでき，CPUプログラム全体の開発が快適になりました．

このような，いわばゲリラ的な〈開発環境の開発〉の実験は，のちのちのマイコン・システム開発のときに，なんらかの形で生きてくる場合が多く，筆者は個人的

〔リスト4.3〕
図4.10の場合のホスト側プログラム

```
#include  <head.h>

int       data;
unsigned  address;

main(){                                          ┐
        io_port_initial();                       │
        program_download();                      │ メイン・ルーチン
        cpu_reset();                             │
}                                                ┘

program_download(){
        int i,fd,length;
        char file_buff[16];
        fd=fopen("test.hex","r");
        if(fd==0){return();}
        io_write_start();
        puts("¥n¥nﾌﾟﾛｸﾞﾗﾑ転送中.....");
        while(1){                                ┐
                fgets(file_buff,2,fd);           │
                if(strcmp(file_buff,":")!=0) break;
                fgets(file_buff,3,fd);           │
                if(strcmp(file_buff,"00")==0) break;
                sscanf(file_buff,"%2x",&length); │
                fgets(file_buff,5,fd);           │
                sscanf(file_read_buff,"%4x",&address);
                fgets(file_buff,3,fd);           │ インテル HEXプロ
                for(i=0;i<length;i++){           │ グラムを読み込んで
                        fgets(file_buff,3,fd);   │ ターゲット上のデュ
                        sscanf(file_buff,"%2x",&data);
                        program_data_put();      │ アル・ポート RAM
                        address++;               │ に書き込む
                }                                │
                fgets(file_buff,5,fd);           │
        }                                        │
        close(fd);                               ┘
}

program_data_put(){                              ┐
        if((address&1)==0) para_set(1,(char)data);
        else para_set(2,(char)data);             │
        _outb((char)(address&255),ADDRESS_LOW_PORT);
        _outb((char)(address>>8),ADDRESS_HIGH_PORT);
                                                 │ データとアドレスを
        _outb(0x38,CONTROL_PORT);                │ セットして疑似ライ
        _outb(0x3b,CONTROL_PORT);                │ ト・パルスを作る
}                                                ┘

cpu_reset(){                                     ┐
        int i;                                   │
        para_set(3,0x80);                        │
        for(i=0;i<10000;i++) { /* Waiting */ }   │ ターゲット CPU を
        para_set(3,0x00);                        │ リセット
}                                                ┘
io_write_start(){
        para_set(4,0x04);
}

io_port_initial(){
        para_set(4,0x00);
}

para_set(p,q)                                    ┐
        char p,q;                                │
{                                                │
        _outb(0,ADDRESS_LOW_PORT);               │ 実験ボード上に
        _outb(0x03|(p<<3),CONTROL_PORT);         │ データをセット
        _outb(q,DATA_PORT);                      │
}                                                ┘
```

には愛好しています．

マルチCPUシステム

マイコン・ボード(汎用ボード・マイコン，専用CPU基板)というシステムでは，中心に1個のCPUが鎮座しているような姿がイメージされてしまいますが，処理量・処理スピード・コスト要求などの理由から，一つのシステムに複数のCPUを使用したような発想も必要になる場合があります．

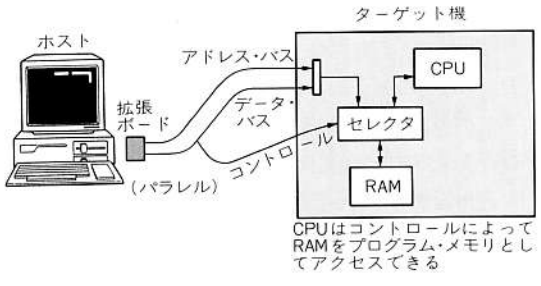
あらゆるケースをすべて網羅することはできませんが，マルチCPUのシステムについて，ここで三つの点にまとめて考えてみることにしましょう．

「処理量」という視点は，マルチCPUシステム化への第1のポイントです．たとえば，パソコンの中心のCPUがDOSなどの多くの機能を担当すると，キーボードのスキャンがお留守になって，少しもキー・バッファにキー操作をためこんでくれない，というもどかしさを，昔のパソコンでは多く経験しました．

ところが今のパソコンは，キーボード・スキャン専用のサブCPUをもっているのが普通で，システムがディスクをアクセスしていても画面を描きかえていても，キーボードにどんどんつぎのコマンドを打ち込んでいけます．このような分業は処理量を考慮したマルチCPUシステムの好例といえます．

また，パネル面に100個のスイッチと100個のLEDがならぶ，というような製品であれば，これは絶対にマルチCPU化すべき対象です．キー・スイッチのマトリクスとLEDが並んだ基板をすべてメイン基板のCPUからコントロールするためには，ラッチ/デコーダ/バッファなどのICやケーブルの本数がばかにならず，部品コストも製造コストも，さらにLEDダイナミック表示のCPUソフトもけっこう大変になります．

〔図4.11〕ICEなしの開発方法 (2)



これは，実際にあった例ですが，LEDドライバ付きの4ビット1チップ・マイコンをマルチに使う，という方法が有効でした．各4ビット・マイコンはそれぞれ20個のスイッチ・スキャンと20個のLEDダイナミック・ドライブを担当し，シリアル通信ポートによりメインCPUとやりとりします．5個のすべてに同一プログラムを使うので，マスクROM化された各1チップ・マイコンは200円そこそこの単価ですみ，メインCPUはLEDダイナミック・ドライブのスピードでなく，キー・スイッチを走査するという，人間からみると高速ながら，非常に低速度の処理として対応できました．考え方としては，**スレーブCPUを「インテリジェントな周辺LSI」ととらえる**，という発想で使いこなすわけです．

マルチCPUシステムの第2のポイントは**「処理スピード」**ということです．これは上の「処理量」という視点と似ていますが，マスタCPUの手足としてのスレーブCPUでなく，並列処理システムによるスピードアップという面を中心においた考え方です．CGなどのグラフィック処理で，ワークステーションやミニコンで一晩かかった作品，などというコンピュータ絵画を見ることがあります．また，物理現象のシミュレーションなどのように，単調ながら，膨大な計算量の必要な分野は多く，CPUスピードが勝負となる世界となっています．ところが，それぞれ8ビットか16ビットのCPUボードながら，タテ×ヨコに64枚を並べたシステムとか，32×32で1000CPUシステム，といった形態で，システム全体としてはミニコン以上のパフォーマンスを達成した報告を最近よく聞きます．

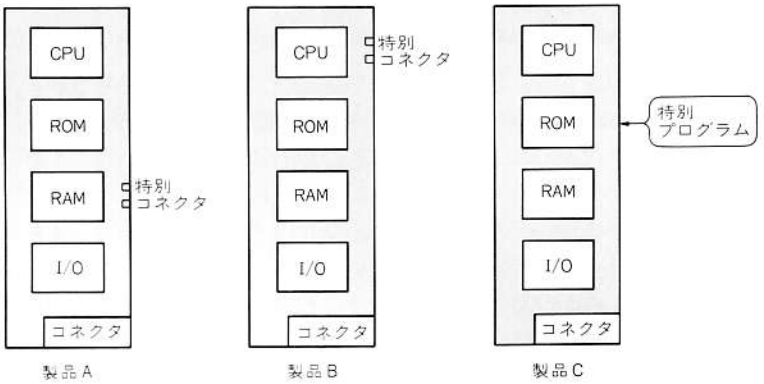
とくに最新のミニコンをどんどん購入できない大学の研究室などの学会報告に出てくるあたり，学術文化助成の貧困な日本ならではの領域なのかもしれません．このようなマルチCPUシステムのポイントとしては

① 個々のCPUブロックの行う処理の設定
② 各ブロック間の相互接続(データ通信)方法
③ 並列処理用ソフトウェア体系
④ システムと外部とのデータ入出力方法

などがあるようです．具体的な数字でいえば，1,000円のブロックが100個で10万円のコストなのに，うまくシステムを組むと1,000万円の専用機なみのパフォーマンスがある，とすれば，これはチャレンジしたい分野といえるでしょう．

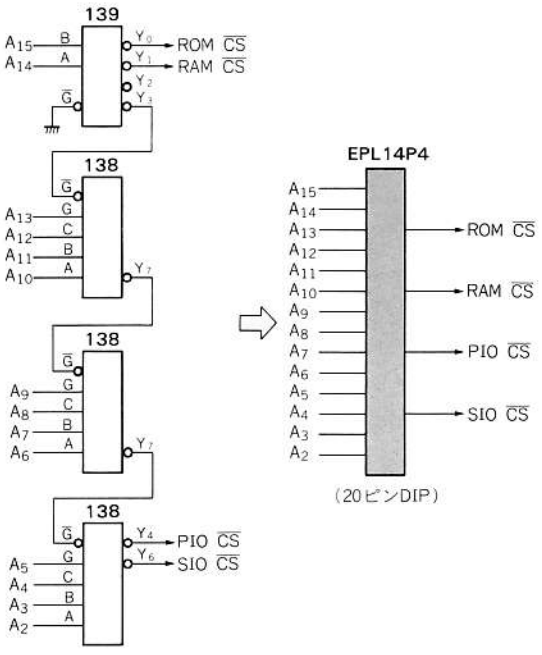
マルチCPUシステムの第3のポイントは，上にあ

〔図4.12〕
いろいろな製品のCPUボードをならべてみると，ハードの構成はほとんど同じだったりする



げた例ほど極端でない普通の意味での，**コスト要求**です．8ビットCPUが全盛の時代に，出たばかりの16ビットCPUを使ったシステムは，それ自体がセールス・ポイントでした．これは現在の16ビット対32ビット，あるいは32ビット対64ビットという形で，まったく同じように繰り返されています．ところが，32ビットをあえて使わなくても，16ビットCPUと8ビット・サブCPU，あるいは16ビットCPUのデュアル・システムで十分に仕事をカバーしている，という例はいたるところにあります．出はじめのCPUはコストも高く，さらに開発環境も不備がある場合が多く，コストと労力は割高でしかないのです．現在でも8ビットCPUはけっしてなくなっていない，という事実がなによりの証拠でしょう．仕事によりますが，高価な1個のCPUを能力いっぱいまで使ってなんとか実現されるシステム，というのはどうも心配で，もっと安くて普通のCPUの分業にならないか，と考える視点は重要だと思います．

〔図4.13〕PALの利用例（1）—— デコーダとして




```
!rom_cs = (^h4000>address)&(address>=^h0000);
!ram_cs = (^h8000>address)&(address>=^h4000);
!pio_cs = (^hffcf>address)&(address>=^hffc0);
!sio_cs = (^hffdf>address)&(address>=^hffd0);
```

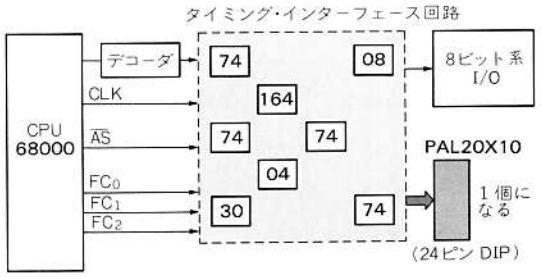
セミカスタムIC

さて，身の回りのいろいろな専用CPUボードを手にして，その上の部品をあらためて眺めてみると，CPU，メモリ，周辺LSI，ロジックICなどが並んでいます．何度か同じような基板を設計すると痛感してくるのですが，若干の違いを除いて，ハードはほとんど同じものです(図4.12)．そこで，毎回いちいち試作から繰り返すよりも，**ハードの違いを吸収して，ソフト的に変更が可能なシステムを作れないか**，という発想が出てきます．ここに登場するのが，ハードをユーザが書き込める，というプログラマブル素子の一群です．CPUのプログラム用EPROMもこの一種といえますが，ここではハードウェアの一部をソフト化する，という視点で考えてみることにします．

まず，**PLD***とか**PAL***と呼ばれるプログラマブル・ロジックICの場合，もっとも単純に，CPUのアドレス・デコーダとして活用できます(図4.13)．これは，**デコーダICの2〜3個分が1チップに収まる**だけでなく，**自由なデコード論理**が指定できたり，**仕様を臨時に変更する場合にハードが変わらない**，という点においても有効です．また，フリップフロップなどの順序

素子を内蔵したものも多いので，**CPU周辺のタイミング回路など，ロジック部分を簡潔な1チップにする場合**にも重宝します(**図4.14**)．また，セキュリティ・ヒューズによって内部の秘密保持を行うタイプのものも，政策的に重要な場合があります．たとえば，高価なパソコン・ソフトの中には，ソフト的なコピー・プロテクトはどうしても破られてしまうので，専用の基板をパソコンのスロットに挿入して，その上のPALの暗号をチェックしてから立ち上がるものも多くあるようです．

〔図4.14〕PALの利用例（2）—— CPU周辺として



●複雑/高速処理のためにハード・ロジックを使う

ここまでは，つねにCPUがシステムのあらゆる仕事を担当しましたが，処理内容やスピードによっては，ハード・ロジックで構成したほうがいい，という部分も非常に多くあります．たとえば，

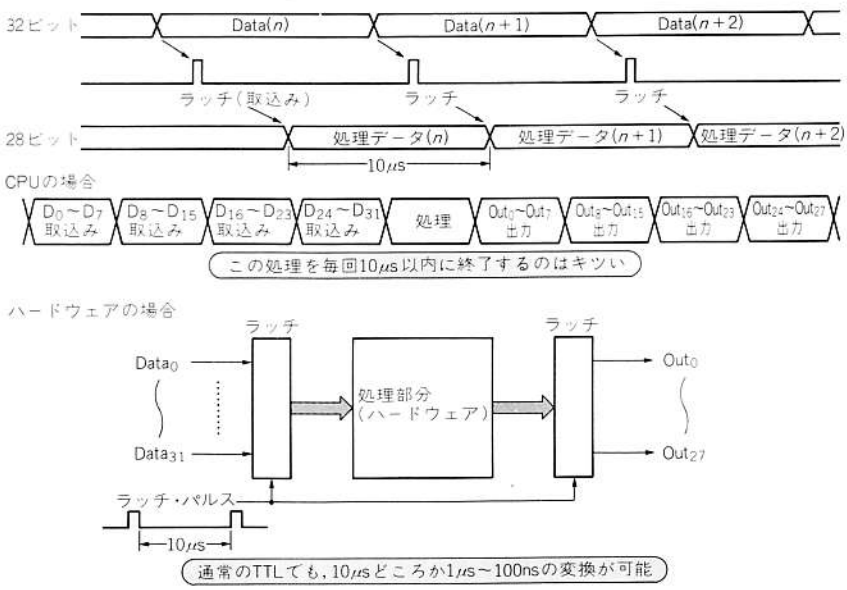
例5

連続的な32ビット幅のデータを，ある規則にしたがって変換処理して28ビット幅で出力する．ただし1サイクルの処理時間は10 μsである．

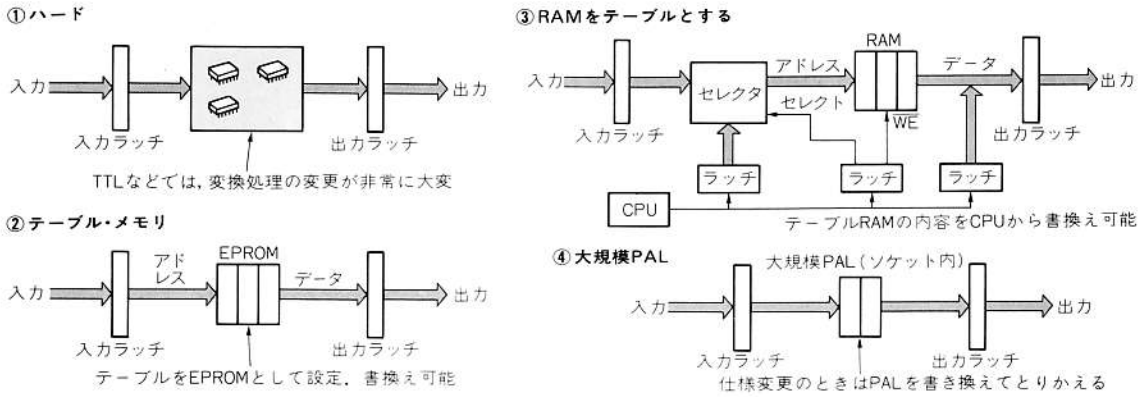
という処理は，他にも仕事を行うCPUでは，やや荷が重い感じです．この場合，変換処理が単純であれば，変換部分をテーブル参照方式のハード・ロジックで組んで，その参照テーブルをRAMにしてCPUから書き換えたり，大規模PALで構成する方法が考えられます(**図4.15**)．こうすることで，データの処理部分を変更しても，ソフト的な対応でかなりの追従が可能になります．

従来なら，このようなハードウェア部分は，多くの汎用ICによってランダム・ロジックで設計し，試作のブレッド・ボードを動かすためのデバッグ作業は，オシロを使ってしこしこと進める，というのが定石でした．ところが，**LCA***とか**プログラマブル・ゲートアレイ**と呼ばれるデバイスの登場により，状況はかなり変化しています(**図4.16**)．ここでは，TTLやCMOSの回路図の代わりに，LCA設計用のパソコンCADで回路を設計します．シミュレーション・ソフトによって，論理テストばかりでなく，遅延シミュレーションもできるので，高速オシロやストレージ・スコープによる実機デバッグのかなりの部分を，あらかじめパソコン上で確認できます．その上で最終回路を決定すれば，

〔例5〕
ハード・ロジック処理の例



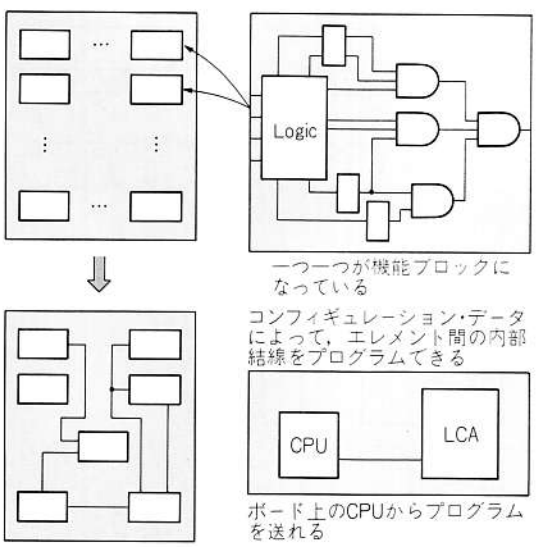
〔図4.15〕ハードウェア処理のソフトウェア処理への変更



LCA チップに転送する内部結線情報を作成し，これを EPROM に書き込んで完成です．あとは，ボード上の LCA と接続された EPROM から，リセット時に自動的に LCA に内部結線情報がロードされ，システムが目的のハード・ロジックとして構成されてしまいます（**図4.17**）．これで，TTL で何 10 個というボードが，数個の LCA と EPROM のすっきりしたボードになります．さらにこのボードは，LCA 開発ツールを用いて，別の仕様にもとづいて設計しなおせば，まったく別のハードウェア・システムに生まれ変わることも可能なのです．

実際には，LCA を取り巻く環境はまだまだ開発途上で，CAD ツールのレベルや，できあがった回路，配線の効率，あるいはコスト的な問題がかなりあります．現在のところ，筆者の場合には，組込みシステムのハード用というよりも，後述する**ASIC のプレッド・ボード用の試作ツール**と位置づけています．しかし，なによりも机上のパソコンだけで，**ソフトによってハードを設計する**，という雰囲気を実感できるだけで楽しいものです．最近の傾向の，〈少量多品種〉〈高機能高級機〉という流れには，上手に活用すればうまく乗っていける開発環境といえます．

〔図4.16〕LCA のしくみ



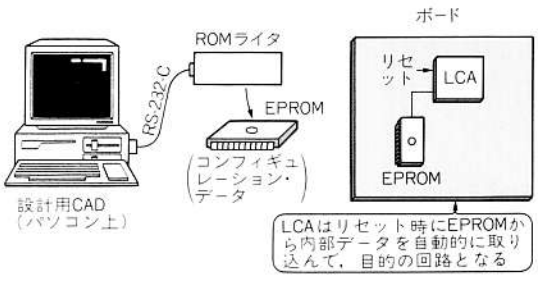
DSP システム

●同じような処理を続けて実行する場合は DSP が有効

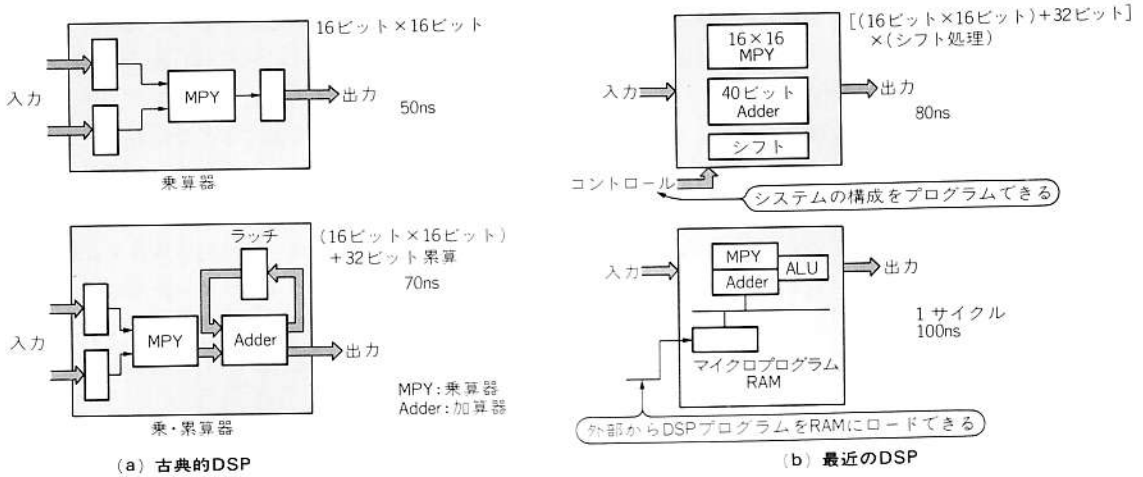
CPU を中心にしたマイコン・システムでは，CPU がプログラムをフェッチして実行するというサイクルよりも高速な処理は，基本的に不可能となります．普通の CPU では，数 μs あたりがこの限界です．これ以上の処理，たとえば，

「加速器実験で，散乱槽のガンマ線センサから

〔図4.17〕LCA の開発



〔図4.18〕DSPの例



の信号を，同時に10万ポイント集計する」

「背景を描画しながら，複数のでかキャラクタの色を変化させつつ別個にスクロールする」

「音声信号をA-D変換して，リアルタイムにFFT*分析結果をディスプレイする」

といった仕事には，専用のハードウェアが必要になります．とくに，同じような処理を続けて実行するような場合，DSPのような専用LSIを用いると，回路規模や開発効率の上で有効です．DSPでは，乗算・加算といった基本的な処理の専用ハードウェアが内蔵されていて，たとえば，16ビットの乗算と32ビットの加算を数10 nsで実行します(**図4.18**)．これだけでは必要な信号処理にはならないので，演算・移動・入出力・条件分岐といった基本命令を組み合わせて，DSPの動作を決定するプログラムを作ることになります(**リスト4.4**)．DSPのプログラム開発ツールも，最近はパソコンをベースとしたシステムとして，かなり出回ってきました．DSPプログラムができれば，EPROMに焼いたり，ホストのCPUからプログラムRAMに転送して，あとはDSP自身が勝手に動作してくれます．もともとDSPのプログラムは，CPUのように複雑で長いものではないので，簡単なものは手でも書けるのです．

さて，この段階になると，DSPというデバイスを使うことよりも，むしろ，ディジタル信号処理システム

〔リスト4.4〕DSPのプログラム例

```
        ORG     MAIN,¥0000
ANGL    EQU     0
BUFF    EQU     1
WAIT    JOC     WAIT,IF
        MOV     EI,D
        MOV     ANGL,D
        LDI     #0.1
        LDI:SUM #¥3FFF
        NOP:AND
        MOV     D,ANGL
        JSR     SINE
        MOV     D,EO
        MOV     #¥1000,EA
        JMP     WAIT
SINE    LDI     #0.5
        NOP:RED
        LDI     #1.0
        JOC     S1,MI
        LDI     #¥C000
S1      MOV     A,BUFF
        NOP:ABS
        LDI     #0.25
        NOP:RED
        NOP:ABS
        NOP:NEG
        NOP:SUM
        MOV     D,A
        MOV     #¥0200,B
        NOP
        LDI:MLT #128
        NOP:RED
        JOC:SUM S2,PL
        MOV     D,A
        MOV     X,D
        MOV     #¥0400,PGT
        MOV     A,X
        LTB     0(X),0
        MOV     BUFF,B
        MOV     D,PGT:X
        RTS:MLT
S2      MOV     BUFF,D
        RTS
        ORG     TABLE,¥0400
        ......................
        END
```

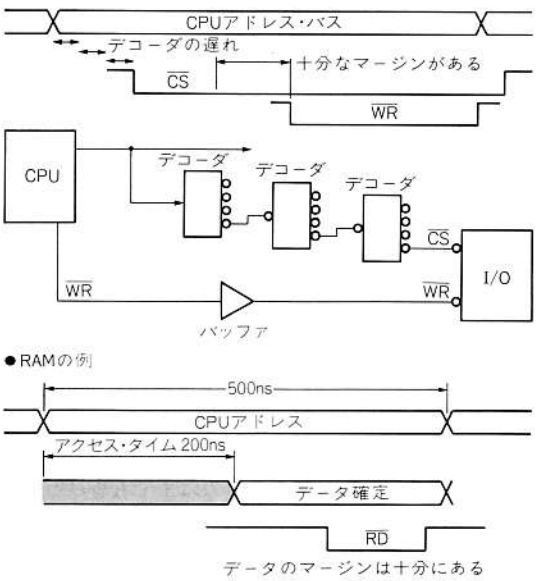
のハードウェアを，DSP的な発想で設計する，という考え方のほうが重要になってきます．ここまでのディジタル技術というのは，CPUのソフトによる逐次処理と，非同期ロジックによる論理処理，というものでした．つまり，CPUのデコード回路を例にとると，デコーダICのスピードによる遅延は10数ns程度で，RAM，ROMのアクセス・タイムと合計してもCPUのサイクル・タイムよりは短く，タイミング的な考慮は不要でした(**図4.19**)．つまり，CPUのスピードの<高速性>の範囲内だったわけで，これはボード・マイコンや，その上のパソコンの階層においては，よりいっそう明らかでした．CPUによっては，高速化のために命令をプリフェッチするものもありますが，ハード的な視点として並列処理を意識する必要はありませんでした．

ところが，CPUを越えたスピード領域のディジタル回路の場合，回路の全体を同期的に設計する必要があり，さらに，時分割・パイプライン処理のような，並列処理動作の概念が必要になります(**コラム4.2**)．通常のメモリについても，CPUオーダの回路の速度ならば，

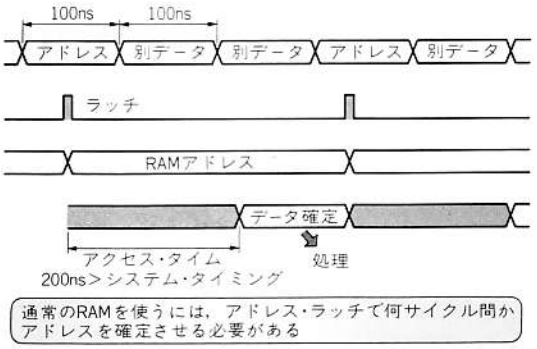
「アドレスを与えればデータが出てくる素子」

という考え方で十分でした．ところが，1サイクルが数10 ns程度の時間スケールになるDSPの世界では，

〔図4.19〕CPUレベルの回路スピード



〔図4.20〕DSPシステム中の普通のRAM



コラム4.1

PALライタ・ROMライタ

メモリICがどんどん大容量化し，PALの新ラインナップが続々と発表されている，という状況の中で，意外にしぶとく対応しているのが，ROMライタ・PALライタの業界です．単体として完結したシステムでは，1ランク上のメモリが出ても対応できなくなるので，レベルを限定してよっぽど定価を下げなければ売れません．それよりも，つぎつぎとバージョンアップして対応できるような，ソフト・ハードの技巧が凝らされた機種が多くあります．親亀の本体に子亀のモジュールを搭載し，さらに孫亀のICソケット・モジュールを買い換えればどんどん対応できる，なんていうのはざらです．対応ソフトもつぎつぎに発表され，たんに品種ラインナップの総数を増やすだけでなく，少しずつPALシミュレーションのバグをとったりレベルを上げる，といった「売り」を含めているあたり，非常に商売がうまいところです．

毎回のバージョンアップに正直に対応していると，そのコストの総額は，やがて本体価格を軽く突破してしまう場合もあります．そういえば，国内ASIC*メーカの開発環境である，LSI設計のシミュレーション・ソフトというのも，ほとんどはアメリカからのものであり，このバージョンアップとなると1回数100万円もするので，やはり国内の大メーカにも弱みがあるな，と変に感心してしまいます．マイコン時代のビジネスというのは，このような意味でのあとあとまで糸を引くようなソウトウェアにこそあるのであって，力仕事でとにかく作って売るだけ，というようなソフトは消えていくのでしょう．

「アドレスを与えてから何サイクルか後に，
やっとデータが出てくる素子」
と考えることになります(**図4.20**)．このため，「全体を統括するシステム・クロック回路」と，「マイクロプログラムによって各部のタイミング信号を発生する回路」と，「パイプライン的に時分割動作によって信号を共存させるバス・ライン」と，そこで「演算処理を実行するDSP」と，「タイミングを合わせるためのインターフェース回路群」という構成のシステム設計が成立してきます(**図4.21**)．これは，従来のディジタル回路やCPU回路が，ある意味で静的であるのに対して，複雑で動的な回路技術の世界ともいえるでしょう．圧縮された説明になりましたが，この部分は，ハードウェア設計技術・思想の，一つの大きなブレーク・スルーとなる，非常に重要なポイントだと思っています．

筆者の場合，この領域の回路設計については，いま

コラム4.2

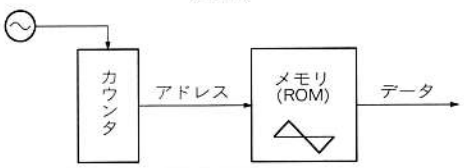
パイプライン処理の考え方

たとえば，図Aのようにメモリにある信号パターンがあって，カウンタの出力をアドレスとして順にメモリの内容を読み出すとします．

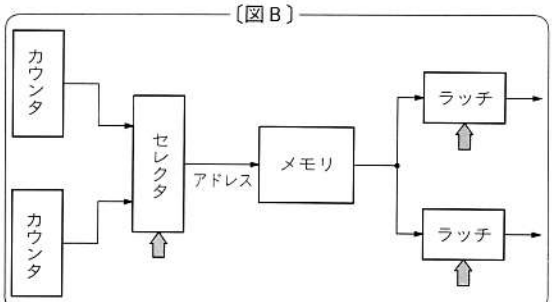
これを2種類の信号パターンにするのに，図Aの回路をそのまま二つ作るのは効率が悪いでしょう．メモリは一般に容量がとても多いので，図Bのような考え方も出てきます．この場合，セレクタとラッチに与える信号として，全体のタイミングを二つの出力系列ごとに交互に使うようにします．

もしこの2出力が16出力とか，64出力になったらどうでしょう．別々のカウンタをもつのは大変なので，図Cのようにしてみるわけです．カウンタとは前回の値からいくつか増減している値があればいいので，二つのメモリを用いると，いくつでも系列を増やすことができます．この場合，図Dのようにタイミングを考えます．これをパイプライン処理といいます．

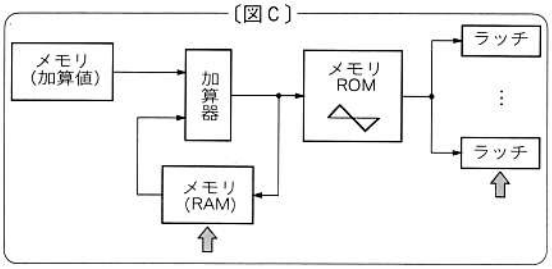
〔図A〕



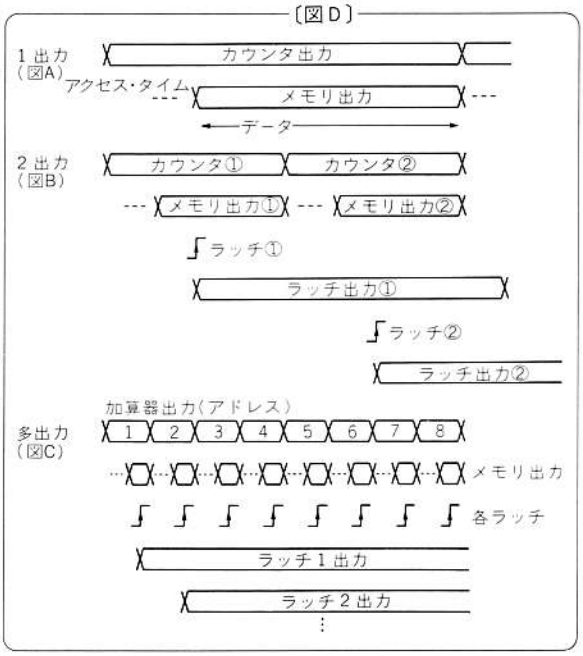
〔図B〕



〔図C〕

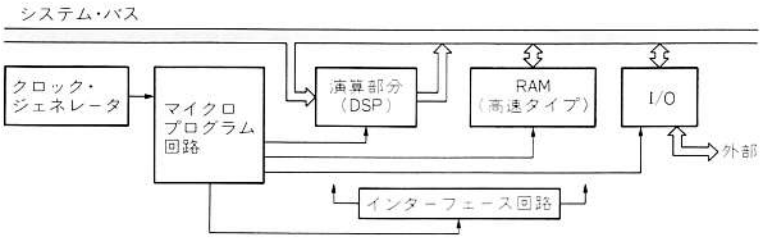


〔図D〕



だに定型的な方法の決定版はありません．あるときは，まず方眼紙にタイム・チャートの罫線を引いて，ここに各ブロック間のデータの流れを，時間順に詳細に記入していったことがあります(**リスト4.5**)．また，パソコンのエディタで，信号処理の関数表現式を思いつくまま列挙しながら，次第に処理系列をふくらませて，タイム・チャートの様相に成長したこともあります(**リスト4.6**)．あるいは，最初にハードウェアのブロック図を書き，つぎにこれを何枚もコピーして，映画のコマ送りのように，各サイクルの信号の流れを記入しながらタイミングを検討したこともあります(**図4.22**)．これらの例のように，この種のディジタル回路は，目的となる信号処理形態によって，CPU回路よりも非常に多くのバリエーションがあるので，設計方法もケー

〔図4.21〕DSPシステムの考え方



コラム4.3

ハイブリッドIC

最近のASICの流行でやや隠れているものの，ハイブリッドICというのも，システムのカスタム化を強力に支える武器といえます．製品が人間とインターフェースするかぎり，どこかでアナログに変換されるレベルがあるので，場合によっては，アナログIC，チップ部品を組み合わせたハイブリッドICの，コンパクト化というメリットが効いてくるのです．通常の場合には，個々のアナログ部品で組んだ場合の2倍程度のコストとなる，という通念があり，これと体積で1桁程度コンパクト化するコスト・メリットと比較する，という発想になります．

ところがじつは，あまり知られていないことですが，この常識が通用しない世界があるのです．それは，世界にとどろく日本の巨大な業界，自動車業界です．自動車に搭載されるとなると，その数量は驚くほどのオーダとなり，通常の電子部品とはケタ違いの量産効果があります．また，自動車の電子系統ほど，悪条件と安全性要求の板ばさみでいじめられる分野も少なくなく，この業界での信頼性設計のレベルもまた，他とは別世界のようなものです．そこで，一般に半導体を外販していないながら，技術レベル・生産規模ともに相当のところにある隠れた巨大メーカ，というものがこの業界には存在しているのです．

あまり具体的な実例を出すわけにはいかないので，たとえばラジコン・カーに内蔵するハイブリッドICの話，として考えてみましょう．無線の受信部分は汎用のICがありますから，CPU周辺・制御系・OPアンプなどの部分を対象に，個別の部品で試作してみると体積が大きくて収容できなかった，とします．ここでもし，通常の電子部品メーカにハイブリッドIC化をもちかけると，ミニフラットICとチップ部品を小型のセラミック基板に載せて，全体をモールド封入したような，ディスクリート構成の2倍から3倍のコストの機能モジュールができあがります．ところがもし，クルマ関係の某メーカに依頼すると，なんとディジタルもアナログもすべて含んだような半導体チップを自社製作してしまい，できあがりの体積も通常のハイブリッドICの数分の1，それでいてコストはディスクリート並，というような芸当を演じてしまいます．さらに，最初に設計したディジタル回路の動作を信頼性のために二重，三重に確認する機能や，OPアンプ周辺の保護・精度補正回路なども，メーカが独自のノウハウで追加した設計としているために，実質的な半導体数は2倍以上になっているのです．もちろんここには条件があって，クルマほどではないにしても相当の数量を前提にすること，またASICレベル相当の開発費用も必要となります．それでも，あらためて日本の自動車産業の実力を見直すような実例であると思います．

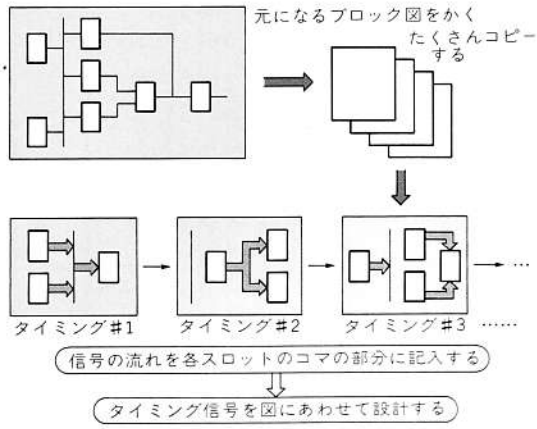
〔リスト4.5〕

Tim/Sig	DB<1>	DB<2>	MPY	Adder	SEL	RAM	GATE
0	Data (0)	Para (0)				WR	Zero
1	Data (1)	Para (0)	D * P 0-0	Cont (A)		RD	
2	Data (2)	Para (0)	D * P 1-0		Total (A)	WR	Ack
3	Data (3)	Para (0)	D * P 2-0	Cont (B)		RD	
4	Data (0)	Para	D * P		Total (B)	WR	Ack
		(0)					
		Para	D * P				
		(4)	3-3			RD	
17	Data (1)	Para (4)	D * P 0-4	Cont (A)		RD	
18	Data (2)	Para (4)	D * P 1-4		Total (A)	WR	Ack
19	Data (3)	Para (4)	D * P 2-4	Cont (B)		RD	
20	Data (0)	Para (5)	D * P 3-4		Total (B)	WR	Ack

〔リスト4.6〕

```
a1)  Addr'[A] = Addr[n][B][A] + Para[n][B][A]

a2)  Addr'[B] = Addr[n][B][B] + Para[n][B][B] + (CY)

a3)  Addr'[C] = Addr[n][B][C] + Para[n][B][C] + (CY)
     Address = Addr[n][B][C] + Para[n][B][C] + (CY)

a4)  Addr[ ][p] = Addr'[p]  (p=A,B,C)
     P    = Pdiff[n][A]

c)   Addr[n][A][p] = Addr'[p]  (p=A,B,C)
     Rdiff = Qdiff[n]

b13) Addr'[C] = Rtst[n][A] + Ctst[n]

b14) Rtst[n][A] = Addr'[C]

b15) Addr'[C] = Rtst[n][C] + (CY)
     Rtest = Rtst[n][C] + (CY)

b16) Rtst[n][C] = Addr'[C]

Timing  <0> <1> <2> <3> <4> <5> <6> <7> <8> <9> <A> <B> <C> <D> <E> <F>
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-1  a1  a2  a3  a4   .   .   .   .  a9  a10 a11 a12   .   .   .   .
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-2  a1  a2  a3  a4  a5  a6  a7  a8  a9  a10 a11 a12   .   .   .   .
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-3  a1  a2  a3  a4  b5  b6  b7  b8  a9  a10 a11 a12   .   .   .   .
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-4  a1  a2  a3  a4   .   .   .   .  a9  a10 a11 b12 b13 b14 b15 b16
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-5  a1  a2  a3  a4  a5  a6  a7  a8  a9  a10 a11 b12 b13 b14 b15 b16
          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
MODE-6  a1  a2  a3  a4  b5  b6  b7  b8  a9  a10 a11 b12 b13 b14 b15 b16
```

〔図4.22〕DSPシステム設計：コマ送り方式



スパイケースとなるようです．

このようなシステム発想に慣れてくると，**ビットスライスCPU**を使った回路設計も可能になります．ビットスライス方式の利点は，目的に応じたビット幅の，最適の構成のマイコン・システムを，DSP的な効率的発想で設計できる点にあります．しかし一方で，汎用CPUによる回路設計とは段違いに難しく，数量的にもコスト的にも，現実に適用できる領域はかなり限定されます．実際には，汎用CPUの機能の進歩と，次章で述べるASICシステムに対するメリットの相対的な低下によって，歴史はあるのに，なかなかお目にかからないようです．

Appendix 4.1

モニタ・プログラムの機能について

ICEなしでCPUプログラムを開発するためのオンボード・モニタROMの機能としては，どのようなものが必要でしょうか，「こんな機能があると便利」「こんな機能も盛り込んでみました」…というような豊富なメニューの例は，本誌のバックナンバをさがすと多く見つけられるので，ここでは最小限のものを考えてみます．

システムのイメージとしては，**図A**のようになるので，じつはオンボードのモニタというソフトとともに，パソコン側のコントロール・プログラム(リモート・デバッガのようなもの)もあわせて製作することになります．モニタのコマンドが一つ増えると，その処理ルーチンとともに，対応するホストのコントロール・プログラムも一つ増えるわけで，この意味でも，いたずらに機能を多くするのは手間となるのです．

(1) 転送データのエコーバック

これは全体の実行スピードをやや遅くしますが，回線が「死んでいない」，CPUが一応ちゃんと走っている，という検証でもあるので，筆者はかならずパソコンから送られるデータはパソコンのプログラムで表示せず，エコーバックされたデータとして表示するようにしています．

(2) プログラムのダウンロード

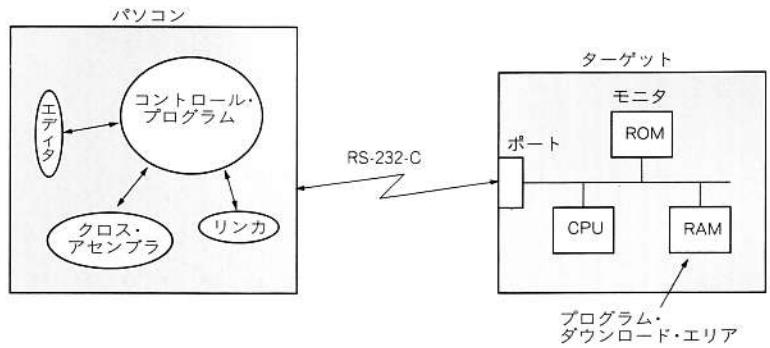
一般のクロス・アセンブラの出力として，インテルHEX形式などのオブジェクト形式があるので，これをそのまま利用するのが便利です．この場合，コマンドは1文字か，もしくはいきなり先頭のコロン(：)を予約語としておけば，アドレスやデータ長をコマンド・パラメータとして送る必要もありません．ただ，文字形式で2倍のデータ量になっているので，多量のデータではスピードの面で問題もあります．

(3) データのダンプ

(4) プログラム実行(GO)

(5) プログラム停止(STOP)

〔図A〕
モニタ・プログラムの機能



● DSP システム設計上のポイント

DSP を用いるようなシステムでは，CPU システムよりもクロック(水晶発振子の周波数でなく，システム動作のタイムスロット)が 10 倍程度は高速になるため，とくに素子のタイミングに注意します．おもなポイントとしては，

① 内部演算用 RAM の選択では，アクセス・タイムばかりでなく，$\overline{CS}$，$\overline{WR}$ 信号の前後のマージンについても，十分データ・シートで確認する．

② パラメータ変更のために PAL を使う場合，信号遅延の最悪値に対してマージンをとる．

③ システム・クロックが回路の各部分に同期して供給されるように，分周カウンタ以降の論理回路/セレクタ/デコーダ/バッファなどの遅延に注意する．

④ 素子の遅延によってマージンをかせぐ設計は，理論上は正しくても危険な発想である．正しい解決法は，システム・クロックを 2 倍にして同期的な時間差を設計することである．

⑤ 動作のあやしい TTL では，スピード・バージョンの上のクラス(LS → F など)で試してみる．

などがありますが，これればかりは経験によって身についてくるものなので，CPU 回路のように一発ではいかない世界だろうと思います．

---

(6) データの修正(EDIT)

これらは最低限のものでしょうか．程度によりますが，このくらいで一応のことはできます．あとは希望すればキリがありませんが，

(7) レジスタのモニタ

(8) 逆アセンブル・リスト表示

(9) データの Fill Up(指定範囲を指定データで埋める)

(10) I/O ポート・コントロール

(11) シングル・ステップ実行

などと，どんどんデバッガそのものの機能に向けていくらでもメニューをあげることはできます．

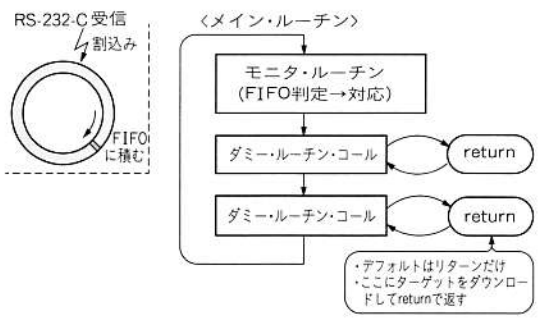
モニタ・プログラムの構成としては，RS-232-C の受信データを FIFO バッファに積むまでを割込みルーチンで処理するとして，この FIFO データの内容を判定して処理するモニタ本体部分をどのように実行するか，という方法がいくつか考えられます．

図Bはモニタとしてメイン・ルーチンがループして，その中にダミーでサブルーチン・コールを入れておくものです．ターゲット・プログラムをダウンロードしたときは，このコール先をターゲットに変更することが，実行コマンドに対応します．この方法はターゲットに無限ループのようなバグがあるとリセット以外に回復できません．

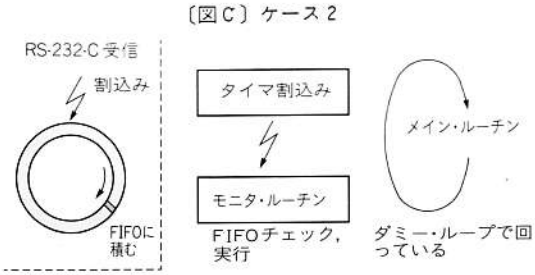
図Cでは別に設けたタイマによる割込みでモニタ・ルーチンを起動し，FIFO の内容から処理を判定するものです．この場合，ターゲット・プログラム自体は本来のメイン・ループを構成し，その先頭にジャンプできます．ただし，ターゲットがさらに別の割込みを使ったり，タイマ割込みを禁止するような場合には問題となるので注意が必要です．

このようにモニタをいくつか作ってみると，いっそのこと，ターゲットのプログラム自体の一部として，モニタを〈隠しモード〉としてもってしまうほうが便利な場合もあります．信頼性向上のための自己診断機能と似てくるのですが，バックグラウンド・ジョブとしてステータスを RS-232-C で出力し，もしオプションで RS-232-C を受信すると，じつはダンプやステータス・モニタができる…というのは，のちのちの保守性にも効いてくるものです．

〔図B〕ケース 1



〔図C〕ケース 2



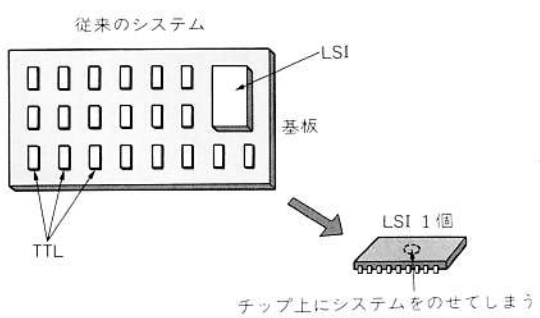
5．上級レベル：ASICシステム

マイコン技術者にとって，すでにASICは無視できない存在になってきた．そこで，本セミナーの最終章では，専用CPUボードの延長上に，ASICシステムを考えてみる．まずASICというシステム・オンチップの意義を概観する．つぎにゲートアレイの例で，① CADによる回路設計，② テスト・パターン設計，③ シミュレーションで検査，というASIC開発の流れを追ってみる．スタンダード・セルの場合についても検討し，メガセルやスーパーセル，コアCPUという最近の潮流にふれる．最後に，マイコン・システム設計の究極のかたちとして，オリジナルCPU開発の夢を語っていく．

（編集部）

さて，前章の専用CPUボードの上に，よりローコスト・ハイスピードのマイコン・システムの階層は存在するでしょうか．じつは，ここからが本番，といえるほど，その先の世界は大きく広がっているのです．ハードウェアとソフトウェアの領域は限りなく近づき，マイコン技術の真髄へと接近遭遇することになります．もちろん，ここから先の〈ローコスト〉とは，それだけの〈数量〉を前提としなければ実現できません．しかし最近の傾向として，この数量的な要因が，より小規模・パーソナルへと移行しつつあるので，「大手メーカでないから私には関係ない」と避けるマイコン技術者がいれば，その人は確実に時代から取り残されることになるでしょう．

〔図5.1〕システム・オンチップ

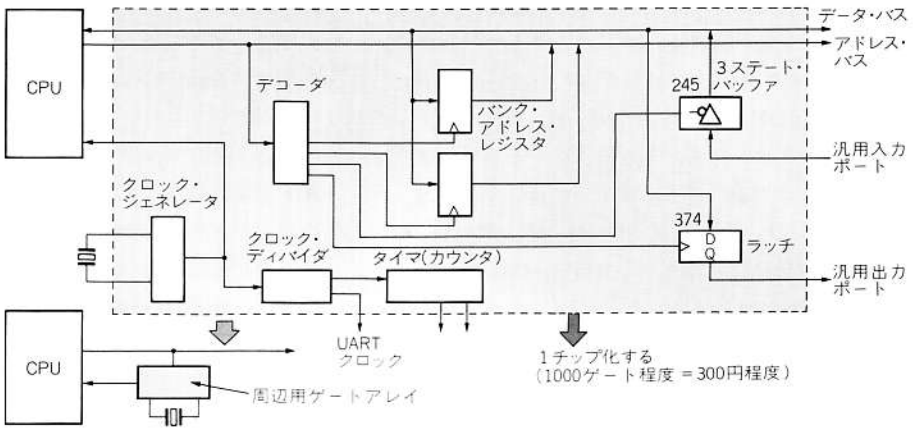


システム・オンチップの意義

マイコン・システムの設計者が，本当の意味でハードウェアを設計できるのが，ASICという新領域の半導体分野です．ここでは，CAD環境のサポートによって，半導体のレイアウト・プロセスといった階層の知識を必要としないで，カスタム機能のLSIを，かなり自由に開発できるようになっています．これは，システムをボード・サイズからチップ・サイズへと進化させるもの，という意味で，とくにメーカ側が〈**システム・オンチップ**〉という合言葉で，さかんに宣伝しているものです(図5.1)．そのLSIのタイプが，**ゲートアレイ**か**スタンダード・セル**か，というメーカの都合による区別は，設計するユーザの側ではあまり気にする必要はありません．CAD上で設計された回路を，専用のLSIにしてくれる便利なブラックボックスとして各LSIメーカを位置付けてしまえばいいのです．とくに最近の傾向として，この両者の境界は相当にあいまいになりつつあり，いずれ消えてしまうかもしれません．

マイコン・システムに関連したASIC*としては，CPU周辺のこまごましたICを，1チップのゲートアレイに全部入れてしまう，というのがもっとも単純な発想です(図5.2)．アドレス・デコーダ，DRAMコントローラ，メモリ・バンク・レジスタ，クロック・ジ

〔図5.2〕
CPU 周辺のシステム・オンチップ例



コラム 5.1

システム・オンチップの歴史：メーカ対応のいろいろ

まだCPUという標準LSIと，ゲートアレイというカスタムLSIとが完全に別種の半導体であったころ，なんとかこの二つが一緒になってくれないものか，とメーカ各社の担当者に勝手な希望を話していたのがほんの3年前，というのも考えてみれば驚くべきことです．このシステム・オンチップには二つのアプローチがあり，

① スタンダード・セル内のCPUコア

② 1チップ・マイコン内のゲートアレイ

という，よく考えないとピンとこない，じつは正反対方向からの別方法があります．もっともこのどちらであっても，ユーザにとっては，CPUの機能とカスタムのロジックとが同じチップ上に搭載されたLSI，と見ることができます．

ところが，大手メーカにとってはこの差はもちろん，そもそも標準のCPUとカスタムのASICとではプロセスもラインも，さらに事業部すら違う，というような壁がいくつも立ちはだかっていたようです．ASICで先行したメーカほど，ASIC事業を軌道に乗せるために専門の事業部として独立していて，標準LSI部門とのデータのリンクという点では腰が重くなりました．そして歴史がある分，大型機を中心とした古いLSI開発環境なので，CADやシミュレータの柔軟性も少なかったようです．

一方，後発の第2グループのASICメーカでは，機動性とユーザ・サポートを武器にして，さらに最新の強力なLSI開発環境を導入して，このシステム・オンチップ分野に参入してきました．スーパミニコン・EWSベースの最新のLSI設計ソフトは，CPUやメモリをたんなる機能ブロックとして処理できる能力をもっていたのです．小規模なゲートアレイを載せた1チップ・マイコンはセイコーエプソンから，コアCPUの載るスタンダード・セルはリコーから，早々と登場してきたのに対して，先行ASICメーカからの発表は遅れに遅れました．

もちろんここには，アメリカのCPUをそのままセカンド・ソースとして生産してきた国内大手メーカが，いざとなったらロイヤリティの問題でいじめられて困った，という事情もあります．あるいは，メモリを中心にした標準LSIの市場のほうがはるかに巨大であり，せっせとミクロン・ルールを進歩させ，工場をどんどん建設して，「汎用品で儲ける」という優先事項があったのも事実です．ASICを作りたいユーザは，メーカの指定形式に合わせれば作ってやるよ，というような今も残っている体質も，ここで身についたのでしょう．

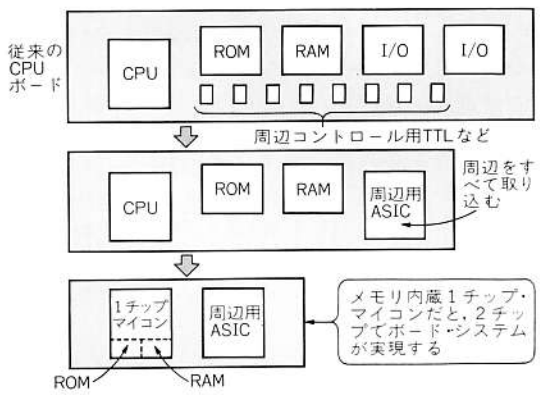
一方，CPUの故郷アメリカでは，実際の半導体生産技術で日本に大きく立ち遅れてしまい，上の二つの両方のアプローチでシステム・オンチップASICを作れますよ，と景気のいいことをいっていたある海外メーカは，その後いつになっても実際に受注開始を宣言できずにいます．おそらく設計はできるのに，レイアウト/マスク/プロセス/品質/歩留まり/コストなどのどこかで問題がクリアできないのでしょう．儲かっているのは，CPUなどのロイヤリティを切り売りしている老舗メーカと，日本やNIESで使用されている最新のLSI設計ツール・ソフトを供給しているシステムハウス，そして泥試合の裁判に関係する弁護士・特許事務所などです．

半導体技術の基本思想とLSI開発環境のソフトは相変わらずアメリカにあり，一方メーカとしての生産はしだいにNIESに移っていくとすると，今後の日本のメーカは一体何をやって儲けていくのでしょうか．これはますます目を離せない，システム・オンチップ劇場の第2幕，というところです．

ェネレータ，汎用ポートなどの，個々に組んでもあまり規模の大きくない回路を，すべて1チップに収容できます．うまくシステムを設計すると，CPUボードに載る部品は，CPU，ROM，RAM，ゲートアレイの4チップだけ，という非常に美しいボードが誕生します．さらに，メモリ内蔵の1チップ・マイコンを使えば，部品数は二つ，という極限も考えられます(図5.3)．ただし，FDCやUARTのような高機能・高密度LSIについては，無理にゲートアレイで設計しても，機能を実現するためにゲートを食うばかりで，コスト・メリットがあまり出てきません．半導体メーカの標準品である汎用LSIは，時間とノウハウをかけて設計されたものであり，さらに量産効果によって十分に低コストだからです．したがって，コンパクトに完成されている汎用LSIをうまく使用し，汎用LSI以外の部分をゲートアレイ化するとして，その場合のメリットが開発費用を回収する数量の見通しに達するかどうか，という設計戦略が必要になります(図5.4)．

また，汎用のDSPチップもずいぶん低価格になりましたが，これでも高くて使えない応用分野は随所にあります．たとえば，オーディオのサラウンド・プロセッサなどは，リアルタイムに高度なディジタル信号処理を行っています．もし，アマチュアがこれを自作しようとすると，高価なDSPを使用したシステムとして，設計・製作することになります．ところが低価格競争のプロの業界では，この信号処理部分は，超ローコストのカスタムLSIで実現しています．家電分野は数量が多いので，ゲートアレイでなくフルカスタムですが，コスト・ダウンのためのハードウェアのカスタム化という発想においては同類です．あるいは，最近の計測器はディジタル化が進んでいますが，ここでも多くの信号処理LSIが使われていて，数量・コストの均衡する部分には，かなりのセミカスタムLSIが活用されています．

〔図5.3〕究極のCPUボード



NIES諸国の台頭というのも，ASIC化の流れを加速する要因になっているかもしれません．LSIチップ上にハードウェアが集積されると，回路の解析やデッド・コピーは非常に困難になります．互換機・特許の紛争や，新商品→同類追従商品の開発競争，という日本の美しくない風土の中では，ASICのもつ〈秘密保持〉という特性も重要なのでしょう．たとえば，CPUのソフトでいくらデータを暗号化しても，プログラムを逆アセンブルして解読すれば，このプロテクトは無力化できます．ところが，ゲートアレイ内のハードによって暗号化するとか，ゲートアレイ内のIDチェック回路によるとかの方法は，かなり強力な保護レベルを実現できます．もちろんソフトのプロテクト同様，人間の作ったものは人間によって，結局は突破できるのですが，セールス・ポイントの一つであるのは確かなようです．

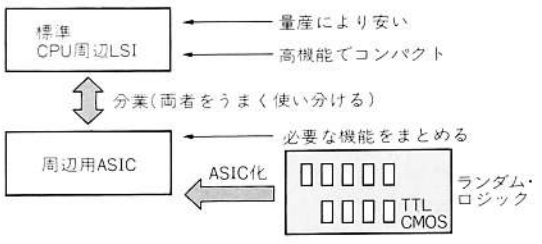
ゲートアレイ

ASICの開発環境はメーカによって異なりますが，ある程度の類型化・標準化も進んでいます．基本的な流れとしては，

① CADで回路を設計する
② テスト・パターンを設計する
③ シミュレーションで検証する

という作業を，シミュレーションがOKとなるまで繰り返す，ということになります(図5.5)．以前は支援環境が貧弱で，メーカ側の技術者がマンツーマンで対応していましたが，現在ではユーザの机の上で，完全に独立した設計作業が行えるようになっています．ツー

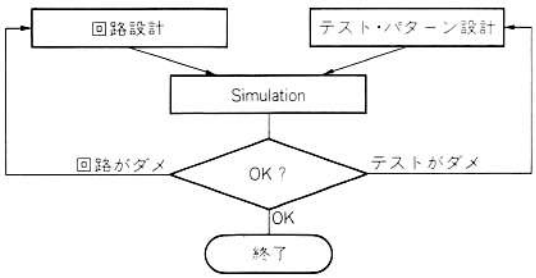
〔図5.4〕CPU周辺のASIC化の考え方



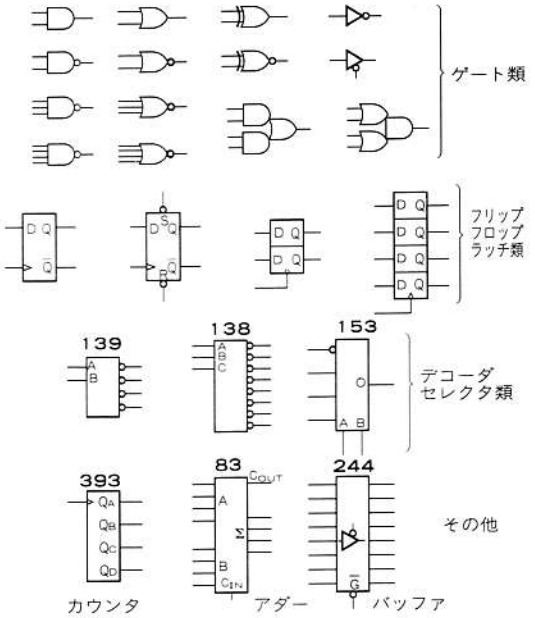
ルはここでもパソコンで，**回路規模が1万ゲート程度を越えれば，EWSのほうが高価であっても有利**になるようです．以下，まずは標準的なゲートアレイの例について，開発作業の流れを簡単に追ってみましょう(p.250 Appendix 5.2「ゲートアレイの設計ポイント」参照)．

第1段階の回路設計は，CADでCPUボードの回路設計をした人ならば，何も目新しいものはありません．メーカがサポートするCADツールによって，設計効率や操作性は千差万別なのですが，基本的には部品を並べて結線する，というものです(**図5.6**)．ただし，ICの回路では汎用のTTLやCMOSだった部品が，LSIメーカに登録された**セル・ライブラリ**の中からのみ，選択できるようになります．もっとも最近のライブラリは，簡単なゲートばかりでなく，TTLに相当する機能のセルを何十も用意しているので，多少の組合せによって，どんな回路でも容易に実現できます(**図5.7**)．ここで，ASICの回路設計CADに特有な点としては，つぎの段階へのインターフェースの関係で，すべての

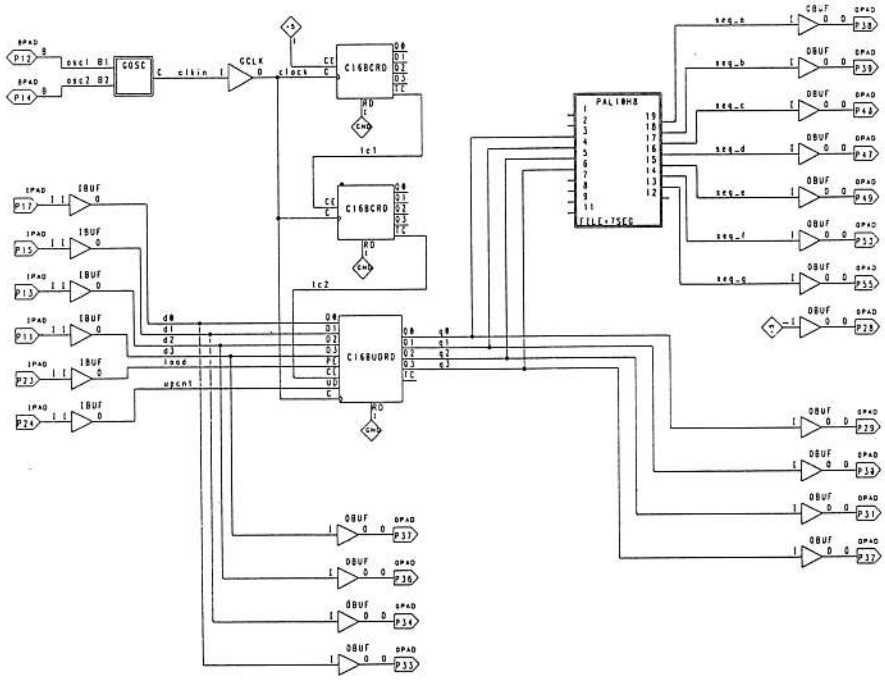
〔図5.5〕ASICの開発フロー



〔図5.7〕ゲートアレイのセル・ライブラリの例



〔図5.6〕
CADによる回路設計の例



セル，ライン，ノード，端子などに固有の名前を付けること，図面を見やすくするために意識的に図面の階層を分割すること，未定義の入力端子のようなエラーがすべて解消されないと，つぎの段階に進めないこと，などの点があります．形式的なチェックをパスした回路図データは，のちのシミュレーションに使用される形式のデータに変換されます．また，ゲートアレイでは，チップ上の使用可能ゲート数に対する占有率なども計算され，規定以上の数値だとエラー・メッセージが出ます．回路の全体にわたる，個々のゲートの入出力のファンイン，ファンアウトもチェックされ，エラーがあれば設計の修正が要求されます．ピン名・信号名のリストなども自動生成されます．これらの処理のかなりの部分は，MS-DOS のバッチ・プログラムのように自動化できるので，検証から回路修正へと戻るループは，慣れてくると意外に速いペースで回転します．

第 2 段階では，LSI のテスト・プログラムを作成します．これは，ハードを確認するという機能としては，

〔リスト5.1〕LSI のテスト・プログラムの例

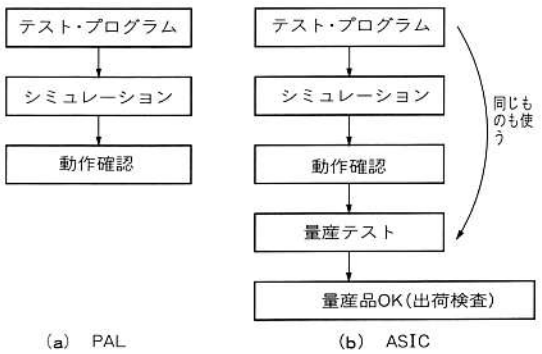
```
>type sample.tst

*-----------------------------------------------
*  Sample Test Pattern
*-----------------------------------------------

VECTOR: CLK(1*2),@1,TEST(1*6),@1,RESET,CT1,CT2,CT3,@1,
        DATA(4*4),@1,ADDA(4*2),@1,ADDB(4*2),@1,ADDC(4*2),@1,
        ADDZ(4*5),@1,CONT(4*2),@1,PT1,PT2,PT3;

PX 000000 1111 ZZZZ 00 00 ZZ XXXXX XX XXX
   010000 1101                          L
   000000                               L
   P10111       0005                    H
   010111                               H
   P01111       0003                    L
   001111                               L
   P00010       0034                    H
   000010                               H
                ZZZZ                    L
                                        L
             0      03 A0    003A0      H
                        01        01 H
                           XXXXX XX L
                                     L
                    03 01    00301    H
                        00        00 H
   P00111                  XXXXX XX L
   000111                           L
                    03 02    00302    H
                        00        00 H
                           XXXXX XX L
                                     L
             1      FF FF ZZ 0FFFF     H
                                       H
                                    XXXXX     L
                                              L
                 0      02 00     00200      H
                              C0          C0 H
                                 XXXXX XX L
                                          L
                       02 86     00286      H
                           FF          FF H
                              XXXXX XX L
                                       L
                       02 87     00287      H
                              01          01 H
                              XXXXX XX L
                                       L
    P00010     0 0000 03 A0     003A0      H
    000100                 00          00 H
                 ZZZZ          XXXXX XX L
                                       L
                       02 00     00200      H
                              D6          D6 H
                                 60000 XX L
                                          L
                       02 08     00208      H
                              FD          FD H
                                 A0000 XX L
                                          L
REPEAT 100:
                 1      FF FF ZZ 0FFFF     H
                                           H
                                 00000     L
                                           L
REP END:

ENDVECT:
```

〔リスト5.2〕シミュレーション結果の例

```
>type sample.sim

Signal      | C CC CCCC DDDDDDDD C AAAAAAAAAAAAAAAA PP
            | L TT TTTT 01234567 T 0123456789111111 TT
Step        | K 12 3456          3           012345 12
------------------------------------------------------
40          |
    10.7ns  | 0 1X 1111 ZZZZZZZZ 0 0011111111111111 11
    12.1ns  | 1 1X 1111 ZZZZZZZZ 0 0011111111111111 11
    17.0ns  | 1 1X 1111 XXXXXXXX 0 0011111111111111 11
41          |
     2.1ns  | 1 1X 1111 X1X11XX1 0 0011111111111111 11
     2.4ns  | 1 1X 1111 01011001 0 0011111111111111 11
42          |
    10.7ns  | 1 1X 1111 01011001 0 0011111111111111 10
    11.8ns  | 0 1X 1111 01011001 0 0011111111111111 10
    16.9ns  | 0 1X 1111 0Z0ZZ00Z 0 0011111111111111 10
    17.3ns  | 0 1X 1111 ZZZZZZZZ 0 0011111111111111 10
    81.8ns  | 0 1X 1111 ZZZZZZZZ 0 1011111111111111 10
44          |
    10.7ns  | 0 1X 1111 ZZZZZZZZ 0 1011111111111111 11
    12.1ns  | 1 1X 1111 ZZZZZZZZ 0 1011111111111111 11
    16.8ns  | 1 1X 1111 000ZZZZ0 0 1011111111111111 11
    17.2ns  | 1 1X 1111 00011110 0 1011111111111111 11
46          |
    10.7ns  | 1 1X 1111 00011110 0 1011111111111111 10
    11.8ns  | 0 1X 1111 00011110 0 1011111111111111 10
    16.9ns  | 0 1X 1111 000ZZZZ0 0 1011111111111111 10
    17.3ns  | 0 1X 1111 ZZZZZZZZ 0 1011111111111111 10
    81.8ns  | 0 1X 1111 ZZZZZZZZ 1 0101100100011110 10
48          |
    10.7ns  | 0 1X 1111 ZZZZZZZZ 1 0101100100011110 11
    12.1ns  | 1 1X 1111 ZZZZZZZZ 1 0101100100011110 11
    16.8ns  | 1 1X 1111 0Z0ZZ00Z 1 0101100100011110 11
    17.2ns  | 1 1X 1111 01011001 1 0101100100011110 11
49          |
                1 1X 1111 11011101 1 010[illegible]
                     1111 10010101 [illegible]
```

PALの設計ツールの中のテスト・プログラムと同様のものです(**リスト5.1**)．ただし，PALプログラマのテスト・パターンは，できあがったPALの論理動作を，あらかじめシミュレーションして確認するだけのものでした．これに対して，ASICのテスト・パターンは，動作の期待値を記述・確認するシミュレーションの性格とともに，LSIメーカとユーザがこの条件をもって完成品と認定する，という品質保証の基準でもある，という点がよりシビアです(**図5.8**)．したがって，入力ピンのあらゆる組合せを想定し，設計した回路全体の細部まで，LSIの外からテストできるような配慮が必要です．なにせASICの場合，もしトラブルが発生してもハードはチップの上なので，オシロのプローブを当てることは不可能で，従来のデバッグ手法は考えられません．ここの段階でのテスト・パターンと，つぎの段階でのシミュレーションが，唯一のデバッグのチャンスなのです．また，論理的チェックばかりでなく，内部ゲートの遅延を含めた，LSIの動作マージンについても，ここでテスト・プログラムのパラメータとして設計し，シミュレーションによって検討します．この段階での作業には，

〔図5.8〕PALとASICにおけるテストのちがい



① **同じ検証内容を何パターンで記述できるか，という経験による圧縮テクニック**

② **システムのどこがポイントで，とくに念入りなテストが必要であるか，という確実性の視点**

の，二つのノウハウがあるようです．

第3段階は，設計された回路データとテスト・データによって，LSIのシミュレーションを行う部分で，ここでの結果は前段階へとフィードバックされます(**リスト5.2, 5.3**)．ひとくちにシミュレーションといって

〔リスト5.3〕
期待値とシミュレーション結果を比較した例

```
>type sample.cmp

     CCCCCCCCAAAAAAAAAAAAAAAAAAAADDDDDDDDTTTTTPPP
     OOOOOOOOBBBBBBBBBBBBBBBBBBBBBBBBBBBBEEEEETTT
     NNNNNNNN11111111119876543210765432105SSSSS123
     TTTTTTTT9876543210                   TTTTT
     76543210                             43210

     XXXXXXXXXXXXXXXXXXXXXXXXXXXXHLLHLHHLXXXXXHHL    103  16932
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXLHHLHLLHXXXXXHHL    104  17098
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    105  17264
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    106  17430
     XXXXXXXXLLLLLLLLLLHHHHLHLLLLHHHHHHHHXXXHHHHL    107  17596
     XXXXXXXXLLLLLLLLLLHHHHLHLLLLHHHHHHHHXXXHHHHL    108  17762
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    109  17928
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    110  18094
*    XXXXXXXX///////HLLHLLLHHLHLLXXXXXXXXHHH/HHL    111  18260 ←
*    XXXXXXXX///////HHHHHHHHLHHLHXXXXXXXXHH./HHL    112  18426 ←
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    113  18592
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    114  18758
     XXXXXXXXLLLLLLLLLLHHHHLHHLLLLHHLHLLHXHHHHHHL    115  18924
     XXXXXXXXLLLLLLLLLLHHHHLHHLLLLHHLHLLHXHHHHHHL    116  19090
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    117  19256
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    118  19422
*    XXXXXXXXL//L/LLHLHLLLLHHLLHLXXXXXXXXXHHHLHHL    119  19588 ←
     XXXXXXXXLLLLLLLLHHHHHHHHHHHHXXXXXXXXXHHHLHHL    120  19754
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    121  19920
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    122  20086
*    XXXXXXXXL//L/LLHHLLHLLLHLLLXXXXXXXXXHHHLHHL    123  20252 ←
*    XXXXXXXXL//L/LLHHHHHHHHHHHHXXXXXXXXXHHHLHHL    124  20418 ←
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    125  20584
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHL    126  20750
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXHHL    127  20916
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXHHL    128  21082
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHH    129  21248
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHHXXLHH    130  21414
     XXXXXXXXLLLLLLLLLLLLLLLLLLHLHLHLHXHHHHHHH      131  21580
     XXXXXXXXLLLLLLLLLLLLLLLLLLHLHLHLHXHHHHHHH      132  21746
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    133  21912
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXL    134  22078
```

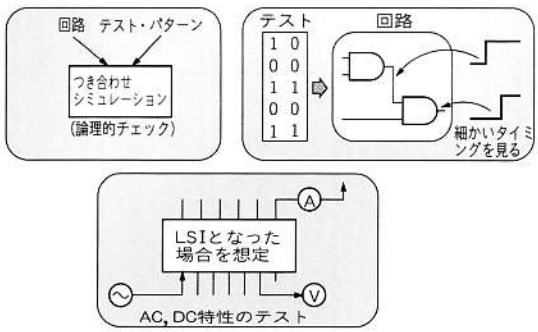
期待値とシミュレーション結果の不一致の部分がある

も，じつはいろいろなレベルのものがあります(**図5.9**)．単純に回路とテスト・パターンとの論理チェックをするもの，個々のゲートの遅延時間の最大・最小条件によってマージンを検証するもの，さながらブレッド・ボードをオシロで見る感覚で，回路の信号の状態変化をゲート単位で対話的に追えるもの，長い時間スケールでの信号状態をテストするもの，LSIとしてのAC特性・DC特性をテストするもの，などがあります．また，つぎのレイアウト作業の段階まで行う場合には，レイアウト後の実際の配線状態のもとでの，実配線長の動作シミュレーションもあります．そして，ここをパスすると，あとはLSIメーカが試作したサンプルを待つ，という，楽しくも心臓に悪い期間がやってくるのです．

さて，この段階でシミュレーションの警告メッセージが出た場合，その原因が以下のようにいろいろ考えられるために，とりわけ注意が必要です(**図5.10**)．

① 回路にミスがあってテスト・パターンと合わない

〔図5.9〕ASICのシミュレーションのいろいろ



② テスト・パターンにミスがあって回路と矛盾している

③ シミュレーションの条件設定に問題があった

④ 回路やテストのデータ変換作業の指定にミスがあった

⑤ ASIC開発ツールというソフト自体のバグ(！)

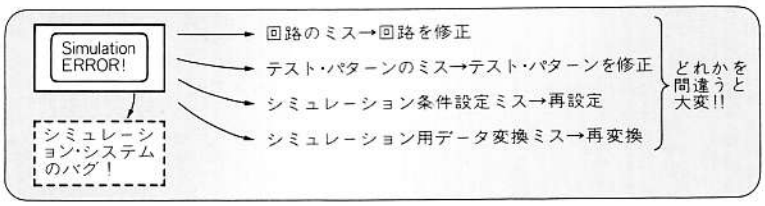
ここに並べた原因は，すべて筆者の経験した例ですが，CPU回路のトラブルの原因がソフトかハードであるのに比べて，問題を発生させる要因が，はるかに広範囲になるのが特徴です．ここでの作業とは，パソコンのプログラムである開発ツールという世界の中で，ソフト上で想定したハードウェアと，ソフト上で想定したテスト・パターンとを，まさに「絵に描いた餅」として突き合わせているのです(**図5.11**)．

現実にLSIが目の前にあって，現実の信号によって誤動作していれば手の出しようもあるのですが，そうはいかないところが難所なのです．安易にシミュレーション結果に合わせてテスト・パターンを変更したり，結果をパスさせるように回路を変更したりすれば，その後の**2次災害**は致命的になります．ここは，マイコン・システム技術者としての自分を信じて，最大限の努力をする，としかいいようがありません．全体の構成と関係づけながら，あらゆる可能性に目を光らせ，冷静にシステムの細部を詰めていく，という地道な姿勢がなにより大切なのです．

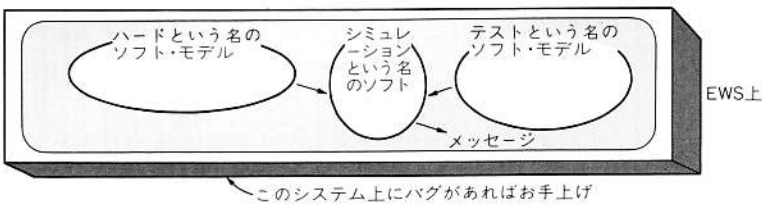
スタンダード・セル

スタンダード・セルのライブラリの，TTL相当や各

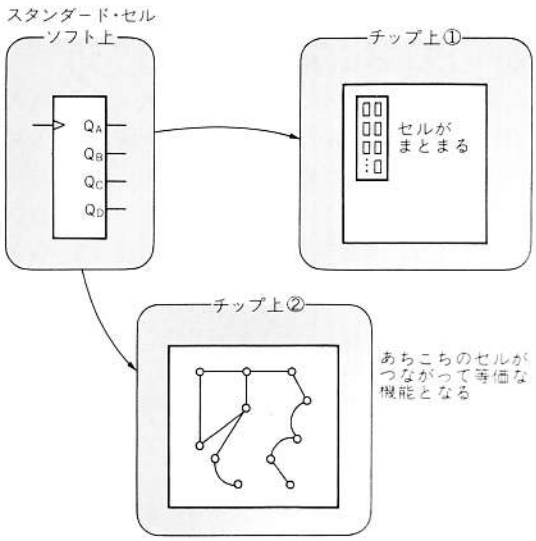
〔図5.10〕
シミュレーションのエラー・メッセージ



〔図5.11〕
シミュレーションに実体はない！



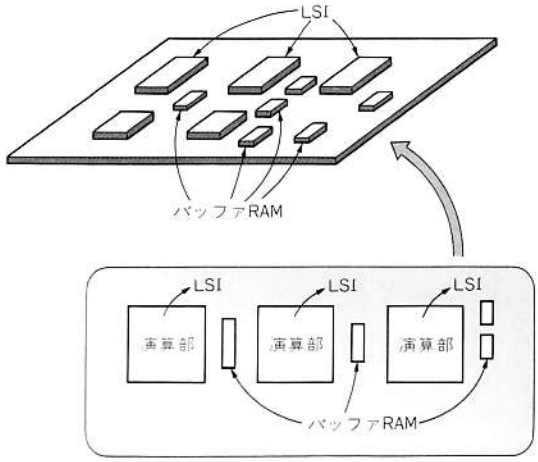
〔図5.12〕ファンクション・セルの実現方法



種のマクロセルを実現する方法には大きく2種類あって，実際にそのブロックを構成するセルが物理的にまとまっているものと，ゲートアレイと同様に，散在するゲートを等価的に結びつけるものとがあります(図5.12)．これはユーザにとっては，どちらでも気にならない部分で，かつては各メーカが，とにかくこのライブラリの数を100種類，200種類と競い合っていました．このようなLSIでは，チップは平坦なロジック用ゲートの海という外見になります．そして，大規模なディジタル信号処理回路をこのレベルのASICで設計すると，DSPシステム的な動作に必要な，小容量のバッファRAMが，LSIの外部に何個も置かれる，という形態しか採れませんでした(図5.13)．基板上には数個のゲートアレイと，10数個の高速RAMがごろごろしていました．

ところが次第に，特定の機能ブロックを独立にまとめて，普通のロジック部分と並べてチップ上に配置させたものが，スタンダード・セルのニューウェイブとして登場しました．たとえば，無理にゲートを使って構成していたRAMに比べて，RAMブロックの専用マクロセルならば，十倍以上のゲート効率で，チップ上に吸収できるようになりました．

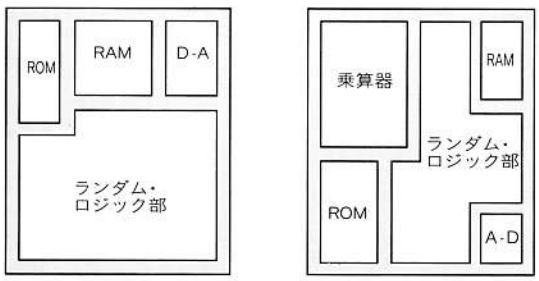
〔図5.13〕かつてのDSPシステム



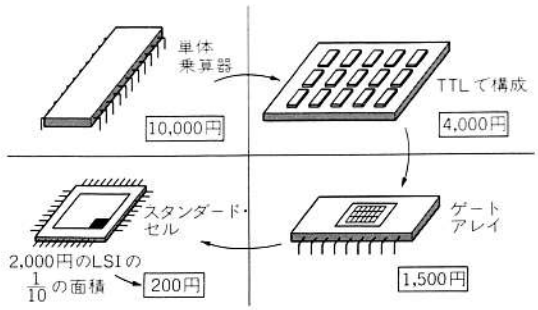
このような流れを受けて，いまやスタンダード・セルは，メガセル，スーパセルの時代です．各LSIメーカのセールス・ポイントは，もはや小さなセルの種類ではなく，ROM，RAM，乗算器，A-D/D-Aなどの大規模なブロックです(図5.14)．これらの機能は，専用に設計されてコンパクトに提供されているので，ランダム・ロジックでユーザが構成するよりも，かなりチップ上の面積という点で有利になっています．

このコスト・メリットを，具体的に比較してみましょう(図5.15)．かつて筆者の開発した，あるディジタル信号処理LSIの例では，一定のビット精度・速度の

〔図5.14〕現在のスタンダード・セル



〔図5.15〕ASICのコスト・メリット(およその例)



乗算器を，独立のメガセルとして使用しました．そこで，基準となるこのコストを，完成したLSIの単価と，チップ上で乗算器が占める面積の割合とから，とりあえず比例計算で仮定します．一方，この実験のためにDSPメーカから購入した，同等の乗算器LSIのサンプル価格を比較してみると，約50倍の開きがありました．また，この乗算回路を高速TTLで組んだ場合には，およそ20倍程度になります．そして，通常のゲートアレイで実現した場合を仮定すると，世間の相場と過去の経験から推定して，5倍から10倍といったオーダのコストになると思われます．もちろん，ここではスタンダード・セルの開発費用や開発期間を考慮していませんし，システムとして1チップに入らずに分割された場合の影響，というものも除外しています．それでも，このコスト比は，ASICの魔力というか魅力の感覚を，十分に考えさせられる材料だと思います．

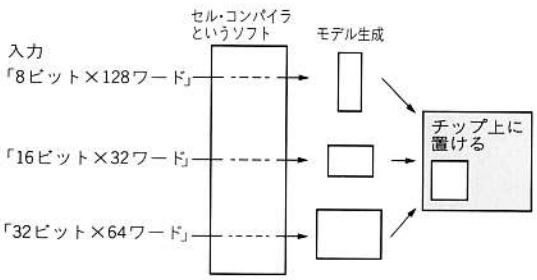
スタンダード・セルに搭載できるROMやRAMなどのメモリにしても，自由度は相当に向上しています．まず，ビット幅とワード数を一定範囲内で指定すると，効率的なメモリ・ブロックを，新規のライブラリとして自動生成してくれるセル・コンパイラというツールがサポートされてきました(**図5.16**)．

また，デュアル・ポートRAM，トリプル・ポートRAMや，よりコンパクトな同期式メモリを指定できる環境もあります．あるいは，チップ上のROM部分だけを変更する場合，1チップ・マイコンのマスク・チャージ程度のコストで対応できるというメーカもあります．メモリ・ブロックに限らず，セル・コンパイラの概念は，自由度をもったDSPブロックの自動生成，という方向にも発展しています．

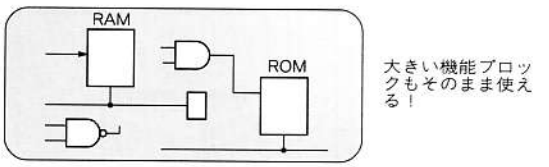
実際の開発作業としては，スタンダード・セルの場合であっても，前述のゲートアレイの場合とそう変わりません．回路図に使用できるメガセルが増えますが，この部分はブラック・ボックスとして，普通の回路図のRAMなどのように扱うだけです(**図5.17**)．シミュレーションのときにメガセルがどう振舞うか，というシミュレーション・モデルは，メーカのほうで用意されています．外見上の違いとしては，一般にゲートアレイよりも大規模なシステムとなる場合が多いので，図面の階層がより深くなって，ページ数が増えるくらいでしょう．

回路設計のテクニックとしては，等価で使用ゲート数の少ないものへ，セルの組合せや信号の論理を変換するとか，ラッチ・タイプを使い分けるとか，バス構成にするかセレクタにするか，といったノウハウがあります(**図5.18**)．従来は，ファミリを構成するチップの種類も少なかったので，このような詰め込みテクニ

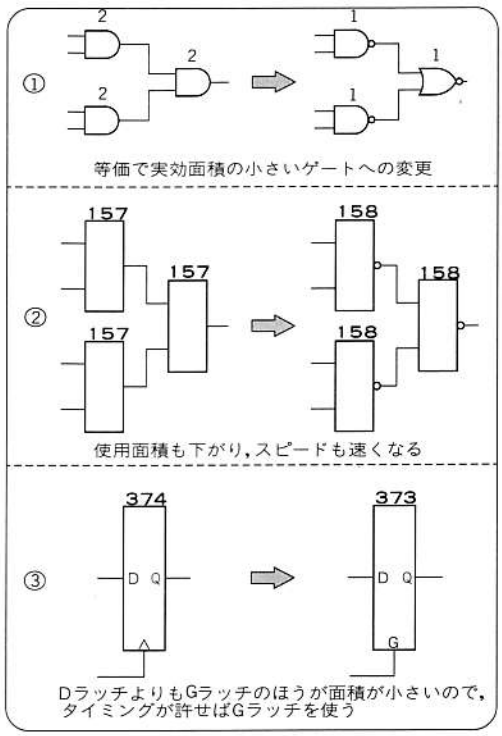
〔図5.16〕セル・コンパイラ



〔図5.17〕スタンダード・セルの回路設計



〔図5.18〕回路設計のテクニックのいろいろ

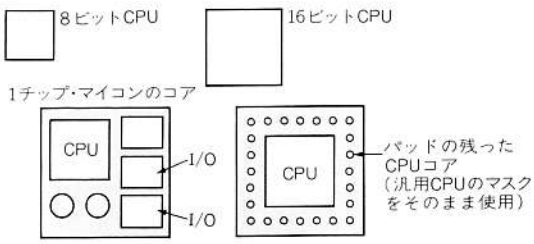


ックも重要でした．ところが，次第に技術の進歩によって吸収されて，この点ではあまり細かいことを考える必要はなくなってきています．むしろ，LSI 設計のポイントがあるとすれば，LSI 化する場合のテスタビリティを考慮した回路設計技術，すなわちテスト・パターン設計と結びついた回路設計テクニックなのではないか，と筆者は考えています．

コア CPU

さて，メガセル，スーパセルという流れは当然，CPU というブロック・セルに到達します．CPU ブロックのセルは，とくに **CPU コア**とも呼ばれます．ファミコンの心臓部が，8 ビットの CPU コアをもったカスタム LSI である，というのは有名な話でしょう．現在のところ，実際に ASIC に使える環境にある CPU コアは，国内ではまだ十数種類ぐらいしかありません．それでも，知的所有権問題によってオリジナル CPU へと移行している国内半導体メーカ各社の最近の事情と，この ASIC ブームの動向とを考えれば，いずれ CPU コアの選択の可能性は，相当に増えることは間違いないでしょう．8 ビットばかりでなく 16 ビットのコアや，周辺をもつ 1 チップ・マイコン全体のコア，さらに海外メーカのユニークなコアも登場しつつあります(**図5.19**)．マイコン・システム構築の発展としてここまで来れば，**コア CPU** という名の，まさに〈システム・オンチップ〉の LSI の開発は，もう容易に手の届くところにあります．

〔図5.19〕CPU コアのいろいろ



大規模セルによるスタンダード・セルを設計した経験があれば，CPU コアが加わったからといって，とくに身構えることはありません．CPU とはいっても一つの巨大なファンクション・セルであり，特定の入力に対して相応の動作を行うだけのものです．ただし，テ

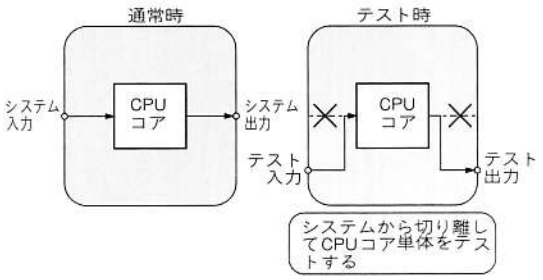
コラム 5.2

LSI と特許

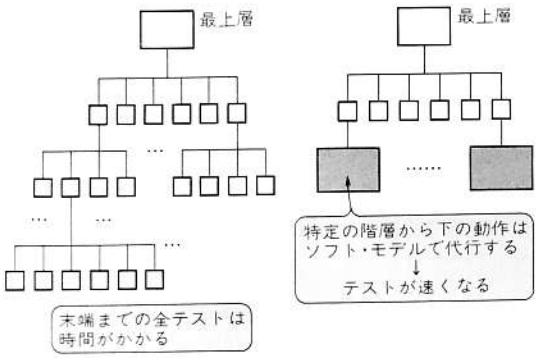
工業所有権，とくに特許の問題というのは，システム屋にとっては頭の痛い問題です．せっかく美しいシステムを考案しても，よくよく調べてみたらすでに同類の出願・公開があった，という場合は少なくありません．LSI として開発がスタートしてしまってから，という段階では損害が膨大になります．かといってスピード競争のこの時勢に，十数年前からの全特許を調査してからシステム化する，というのも悠長な話です．さらに日本の特許制度は国際的にも異端児であり，いずれ変更になるという不定要素すらあります．特許庁のシステムも先進国とはいえない寂しさで，儲けているのは特許サービス業と弁理士ばかり，というのが実状？(もっともアメリカの特許紛争でも，最終的に儲けるのは弁護士だけなのですが)．

あるシステムをうまくコンパクトにまとめて，オリジナル LSI 化したいというのは，LSI 設計に関係する技術者の共通の目標でしょう．ところがここでも，面白い現象があります．それは，「その LSI を買うのは OK でも，使うのはダメ」というケースです．たとえば，ある A 業界の用途に非常に有効な LSI というものを，その世界に興味のある別の B 業界の人が開発したとします．この場合，この LSI を使って，個人の趣味としてシステムを製作することは何の問題もありません．ところが，A 業界の範疇の特許出願を詳細に調査してみると，じつはこの LSI が抵触するものがいくつもあり，A 業界のメーカがこの LSI を購入して製品に使用することはできない，ということになるのです．筆者の経験でも，かなり意外な国籍の海外メーカや，国内のベンチャなどで，このような売込みを何度も見ています．ここでは，システムの発想がじつに素直で，特許検討をしていないのが一目瞭然である LSI によく出会います．仕方がないので，「これはアマチュアしか使えないですよ」といって帰ってもらうのですが，たいていその後，そのメーカの名前はお目にかからなくなります．しかし，このような発想優先の LSI が，アマチュアの使用を前提として秋葉原あたりに並ぶようになると，それはまた別の面白さがあるような気もします．そんな時代にならないものでしょうか．

〔図5.20〕CPUコアのテスト・モード



〔図5.21〕階層シミュレーション

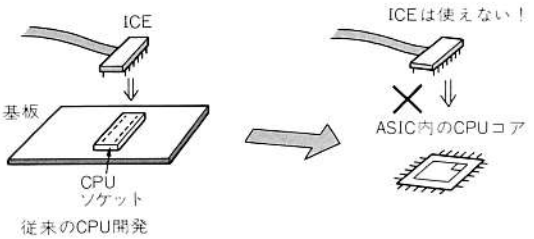


スト・モードにおいては，メーカがCPU単独のテストを行う必要があるので，テスト・モード時にはCPUコアの全信号線を切り替えて，外部端子からアクセスできるように設計する必要があります(**図5.20**)．これは，メモリやDSPなどのメガセルの場合にも，同様に要請されることです．普通は，CPUコアの実際のブロック・テストの内容はメーカ側で用意するので，回路設計の段階では，テスト・モードに入る部分だけを作ります．テスト・パターンも，テスト・モードへの入口を準備しておけば十分です．

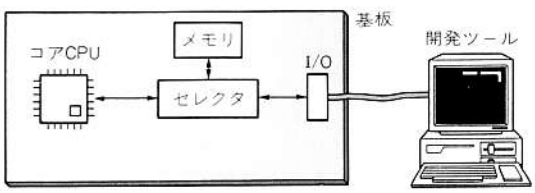
一方で，この段階のシステムに特有の現象も出てきます．まず，大規模セルを使用しない普通のスタンダード・セルのシミュレーションでは，歴史も古いので，検証を高速化するための方法がソフト/ハードの両面でいろいろと提供されてきました．たとえば，シミュレーション・エンジンと呼ばれるハードウェアや，ゲートアレイの環境と共有化された高速シミュレーション・ツールなどです．ところが，新顔のメモリ，CPUコアのような機能ブロックでは，これらの特効薬が使用できない場合もあるので，開発スケジュールの面で注意が必要です．また大規模セルは複雑なブロックなので，全ゲートを対象に指定した詳細シミュレーションは，予想以上に時間がかかります．このため，検証の対象を限定して，階層ごとのシミュレーションという手法を用いたりします(**図5.21**)．あるいは，1サイクルを100 ns程度としたDSPシステムでは，CPUコアの初期設定ルーチンをテスト・パターンとして少し組んだだけで，あっという間に何千ステップになる場合があります．ハード・ロジックに比べてCPUがいかに遅いか，という事実でしかないのですが，これもシステム設計の最初から考慮すべき点になります．

コアCPUのもう一つの重要な注意点は，CPUソフトの開発環境です．LSIのチップ上にCPUがあるので，たとえそのコアが汎用CPUとコンパチブルであっても，ICEなどは使えません．CPU自身に直接触れることはできない，という状況を正確に想定しなければなりません(**図5.22**)．外部にプログラム・メモリが置かれてそれをアクセスするシステムであれば，前に述べた方法のように，メモリを開発ツール側のシステムRAMと切り替える，といった装置を作れば開発可能です(**図5.23**)．また，ターゲットLSIのブレッド・ボードを製作し，そこで使いやすいICEによって開発する，というのも有効な方法です(**図5.24**)．それでは，CPUコアがオンチップのマスクROMで走る，というシステムのROMプログラムは，どのように開発するのでしょうか(**図5.25**)．これはもうパズルのような問

〔図5.22〕コアCPUの開発環境

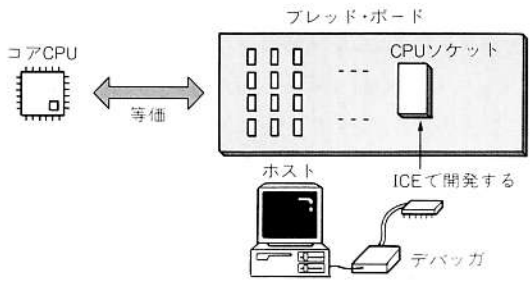


〔図5.23〕コアCPUのソフト開発例(1)



コアCPUがアクセスするメモリ(外部)にプログラムを書いてやる

〔図5.24〕コア CPU のソフト開発例 (2)



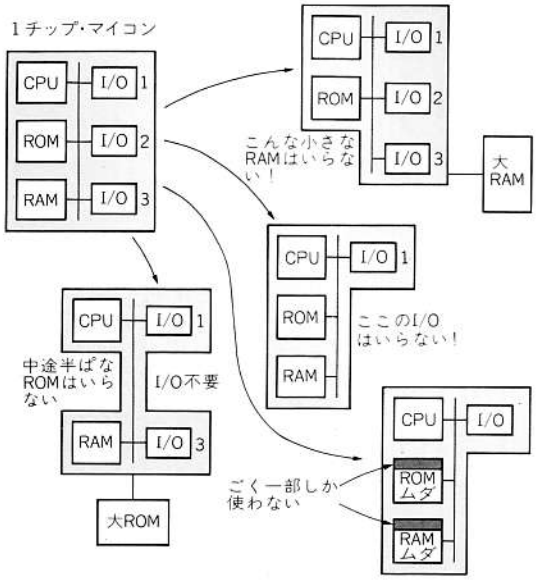
題です．ここまで来ると，ハードのためのソフトか，ソフトのためのハードか，という柔軟な発想が必要でしょう．ヒントはすでに十分に述べましたから，この〈開発環境の開発〉クイズの解答は伏せておくことにします．

オリジナル CPU

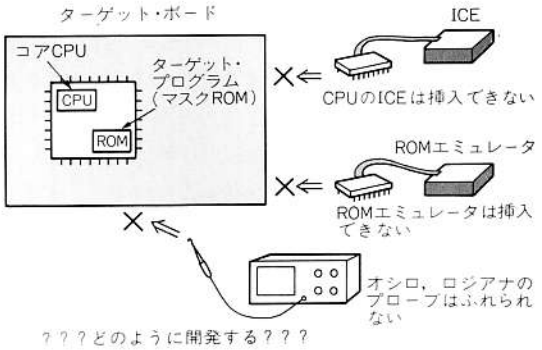
さて，長い道のりでマイコン・システムの技術を考えてきましたが，いよいよ終点です．ここまでの流れを受けると，もはや最後には，CPU そのものをオリジナルに設計してしまおう，という段階になります(もっとも，この部分だけは筆者にも経験がないので，想像と期待を込めて書いていきます)．

標準の CPU や 1 チップ・マイコンを使ったり，メガ

〔図5.26〕標準 CPU の不満な点



〔図5.25〕コア CPU のソフト開発例 (3) —— CPU コアがオンチップのマスク ROM で走るシステムの ROM プログラムはどう開発する？



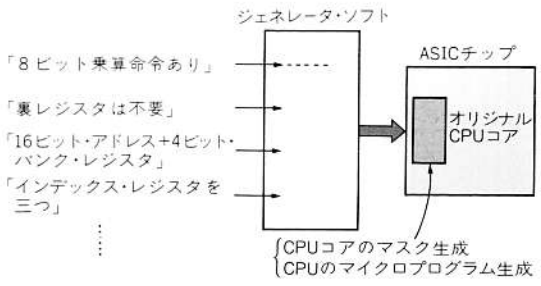
セル，CPU コア・タイプの ASIC を設計してみると，汎用 CPU というものについて，ちょうどボード・マイコンに対して感じたような不満が出てきます．つまり，

「帯に短し・たすきに長し」

です．たとえば，タイマは 3 本も要らない，A-D ポートは要らない，だからもっと安いバージョンの CPU が欲しいとか，マスク ROM は不要だから 2 倍の RAM が載らないかとか，命令数は 3 分の 1 でいいから，もっとセル・サイズの小さい CPU コアがないか，とかの注文です(**図5.26**)．

もちろん，ビットスライス CPU コアと DSP のマイクロプログラム手法によって，簡単な CPU を設計してみることは，現在のレベルでも可能です．しかしこれでは，マニュアル配線でフルカスタム設計された汎用 CPU に比べて，はるかに低密度の，コストの見合わない CPU にしかなりません．ここで想定しているカスタム CPU コアとは，必要な機能をメニューから選択すると，高級セル・コンパイラによって，自動的に生成されるものでなければなりません(**図5.27**)．もち

〔図5.27〕カスタム CPU コアのジェネレータ



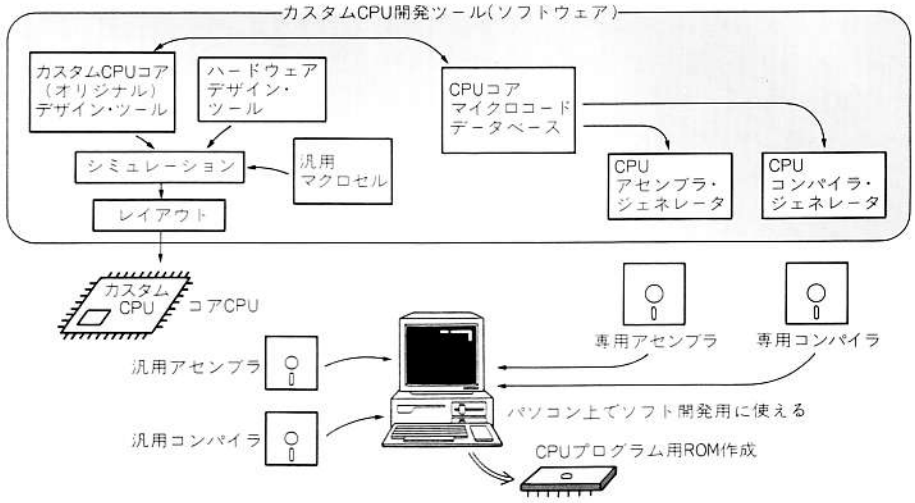
ろんチップ上のコアのサイズは十分に小さくなります．そして，CPUコアのためのテスト・パターンや，シミュレーションのための機能モデルも，あわせて自動生成されるでしょう．仕様を指定すると，周辺LSIに相当する回路も，すべてオンチップに搭載されます．また，カスタムCPUコアのマイクロコードから，標準の開発言語システムに合わせた，クロス・アセンブラかクロス・コンパイラも作られる，というのが望ましいでしょう(**図5.28**)．DSPブロックについても，演算式の記述を入力として，タイミング分割・マイクロプログラムの自動生成までを実行します．プロセスも1ミクロン・ルール程度でとどまらずに，スケールもさらに1桁ほど小さくして，オリジナルCPU設計の冗長度を吸収するようなゲート単価にします．希望はいくらでも書けます．メーカの皆さん頑張ってください．

さて，このような環境が出現しても，マイコン・システムというものの考え方は，基本的には変わりません．最初のステップは，やはり目標となる仕事を考えて，ソフトとハードの領域に分割することです．これはスピードと事象(イベント)の時間密度の要因によって，

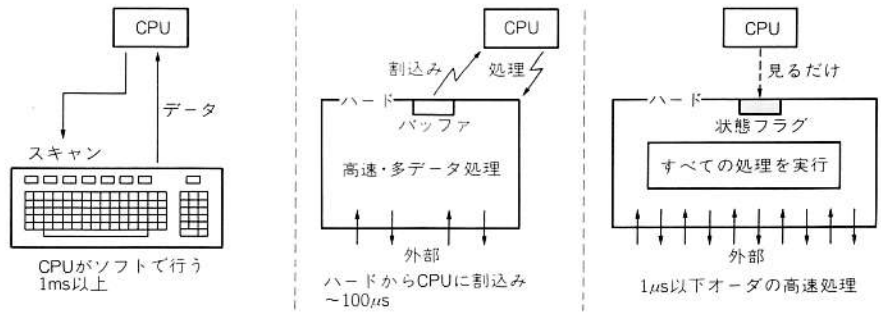
① CPUがソフトで行う
② ハードが行ってCPUに割込みをかける
③ 専用のハードが行う

の3種類になります(**図5.29**)．ここを慎重に見定めて，つぎのシステム設計へと移行するわけです．CPUとハードウェア部分とのインターフェースの方法については，CPUのメイン・ルーチンからステータスをモニタするもの，一定のタイマ・イベントごとにポーリングするもの，ハードのイベントから割込み要求するもの，ハードのタイミングで強制的にCPUを停止させて転送するもの，等々の方法があります(**図5.30**)．この，時間的な意味でのシステムの最適化というのも，マイ

〔図5.28〕
カスタムCPUコアの
ソフト・ジェネレータ



〔図5.29〕
スピードによる
CPUとハードの分業



コン・システムの重要な設計指針といえるでしょう．

さて，マイコン・システムの開発手法についても，まとめの意味で，すこし前向きの想像を書いておきましょう．**今後は，開発の対象は，すべてEWSのソフトの中に存在する，**ということになります．ハード的な機能を指定すると，CAD上で回路が自動設計され，このハードウェアに対する検証は自動化されます．ソフトウェアについても，CPU機能の条件を指定して，ソフト的な仕様を日常言語によって記述すると，曖昧な部分をシステムが対話的に確認して，標準の機能仕様記述言語に変換され，これにもとづいて，CPUのプログラムが自動生成されます．そして，開発ツールのシミュレータにおいて，「完成したハードウェア・モデル上で，完成したソフトウェア・モデルを走らせた場合の動作」というものが検証できるのです．これでOKとなると，必要なカスタムLSIが自動設計され，サンプルが試作されます．一方で，回路基板が自動設計・試作され，プログラムROM，カスタムLSIサンプル，全部品とともに製造の自動組立機に投入され，プロトタイプの基板が完成します．これを，あらかじめ3次元CADによってデザインされ，自動試作されていたケースに組み込むと，ここに完成品と同等の試作品ができあがります(**図5.31**)．これはきわめて美しいシステム開発の姿ですが，夢物語でなく，このような体系のかなりの部分が，すでに現実になりつつあるところなのです．

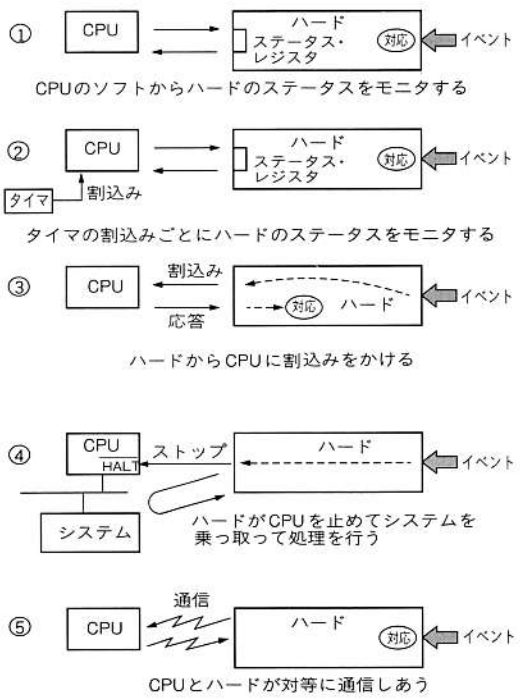
おわりに

人間の創造したマイコン技術ですが，このところ，逆に人間を振り回しているように感じるのは筆者だけでしょうか．技術の特性として，

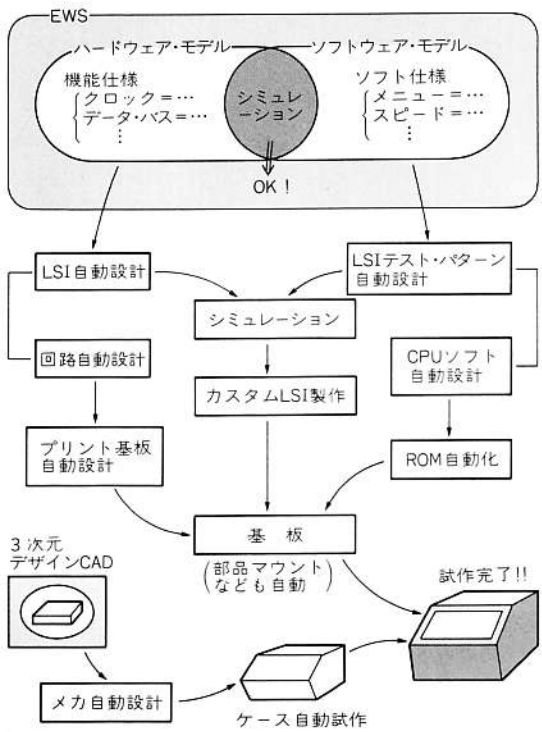
「進歩こそ美徳で，成熟は怠慢だ」

が宿命であるのは事実ですが，当の人間はといえば，マイコン技術どころか，自然科学という概念すら，本質的には消化していないように思うのです．自分の日常を振り返ってみても，CPUソフト開発やLSI開発，回路設計やシステム実験のなかで，〈まったく統一性のない種々の装置〉と，〈まったく統一性のない数十のソフトウェア・ツール〉とを，そのつど，頭を切り替えな

〔図5.30〕CPUとハードウェアとのインターフェース方法



〔図5.31〕これからのマイコン・システム開発



がら同時に使っています．そして，そんな混乱した状況から身についた，唯一の自衛策といえば，

① 特定の言語を覚えない
② 特定のシステムに慣れない
③ 情報は必要なときにマニュアルから得る

という姿勢です．いいかえれば，つねに自分の脳の活性領域をRAMにしておく，ということでしょう．マイコン技術の一面なのでしょうが，ある分野のエキスパートになった瞬間，この世界では「過去の人」となってしまう気がします．

さて，いろいろ文句もいいましたが，筆者はこの世界が気に入っています．はたして，どこまでこの進化についていけるか，自分をためす場としては十分な相手です．そして一方で，心のどこかでは，ある種のゲームとして参加しているような，距離を置いて楽しむ余地を残しておきたい，とも思っているところです．

コラム 5.3

海外メーカか国内メーカか

システム設計の中で，そこにどのメーカの半導体を使うか，という選択の場面はしばしば登場します．ここではその基準のうち，海外メーカか国内メーカか，という視点について考えてみましょう．

すこし昔であれば，アメリカの半導体の信頼性は国産メーカよりも3桁悪いとか，海外メーカは民生用ICの場合，不良品があっても取り替えればいいという傲慢な姿勢だからいやだとか，などの感情的ともいえる基準がありました．しかしさすがにこれは改善されているようで，最近は筆者もあまり気にしません．むしろ，いくら日本国内に生産拠点を置いていても，データ・シートやマニュアルが全部英語であるようなメーカに，多少とも抵抗を感じるものです．もちろんマニュアル程度の英語を読むのは技術者の義務ですが，同一機能のLSIを選ぶとして，一方が日本語で一方が英語のマニュアルであれば，とりあえず国内メーカに傾くのが人情というものでしょう．そこで筆者の場合，英語のデータ・シートしかないのにあえて海外メーカを選択する場合，そこにはしっかりした理由がある，ということになります．

その一つは，海外メーカにしかないような，ユニークなICを求める場合です．これまでの国内メーカは発想が貧困で，よそで売れたものを大量生産するのが常であったので，ダラス・セミコンダクターとかマキシムといった，ユニークな発想のICメーカの半導体は，そのアイデアが世に出てすぐに採用しようとすれば，オリジナル・メーカしかありません．やがて何分の1かのコストの同種ICが国内メーカから出るまでは，ここは海外メーカの独壇場なのです．

また，最近のアメリカの政治的圧力にもかかわらず，コスト的な理由から，アジアNIESの半導体にシフトしている部分もずいぶんあります．アメリカが文句をいうと，通産省がメーカに圧力をかけて海外半導体を買えといい，メーカはアメリカよりも安いNIESの製品を買い，日本の輸入比率はちゃんと増加し，アメリカは停滞してさらに文句をいう，という漫画のような図式となっているのです．

ASICの分野の場合，実際に相当数のメーカを対象に検討してきた経験からいえば，やはり国内メーカを第一候補にしてしまうようです．これはもう，サポートの違いでしょう．国内メーカであれば，あらゆる開発環境のマニュアルが日本語で，サポートする担当者・エンジニアも国内におり，必要に応じてすぐに移動して直接相談できます．シミュレーションは衛星回線でアメリカのホストと直結されています，というのは聞こえはいいのですが，ちと遠い印象は否めません．「うちもスタートするよ」といってから1年以上も稼働していない某海外メーカは例外的としても，国内メーカはいったことはやってくれる，という信頼度が一枚上手なのは事実です．

海外メーカの窓口というのは，日本法人の営業担当者であったり，商社や代理店の営業マンである場合がほとんどです．ところが，業界の体質がアメリカ的であるためか，しばしば担当者が転職してしまいます．これは正確にいえば，同業他社への「転社」，すなわち引き抜きです．たしかに，この人はなかなかやるな，というようなキレる人が，気がつくと会社を移っているという例がありますが，少なくとも海外メーカの窓口の腰の軽さは，ASICのような立ち入ったビジネスには多少のマイナス要因といえるかもしれません．

ASIC：どこで・誰が・いつ設計するか

半導体メーカ以外で，自社でLSI設計の設備をもつ企業は，設計までを自社開発して，製造をLSIメーカに任せることになります．ところが，程度と規模によりますが，LSI設計に必要な環境をすべて自社で揃えるのは結構大変で，月に何個かはASICを作る，とかのペースでないとうまく回転しません．そこで，自社ではLSI設計の設備をもたないという場合，製造はもちろんLSIメーカとして，設計の段階をどうするかという問題が発生します．ここでは，そのようなケースについて，LSI設計のTPOを考えてみます．

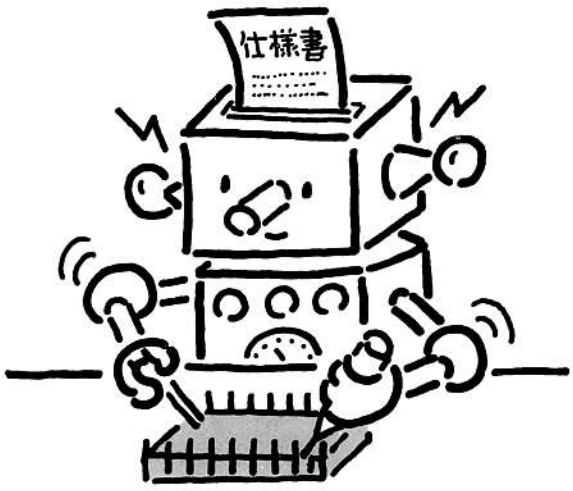
一番簡単な方法は，LSIメーカに設計をすべてやってもらう，という手が考えられます．ユーザは詳細な機能仕様書だけを提出して，内部についてはメーカに任せてしまう，というもので，できあがったLSIのトラブルはメーカの責任，仕様の不備はユーザの責任とはっきりしています．ただし，メーカの設計担当者はユーザの業界固有の回路技術・特許状況・ノウハウなどに関しては素人なので，チップ効率やコストの面ではあまり美しいLSIとはなりにくく，さらに相当の開発費が必要となり，開発期間も長くなります．CPUのソフトやマスクROMを含む，システム全体と抱き合わせでのプロジェクトなどで採用される方法ですが，担当する技術者としてはあまり面白味がありません．

つぎに，メーカや代理店のデザイン部隊に回路とテスト・パターンを渡す，という昔からよくあるインターフェースが考えられます．これはブレッド・ボードのTTL回路図や，フリーハンドで描いたテスト・パターンを，LSI開発システムの固有の言語に変換する作業を代行してもらう分，ユーザが余計な仕事をしなくてすむもので，小規模のシステムならばメリットが先行します．これもメーカとユーザの責任分担の境界がしっかりしているので，ASICの歴史のかなりの期間，主流となってきました．

さらに，システムが複雑になり，コスト制約・回路ノウハウなどのテクニックが要求されるようなレベルとなると，ユーザの開発担当者が直接，LSI設計ツールを使って開発するような段階も登場してきます．ここではEWS，ミニコンなどの開発ツールやCAD，シミュレータといったソフト・ツールをすべて理解し，手足のように駆使してシステム開発を行う必要があるので，ユーザの技術者の負担は相当に増大します．しかしその分，回路の隅々まで設計思想の行き届いた，効率の良好なLSIが開発できます．さらに，ここには2種類の形態があり，メーカが用意した「デザイン・ルーム」に出向いて設計するものと，メーカ貸出しやレンタルによって設計ツールをユーザの手元に置いて設計するものとがあります．

ユーザの手元に開発環境がある場合のメリットは，システムのブレッド・ボードなどの周辺環境と一緒なので，全体システムの開発と並行した場合に非常に有効であることです．たとえば筆者の経験ですが，LSIに相当する回路をブレッド・ボードで組んでICEのプローブを挿入し，CPUのファームウェア開発とLSIのハードウェア開発を同時に進行させたことがあります．机の上には，実験用のブレッド・ボード，CPU開発用のパソコン，LSI設計用のパソコンが並びました．ソフトの問題が判明すると，ソースをCPU開発パソコンによって変更し，あとはバッチでアセンブル，リンク，デバッガ，ダウンロードまで自動的に実行します．またハードの問題点があると，まずブレッド・ボードの回路をハンダごてで修正するとともに，LSI開発パソコンで回路データを修正し，対応するテスト・パターンを変更すると，あとはバッチでシミュレーションまで走らせておく，という作業を繰り返しました．言葉では伝えにくいのですが，驚くほど快適にシステム開発ができる雰囲気がわかってもらえるでしょうか．

一方，わざわざメーカのデザイン・ルームをリザーブして出向く，という方法にもメリットがあります．メーカはパソコン・レベルのツールなら貸し出してもくれますが，あの大きなSun-4をたびたび借りる，というわけにもいかないので，回路規模が相当のLSIであれば，ミニコンやEWSのあるデザイン・センタのほうが便利なのです．たとえば2,000ゲートのシミュレーションをパソコンでやるのとミニコンでやるのとでは，それほどス



ピードの差が気になりませんが，これが2万ゲートとなると明瞭になるのです．何千ステップというテスト・パターンをいくつも走らせる，という状況では，全体の開発効率の差は非常に大きくなります．

そして案外知られていませんが，**デザイン・ルームという隔離された静穏な場所に籠って集中できる，というメリット**も重要です．通常の環境によって評価は分かれるのでしょうが，少なくとも筆者の場合，電話・会議・来客・騒音・煙草などの悪環境と闘いながらの設計と，デザイン・ルームでの短期集中設計の両方を知っているので，たとえば1万ゲート程度以上のLSIを開発するとすれば，絶対にデザイン・ルームの使用を条件として提案すると思います．

これらの方法以外にも，ユーザの手元には回路設計，テスト・パターン設計用のパソコンを置き，電話回線でメーカの大型機と接続してシミュレーションを行う，という方式が流行したことがあります．これは一見スマートな感じなのですが，電話回線というボトルネックがあり，接続されるのを何十分も待たされたり，インタラクティブなシミュレーションでないもどかしさがあって，どうも思うように進まず，遅くても手元のパソコンのほうがまし，となってしまったようです．少なくとも現在では，手元の装置にかなりのインテリジェント機能をシフトさせる方向で発展しています．

そのうち，PCエンジンに「LSI設計」というゲームが発売されると，子供たちはCADゲームで設計し，シミュレーションというゲームをクリアすると，その結果からレイアウト・データを作成して電話で転送し，オリジナルの電子オモチャを作れる，などというのはどうでしょうか．

Appendix 5.2

ゲートアレイの設計ポイント

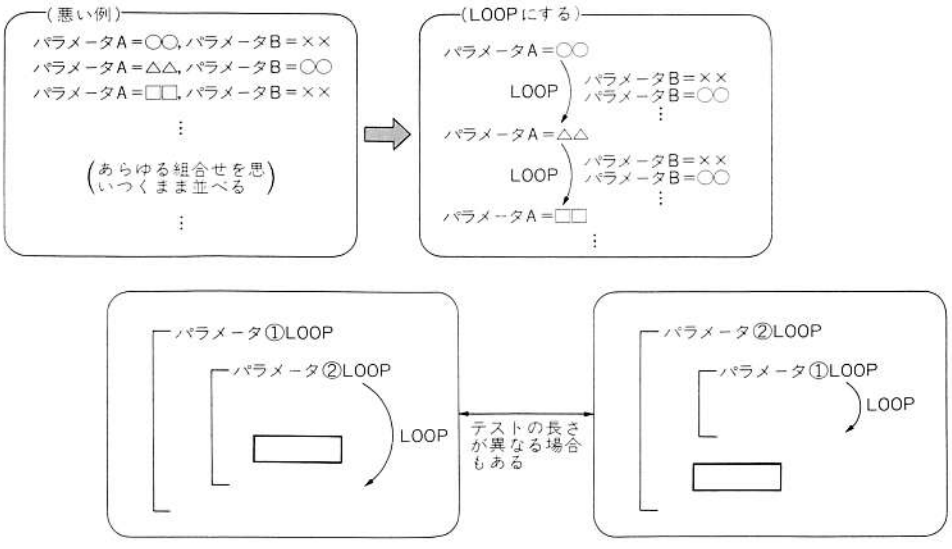
1 テスト・パターン圧縮のテクニック

(1) 最終的に展開されたテスト・パターンが同じステップ数になるにしても，ソースのテスト・プログラムで楽をする(手抜き)ようにします．これにはテスト・パターンのエディタのマクロ機能やループ機能が活用されます．ソースが簡潔であれば，修正や拡張も容易になります．

(2) 2種類以上のパラメータがあるときには，まず一方を固定してもう一方を変える，というループ構造にします(図A)．やみくもに多くのパラメータの組合せを並べるのは適切ではありません．また，2重ループのときは，どちらのパラメータをループの外側にするかでテストの長さが変わる場合もあるので，十分に検討します．

(3) 信号のグループ分けとダブル・パルス定義などを活用して，冗長なステップを圧縮できないかを考えます．図Bの最初の例では各ステップでの信号変化が1箇所ずつで計6ステップですが，信号Ⓑのダブル・パルス定義によって3ステップに，さらにもし信号のタイミング・マージンに余裕があれば2ステップで記述できることに

〔図A〕
2種類以上のパラメータがあるときは一方を固定し，もう一方を変える



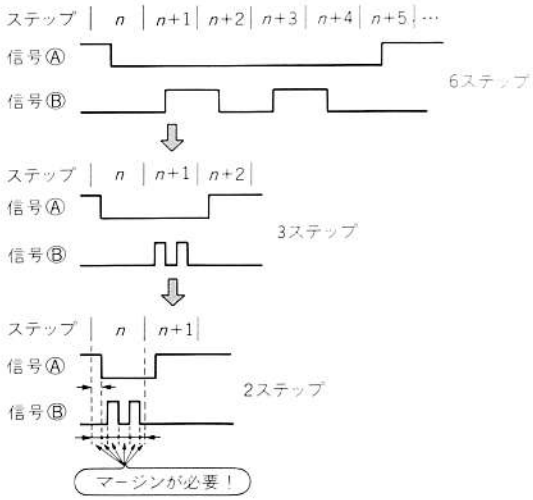
なります．

2 テストのポイントをおさえるテクニック

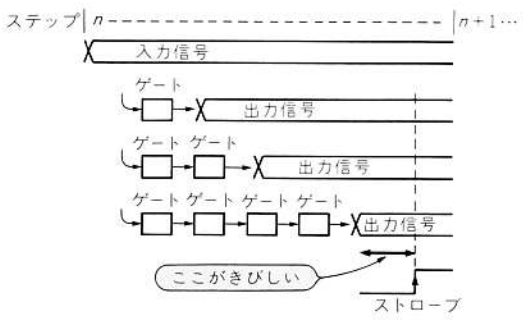
(1) シミュレーションにおいては，もっとも長いパスでエラーがなければ，それより短いパスでは手抜きができます．図Cの例では，同じ入力に対してもっとも多くのゲートを経由して遅延の積み重なったパスの出力信号とテストのストローブ位置の部分がもっともきびしく，注意が必要になるわけです．チェック・ポイントとしては，多段のバッファ，セレクタ，ゲートを通るパスの周辺のテストを強化するようにします．また，図Dのように加算器を多段に重ねると遅延が加算され，とくにキャリ・ビットの扱いが要注意となります．タイミングをかせぐときはルックアヘッド・キャリ(HC182などに相当)のロジックを追加します．

(2) 二つ以上の別々の経路からの信号が合流するところが要注意です．図Eの場合，同じ入力信号から複数の経

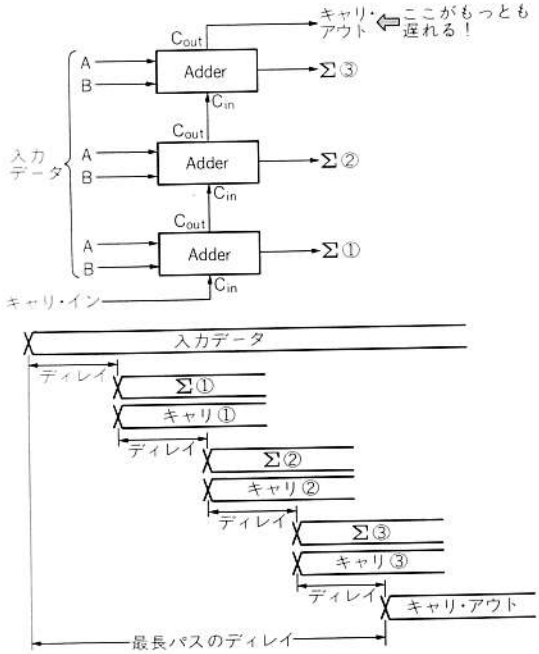
〔図B〕冗長なステップを圧縮する



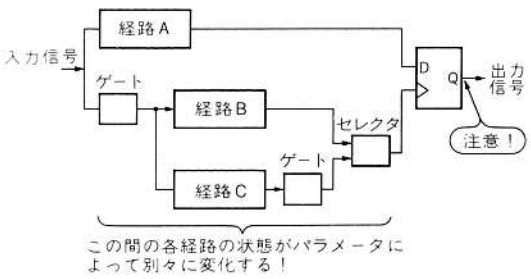
〔図C〕もっとも多くのゲートを経由するパスに注意が必要



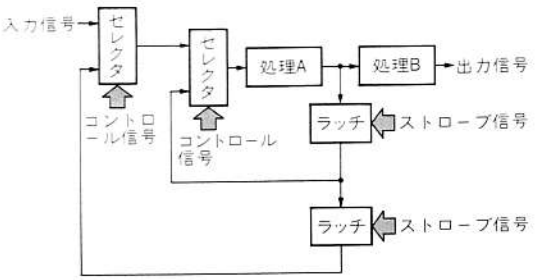
〔図D〕加算器を多段に重ねると遅延が加算されるので要注意



〔図E〕二つ以上の信号が合流するところが要注意



〔図F〕信号がループのような形になる部分のタイミングも要注意

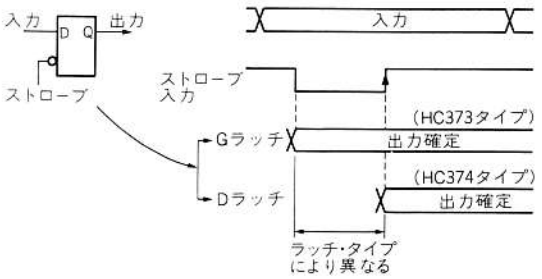


路を経た信号が合流するので，途中の経路の状態がパラメータによって変化する可能性があります．これは悪い例ですが，結果的にこのような形になる場合には，テスト設計の際には十分に注意する必要があります．

(3) 信号がループのような形になる部分のタイミングも要注意です．図Fは信号が二つのループを構成している例で，うまいテスト・プログラムを組むのも意外にめんどうなのですが，それと同時に，二つのセレクタのコントロール信号と，二つのラッチのストローブ信号を発生する回路のタイミングにも十分注意してやる必要があります．

(4) ラッチ・タイプにも注意します．図Gのように，同じ入力信号と出力信号でも，ラッチ・タイプがGラッチとDラッチとでは出力が確定する場所が異なります．Gラッチの場合にはストローブがアクティブである区間に入力データが確定しつづけていないと，シミュレーション・エラーとなりますが，Dラッチではストローブの立上り部分で入力データが確定していればいいのです．また，ラッチを多用するシステムでは，

〔図G〕ラッチのタイプがちがうと出力が確定する場所が異なるので注意



Gラッチ＜Dラッチ

というゲート数の関係も重視して，ゲート効率をかせぐために，なるべくGラッチを使用するようなタイミング設計を心がけます．

コラム

記憶媒体の栄枯盛衰

パソコンやマイコン機器の外部記憶媒体，補助記憶媒体というのは，半導体の進歩に負けない，激動の歴史をもっています．最初はまず，カセット・テープやカートリッジ・テープが主流でした．CPUも遅く，メイン・メモリも小さいので，テープ記憶の手間は低コストによって許容されていました．現在のパッチ・プログラムに相当するものを走らせると，カートリッジ・テープがまず先頭まで自動的に巻き戻され，ヘッダを読み，つぎつぎとテープを読み飛ばし，目的のヘッダに到達するとプログラムをロードし，そこから自動的にスタートしました．

その後，一時バブル・カセット・メモリなるものも流行しました．フロッピと違って磁気の影響を受けない，テープよりかなり高速，と宣伝されたのですが，とにかく高価なためにほとんど消えてしまいました．

フロッピ・ディスクも，8インチ・5インチ・3.5インチと，あるいは片面・両面・単密度・倍密度と進化していますが，これらの互換性の混乱に泣かされた人はかぎりないことでしょう．また，途中で5インチの次世代のときには，電気メーカ各社が大同団結した3インチの規格が，少数派ソニーの3.5インチ規格に撃退される，というドラマもあり，観客としてはなかなか楽しめました．(ビデオの場合，ベータ対VHSの勝負を決定したのは一般大衆でしたが，フロッピの「大どんでん返し」を演出したのは，IBMの採用でした．)

さらに2インチや超高密度もの，そしてクイック・ディスクもあり，フロッピ・ディスクという媒体は，現在でも低コスト記憶媒体の主流となっています．ファミコンのディスクのコピー・ツールを圧殺するために，任天堂がクイック・ディスクを買い占めて市場に生ディスクを流通させないようにした，などという話は，真偽のほどは別としても面白いものです．

このままCDのような光ディスクにつながるのかな，と思っていると，ここにきてなんとICカードが浮上してきました．半導体メモリの特徴は，ディスクと段違いの高速・高信頼性であり，もちろん従来から，メモリ・カートリッジの形で高級な機器に使用されてきました．それが，メモリ・カードという共通規格の量産効果によって，予想以上に民生機器にも登場してきています．これは，大容量メモリの進化による成果であり，今後はフロッピ1枚分ぐらいの容量のICカードが流行するかもしれません．(ただし，現在よりも1桁ぐらいは安くしてほしいものですが．)

そして，さらにつぎの世代の記憶媒体も，どこかでは着々と開発されているはずです．これは筆者のたんなるカンなのですが，今度は電気的な仕掛けでなく，高分子か何かの，化学的または物理的な原理によって，より低コスト・高密度な記憶が実現されるような気がします．

アルファベット略語解説

A-D 変換[えいでぃー](Analog-to-Digital)
アナログ信号をパラレルまたはシリアルのディジタル信号に変換すること．その素子としては，8ビットから2ビット刻みで16ビットあたりまでのものがポピュラである．実際のA-D回路では，サンプリング定理を満たすようなLPF(ローパス・フィルタ)とS&H(サンプル・アンド・ホールド)によって，正確なタイミング設計をしてやらないと，十分な精度が出ない．最近の猛烈な価格低下によって，応用範囲が拡大している．

ADC[えいでぃーしー](Analog-to-Digital Converter)
A-D変換を行う素子．

ASIC[えいしっく](Application Specific IC)
ユーザが希望の機能を盛り込んで設計するタイプのLSIの総称．従来の回路技術がメーカの標準LSIを購入するだけのものだったのに対して，カスタム機能のLSIをセミカスタムの低開発費・短期間で実現するもの．「ASIC分野で活躍しているメーカの体質は若々しくて明るい」が，「かけ声ばかりで実質的な動きの遅いメーカの体質はオジンくさい」という一般ユーザの感覚はかなり妥当なものである．

BASIC[べーしっく](Beginner's All-purpose Symbolic Instruction Code)
対話的に実行できる，入門的なパソコン言語．解説書で勉強するよりも，実際に遊んでみれば理解できてしまう．

BIOS[ばいおす](Basic I/O System)
パソコンのハードウェアを操作するサービス・ルーチンを，システムのプログラムとして，通常はROMに格納したもの．個々のソフトウェアで直接にハードウェアをアクセスするのは非常に困難なので，所定のパラメータをセットしてこのBIOSを呼ぶと，必要な処理を実行してくれる．DOSもここを呼ぶ．また，パソコンの電源ON時やリセット時の初期設定も，このBIOSからスタートする．BIOSの内容を公開するような，良識あるメーカが少ないので，必要に応じて解析しなければならない．

bps[びーぴーえす](Bits Per Second)
シリアル通信の転送速度を表す単位．たとえば9600 bpsといえば，毎秒9600ビットということになり，スタート1ビット＋データ8ビット＋ストップ1ビットという規約の場合，1バイトを10ビットで表現しているので，最大で毎秒960バイトの転送が可能となる．

C[しー]
最近はやりのコンピュータ言語．種類が増えるとともに，方言も多様に分化している．

CAD[きゃど](Computer Aided Design)
コンピュータ支援による設計およびそれ用のツール一般．

CMOS[しーもす](Complementary Metal-Oxide Semiconductor)
半導体のプロセスで，それまでのバイポーラに代わって主流となっているもの．消費電流・発熱が少なく，スレッショルド・レベルのマージンが広い．スピード面の短所も相当に改善されていて，ASICのプロセスでは，ほとんど全部がCMOSである．

CPU[しーぴーゆー](Central Processing Unit)
マイコンの心臓部であるLSI．プログラムを読み，解釈し，実行する，というノイマン型のものがほとんどである．

CRT[しーあーるてぃー](Cathode Ray Tube)
ブラウン管タイプのディスプレイ．技術者の眼にとって敵である．

D-A 変換[でぃーえー](Digital-to-Analog)
パラレルまたはシリアル形式のディジタル信号をアナログ信号に変換すること．その素子は，3年前ぐらいまでは16ビットのD-Aといえば1万円，というイメージであったものが，コンパクト・ディスクという民生の巨大な需要によって新製品が続々と参入して価格が崩壊し，いまや数100円は常識で，自動車メーカには100円台で供給されている，といわれている．もちろんコーヒー1杯分の16ビットD-Aは，実質的には14ビット程度の精度でしかなく，同じ音声信号をA社・B社・C社のD-Aに与えると，聞こえる音がまったく異なる，などというのも常識となっている．

DAC[だっく](Digital-to-Analog Converter)
D-A変換を行う素子．

DIP[でぃっぷ](Dual In-line Package)

接続端子が2列に並んだ，もっとも標準的なICパッケージのタイプ．あのゲジゲジのような外観のイメージから，IC屋が女の子にもてないのだ，という通説がある．最近のプリント基板は，あの0.1インチのピン間に，配線パターンを5本とか6本通す，というレベルの競争で，だからこの世界は女の子にもてないのだ，とやっぱり納得してしまったりする．

DOS[どす](Disk Operating System)

パソコンの動作環境で，ディスクという媒体をベースに構築された，非常に高機能のシステムのこと．パソコンの一般化というのも，DOSがなければここまでは浸透しなかったと思われる．インターフェースの一般化のためのDOSが，最近では某N＊C社のわがままによって，互換性の混乱という困った事態になってきている．

コラム

技術者の情報源

日進月歩する技術状況のなかで，あらゆる情報を取り入れ，整理・把握して業務に生かしていく，というのは技術者の宿命であり責任です．ここで問題となるのは，どのように効率よく情報源をおさえるか，というポイントです．評論家なら，多少古いような雑誌類でも系統的におさえればいいのでしょうが，他社との競争下でスピードを要求される現場では，もはや活字となって印刷されたような情報は「過去」なのです．あまり美しくないサンプルですが，ここで筆者の情報源を例にとって，この問題について考えてみます．

まずは月刊の雑誌類があります．おそらくどのような職場でも，関連する分野の雑誌を10誌や20誌は購読しているでしょうから，これはもっとも身近な情報源です．記事ばかりでなく，広告までをしっかり読むと，行間に情報があふれています．たとえば，トラ技を初めて読んだ人にとって，あの広告全部のボリュームは閉口してしまうのですが，慣れて毎月読んでいる人にとっては，むしろ毎月ほとんど変化しない広告のごく一部が変わっていく，その傾向こそが重要な情報なのです．それぞれの雑誌を斜め読みする所要時間も，慣れると非常に少なくてすみます．また，直接電気屋に関係しない，化学・科学・機械・経済といった分野の雑誌についても，ときどきチェックを入れるのは意味のあることです．また，つぎつぎと新しい雑誌が創刊されているのに会社が購読しない，というものについては，書店で軽く立ち読みして，必要なものは特別に購入したりします．

日経＊＊＊という雨後の筍のような一群の雑誌は，相互に重複した情報を割り引いて読めば，また，各種の専門新聞も，個々の記事の技術的な未消化を前提に読めば，それぞれ有効な情報です．あるいは，各種の学会誌や，公的機関の報告書も，あまり有益なものは多くないにしても，情報源の一種といえます．また，特許公報も，状況によっては貴重な情報源として役立ちます．

最近は，登録された技術者に直接発送するタイプの，無料情報誌というか，ダイレクト・メールの雑誌も多く創刊されています．これは資料請求ハガキまで含めてすべてタダですから，利用しない手はありません．各誌の情報はどうしてもかなり重複しているのですが，バランスをとってうまく活用できるものです．

半導体メーカの例でいえば，メーカの営業担当者，あるいは代理店の担当者という存在もまた，重要な情報源ルートです．データ・ブックなどの，一般向けに発表された製品についての資料を請求するばかりでなく，未発表の開発品種の情報も多くあります．場合によっては，LSIがメーカから一般に発表されるときに，すでにそのLSIを使った応用製品をユーザが量産しているような開発形態もありうるのです．とくにASIC関係では，一般には最後まで絶対に公開されない，という性格の資料が多くあるので，この部分ではこのルートが唯一に近いものです．余談ですが，それぞれの担当者のレベル(技術・性格・人間的)というのはじつに千差万別なのですが，個人差を越えた代理店・メーカの体質の差，のようなものも見えてきて，ある意味での人生勉強にもなります．

これらの日常的に過ぎ去っていく情報をひたすら食べていけば，すぐに胃袋はパンクしてしまいます．そこで重要なのは，ここからの対応です．書籍のコピーというのは問題があるので，ここではデータ・シートなどをファイルする，という例について考えてみましょう．標準ロジックの新バージョンが出た，という何枚ものデータ・シートが送付されてくると，筆者は一瞥して倉庫に入れてしまいます．手元には標準ロジックのデータ・ブックがすでに1冊あるので，「少し高速のものが出た」という情報をメモしておけば，あとは不要なのです．また，新年度のデータ・ブックが3冊来たとすると，1冊は倉庫に入れ，もう1冊の中から関心のあるものの部分だけを切り取ってファイルし，あとの残りは捨ててしまいます．こうやって，手元には最低限のデータ・ブックと，自分なりの視点でセレクトされたファイルと，思いついたことを記録したノート，という3点セットをもつようにしているのです．これらの資料を定期的に見直して整理してみると，技術動向の確認や，新しい特許を書くような場合に，非常に役立つようです．

DRAM[でぃーらむ](Dynamic RAM)
電源が供給され，規定条件でリフレッシュされている場合にのみ動作する，というタイプのRAM．そのかわり，メモリ・セルをコンパクトに構成しているため，コストあたりのビット密度にメリットをもっている．パソコンではほとんどがDRAMなので，メモリ中のRAMディスクにファイルを置いたままのときに停電したりすると，悲惨なことになる．

DSP[でぃーえすぴー](Digital Signal Processor)
本文では，単体のチップとしてのディジタル・シグナル・プロセッサという意味と，ディジタル・シグナル・プロセッシング(ディジタル信号処理)回路という意味の2種類として使っている．

EPROM[いーぴーろむ](Erasable PROM)
書込み器でデータを書き込めるプログラマブルROMのうち，データを消去して再使用できるもの．パッケージに丸い窓があいていて紫外線で消去するUVEPROMが一般的で，電気的に消去するタイプのEEPROMも増えてきている．最近のマスクROMは非常に大容量化していて，試作段階ではEPROMで実験するにしても，マスクROMと同じ容量にするには4個とか8個を並べる必要がある．ところがあるメーカでは，パッケージ上面が全部窓となっていて，そこに4チップをびっしり並べた，異様な外観のEPROMを供給している．密度が高いので重宝であるが，今度はサポートするROMライタがないので，アダプタの自作が必要となる．

EWS[いーだぶりゅーえす](Engineering Work-Station)
当初はミニコンとパソコンの間のような位置にあったのに，最近ではパソコンに毛の生えた値段で，少し前のミニコンを超える性能のものが多い．これからは技術者一人に1台の時代といわれているが，実際にそんな恵まれた環境にいる人は1％にも満たない(……と思いたい)．どこかに不要パソコンを下取りしてEWSに替えてくれるようなところはないかなあ．

FCC[えふしーしー](Federal Communications Commission)
アメリカの連邦通信委員会による，電子機器の不要輻射に関する規則．いくつかの「クラス」があり，それぞれ該当するものをクリアしていないと輸出できない．安易な設計のために，FCC対策で開発期間が倍増するような悲惨な実例もあるという．

FDC[えふでぃーしー](Floppy Disk Controller)
FDDをコントロールするLSI．FDCのタイプによっては，特定のコピー・プロテクトが外せない，といった差がでる場合もある．

FDD[えふでぃーでぃー](Floppy Disk Drive)
フロッピ・ディスクをドライブするメカ部分と，FDCと接続されるまでの電子回路とが一体化されたモジュールのこと．システム屋からすると，FDCまで搭載して，たんにCPUのバスに接続される単純I/Oのような，インテリジェントなFDDもあっていいと思うのに，なかなか登場してこない．

FFT[えふえふてぃー](Fast Fourier Transform)
ディジタル信号処理の一種の，高速フーリエ変換のこと．パソコンのソフトとして実現すると案外に低速なので，専用のDSPのハードによって行う場合も多い．技術の進歩によってますます可能性が広がってきた，という意見と，もはやFFTの限界が見えてきて大したことは期待できない，という意見とが対立している．

FORTRAN[ふぉーとらん](FORmula TRANslation)
コンパイル型の高級言語の一種(だいぶ昔のことなので，もはやこれ以上コメントすべき印象が残っていない)．

GPIB[じーぴーあいびー](General Purpose Interface Bus)
パラレル・バスによる，計測器・パソコンなどのための通信規格の一種．

HEX[へっくす](HEXadecimal notation)
16進．16進で7Fといえば10進の127のことである．

IC[あいしー](Integrated Circuit)
かつての「産業のコメ」も，もはや水か空気のような印象である．筆者はマイコン少年として，球→TR→ICの推移を体験したけれど，今の子供はフラット・パッケージのCMOS世代なので，初期のスタンダードTTLが走ると熱くなるのを見たら，ちょうど筆者が真空管に感じたような古風な印象をもつのだろうか．

ICE[あいす](In-Circuit Emulator)
CPUの動作を，実際の基板回路に挿入して代行するタイプのデバッグ・ツールのこと．あるメーカのシステムでは，ホストのパソコンと通信する本体(親)があり，そこからケーブルでコントロール・モジュール(子)に行き，さらにケーブルでエミュレータ(孫)に行き，さらにケーブルでボックス(曾孫)に行き，その先にCPUプローブのケーブルがつながっている．このシステムを立ち上げるまでには，コーヒーを飲める時間がある．しかもこのシステム，昔話でなく現役なのである．こ

んなメーカに律儀に付き合うユーザが，いつまで残るのか興味あるところである．

I/O[あいおー](Input/Output)
入力・出力の総称．

LAN[らん](Local Area Network)
各種通信回線によって，近距離にある複数のコンピュータを結びつけたネットワークのこと．

LCA[えるしーえー](Logic Cell Array)
ユーザによってプログラム可能なゲートアレイ．ソフト的に内部情報を転送することで，1,000ゲートから2,000ゲート程度の回路を実現する．実際には配線効率の問題で，この何分の1，といった感覚が正しいようである．(商標)

LCD[えるしーでぃー](Liquid Crystal Display)
液晶ディスプレイ．かつては寿命が5年とか10年なので問題があったのに，最近はあまり聞かれない．寿命が20～30年に延びたのか，それとも最近の電子機器は5年もしないうちに捨てられるから，問題にしなくてよくなったのか．

LED[えるいーでぃー](Light Emission Diode)
発光ダイオードのディスプレイ．LCDではいくらバックライトをしても，LED表示の力強さがない，という分野ではますます健在である．実際，電流を食うとはいっても，電源ON表示のLEDが一つ光っているだけで安心するし，もしLEDが点滅でもすればドキッとして十分に注意を喚起する，と思う筆者は古いのだろうか．

LSI[えるえすあい](Large Scale Integrated IC)
VLSIだULSIだ，と展開されている，現代の「産業のコメ」．

MPU[えむぴーゆー](Micro Processing Unit)
CPUと同じ．モトローラはCPUをMPUと呼んでいたわけだが，この用語に関してもモトローラの負けである．

MS-DOS[えむえすどす](Microsoft-DOS)
マイクロソフト社のDOSから展開された，現在もっともポピュラなパソコン用のDOS．日本のMS-DOSは，バージョン3あたりからおかしくなってきたようで，これ以上互換性の混乱が拡大すれば，名前を分化させたほうがよくなるのではないだろうか．(商標)

OS[おーえす](Operating System)
コンピュータを階層的に制御する場合の，ハードウェアとアプリケーション・ソフトウェアの間に位置するシステム．DOSというのもOSの一種である．すっかりマイナーとなってしまったOS-9というOSは，DOSのようにディスクだけを特別扱いしないで各種の資源を平等に扱う，非常に高機能で美しいOSであるが，MS-DOS軍団に圧倒されてしまった．技術者が美しいと思うものが必ずしも主流とならない，という図式はここにも存在する．ついでに8ビットCPUを公式化すると，

リトル・メジャー＝Z80
ビッグ・マイナー＝6809

となるだろうか．

PAL[ぱる](Programmable Array Logic)
ユーザが書込み器でプログラムできる，小規模なロジックICのこと．最近は，公称1000ゲート程度の「大規模PAL」もある．PALに使い慣れると，何も考えずにとりあえずPALで埋めた回路を設計して，あとでPALをプログラムするときに，必死でつじつまを合わせるように考える，というズボラな発想に流れがちであり，要注意である．(商標)

PIO[ぴーあいおー](Parallel I/O)
CPUからのパラレル信号を外部とインターフェースするための周辺LSI．ポピュラーな8255のCMOS版の71055の場合，8ビットのラッチや入力バッファの3個分よりも安くなっている．ここへ参入した別メーカでは，8255が二つ分入るという仕様で対抗し，さらに別メーカは，8255が二つ分＋おまけ仕様で追随している．この涙ぐましい姿勢が，日本の半導体メーカの原点である．

PLD[ぴーえるでぃー](Programmable Logic Device)
PALと同じ．

RAM[らむ](Random Access Memory)
データを読み書きできるメモリ．

ROM[ろむ](Read Only Memory)
データが読出し専用のメモリ．

RS-232-C[あーるえすにーさんにしー]
シリアル通信インターフェースの1規格．いまひとつ美しくないのに何故かポピュラになってしまった規格で，アマチュア的にはこれでかなりの仕事ができる．特殊な信号電圧のためのレベル・シフトICも多く出回ってきたので，UARTとペアにすると，簡単にシリ

アル通信を実現できる.

SCSI[すかじー](Small Computer System Interface)
パソコンの周辺拡張用バスとして，最近流行の高速パラレル通信インターフェース規格.

SIO[えすあいおー](Serial I/O)
CPUと外部のシリアル信号をインターフェースするための周辺LSI．ポピュラな8251のCMOS版の71051の場合，ここへ参入した別メーカでは，8251が二つ分入るという仕様で対抗し，さらに別メーカは，8251が四つ分入るという仕様で追随している．この涙ぐましい姿勢が，日本の半導体メーカの原点である.

TTL[てぃーてぃーえる](Transistor-Transistor Logic)
CMOSが出るまでは，ICといえばTTLであった．電流を食うといわれればLS-TTLを出し，遅いといわれればS-TTLやF-TTLを出し，改良だとALS-TTLを出し，ついにはピン配置の実績を捨ててまで高速化にこだわり，膨大なファミリとなっている．ところが世はCMOSのASICの時代であり，ブレッド・ボードとしてASIC内のゲート遅延をシミュレートするTTLがない，というほどの高速化に追随するのに必死となってしまった．さあどうなるのか.

UART[ゆーあーと](Universal Asynchronous Receiver/Transmitter)
各種のプロトコルを設定できるSIO．8251は同期もできるがここではUARTとして使っている.

筆者のひとりごと——あとがきのあとがき

はやいもので，本稿の最初のバージョンを書いてから半年もたってしまいました．この特集が世に出るのは，構想から7ヵ月以上も経過した「未来」ということになります．『情報はなまもの』といいますが，テーマがマイコン技術であるだけに，筆者としては世の技術の進歩にはらはらする半年間となりました．もともと最先端の技術を紹介するというよりも，なかば過去の技術になりつつある，地に足のついた技術の概観が目標でしたから，自分のモノになっていない「知識だけ」の内容をあえて除いていました．ところが，半年前にはまだアナウンス程度であった技術が続々と現実になり，本稿にも登場させたいような実用レベルとして出現してきました．それを横目に，つぎつぎと発表される技術をじっと受け入れるだけ，というのは筆者にとって多少つらいことだったのです.

デバイスの世界では，ますます高速・大容量なRAMや疑似SRAMが発表され，ミクロン・ルールも着々と進行しています．DSPではオーディオや画像用途のために機能を特化させた専用DSPが出現して，この分野の広がりを証明しました．新しいCISCのCPUが続々と発表される以上にRISC陣営は盛り上がり，トランスピュータのような並列CPUも定着してきました．ニューロやファジィのチップによって，理論でなく現実にパソコンと結びついたニューロ・ボードやファジィ・システムまで登場しました.

ソフトウェアの世界でも，CASE環境が相当に普及したり，Unixのゴタゴタ，ウィンドウ・システムの大流行，TRON，非常に安いEWS，そしてISDNなどのネットワーク技術の遅いか速いかわからない拡大など，話題にはこと欠かない半年間でした．NeXTコンピュータやラップトップ，またラップクラッシャの各種パソコンの動向も，AXを除いては面白いものです．いちいち各個人が追跡しきれない量の情報，というのはハードに限ったことではなかったようです.

そしてまた，筆者にとって一番気になるのは，本誌やトラ技を含めた各種の記事でした．ほかの雑誌や記事を一切見ないで一気に原稿を書き上げて，その後いろいろな機会に本誌のバック・ナンバを見てみると，本稿と部分的に同じような特集をあちこちに発見したり，この半年間に毎月出てくる本誌やトラ技の記事に似たテーマがあるとドキドキしたりしました．技術に関する話なので，同じ結論はどういっても同じなのだ，と開き直って妙に納得したりもしました.

いずれにしても，この特集はまさに筆者のオリジナル，それも最初の作品であることに間違いはありません．理解の不足や説明の不備，あるいは視野の狭さについての色々なアドバイスを受けて，筆者自身がエンジニアとしての成長の糧としたい，というつもりで取り組んだものです．そして，あるものは無意識的にまたあるものは意識的に，行間に隠れてしまっている部分もかなりあって，この点での指摘も秘かに楽しみにしています.