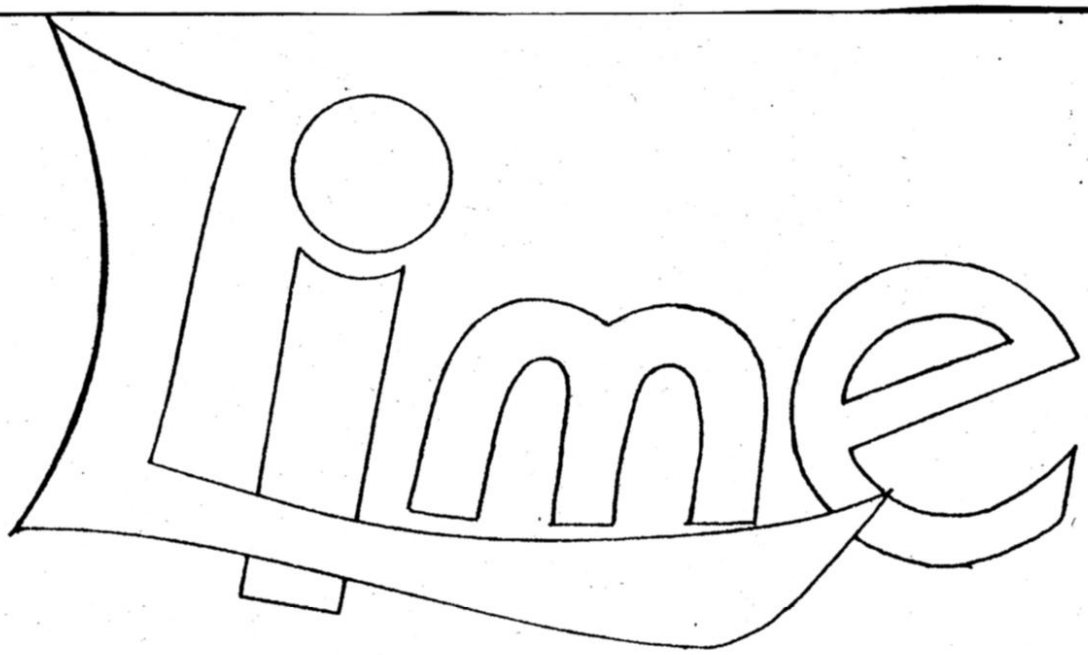

~ Limited Expression Report ~

1994. 11

K.I.T
Computer Club

・目次

今どきのマイクロ・プロセッサ94

竹岡@*AXE*, Inc.

0.始めに

IBM互換機遊びはもう飽きた。
今度は、変なCPUで遊ぶのだ。
変なCPUと言っても、MicroProcessorだ。
所詮はマイコンのくせに、近頃のマイコンは速い。大型機が滅びて無くなってしまうぐらい速い。

1.ARM

例えば、ARMというMPUがある。
ARMは3DOのゲーム機（松下版:3DO-Real)に入ってる。キャッシュが効いてれば15Mipsぐらい出る。ついでに、AppleのNewton(いわゆる携帯情報端末だが、Zaurusにはほど遠く、全然売れていない)にも入っている。
15Mipsはいまでは、ごく普通の速さだ。でも、ARMは単位電力あたりのMips値という怪しげな単位を作って、「Mips/Wでは、一番だ」と強気に出ていた。それというのも、携帯端末Newtonに採用されているからだ。
ARMというのは、ほとんどイギリスの国策会社で、ハイテク振興のために国がバックアップしている。
で、ARMのアーキテクチャは、これもイギリス製のTransputerとよく似た部分がある。例えば、命令フィールドが機能ごとにきれいに分割されていて、一命令で加算やシフトなど、複数のオペレーションが同時に可能だ。
これは、70年代のミニコンピュータに時々あったノリだ。しかし、一命令で加算もシフトもできるという事は、シフトしないときは、シフト指定フィールドが余ってしまうという事だ。
それはまだ、我慢できる。きょうび、我慢できないのは、加算とシフトを１命令でやらなければならないので、１クロック中の回路遅延が大きくなるか、パイプラインの段数を増やさねばならない事だ。これは、ちょっ

と苦しい。ARMはTransputerと同じく、シンプルさを身上、信条、思想として、もっとも重視しているからだ。(とはいえ、Transputer T10000も、むちゃくちゃなSuperScalarマシンになったが)

でも、それもまだ我慢できる。もっとも我慢できないのは、条件判断がほとんどの命令について回る事だ。条件判断は、一つ前の演算命令の結果を利用する。演算結果は、確定するのがすごく遅い。だから、パイプラインを行なっていても、直前の結果を利用するのは、非常に辛い。(特にCarryがめちゃ遅い事は、ちょっと考えればすぐわかるだろう)

実際、最近のまともアーキテクチャの高速RISCは、なるべく条件判断を普通の命令から遠ざけている。

例えば、Mipsアーキテクチャはその初期から条件フラグをすべて廃止し、演算命令と、条件判断命令間に依存関係が出る事を、無くしてしまっている。(Carryフラグがないので、多倍長演算が遅いのは、ちょっと悲しいが)

というわけで、ARMは70年代、イギリス的マニアックさはあるが、90年代的速さには、すでにおいてきぼりを食らったアーキテクチャだと言えるだろう。

さらにどうでもいい、うんちく:

最初、AppleはNewtonをHobbitというAT&T(の子会社)のMPUでやろうとし、Appleも設計に口出しをしていた(Hobbitというネーミングも Appleのコード名)。しかし、Hobbitはいつまでも出てこないので、Newtonは見きり発車して、ARMを採用してしまった。

Hobbitはというと、AT&Tの携帯端末の MPUとして採用され、EO(とGO)のソフトウェアを動かしている。(が、この秋、 AT&Tの携帯端末は店じまいし、Motorolaの携帯端末を採用する事が発表になった)

Hobbitも15Mipsぐらい出る。

また、ARM採用の3DOも、CPUパワー不足には勝てず、次期3DOはPowerPCを採用すると発表している。3DO社は単にApple社に影響を受けたオタクなのかも知れない。

わが社AXEはARMにはちょっと詳しい。ARMの開発環境にも詳しい。なぜ、詳しいかはここには書けない。:-P ゲーム機には関係無い。:-)

なお、ARMはbig-endianもlittle-endianも使えるが、開発環境のデフォールトはlittle-endianである。(さすが、イギリス人もわかっている)

2.NexGen

NexGenというのは、IBMもASCIIも肩入れしている、ナゾなMPUであろる。

NexGenは3チップぐらいのチップセットで構成され、ハードウェアとソフトウェア(と、多分マイクロコード)による、エミュレーションで、Pentiumと同じプログラムを Pentiumと同等以上のパフォーマンスで実行すると言われている。

しかし、今どき、3チップにバラバラに存在しているものが、100MHzとかの速さで動けるものだろうか？ついこの間まで、NexGenは5チップだった。しかし、それではあんまりなので、IBMが後ろについて、量産するにあたって、3チップに作り直したのだ。

なぜにそんなにチップ数が多いのかと言えば、一つには、ピン数である。チップ面積はともかく、今のLSIでもっとも問題になるのは、I/Oピンの数なのである。LSIの集積度はあがって、1チップに色々なものを作り込めるようになったが、ピン数が多い、パッケージは値段が高い。300ピン近いセラミック・パッケージのICは、そのコストのほとんどを京セラに支払っているようなものなのだ。ほんと ;-)

で、NexGenはなぜか、ピン数を要求したので、パッケージを複数に分けてピン数を稼いでいるのだそうだ。(データ・シートも見た事ないので、伝聞でしか書けない)

NexGenというのは、外部データバス幅が128bitぐらいあったかも知れない。そういう記述を日エレかSuperASCIIで読んだかも知れない。ひょっとすると、もっと多かったかも知れない。

ぼちぼち、IBMの工場で量産開始らしいので、本当に所定の性能が出るかどうか楽しみなところだ。

また、Pentium用のソフトウェアは一度、変換しないとかからない様な記述も見た事があるのだが、 WindowsやDOSのアプリケーションがどれだけNexGenで動くかも楽しみなところであろう。

ちなみに、DOS/Vマガジン11月号には、NexGen採用システムの広告が載った。

さらにどうでもいい、うんちく:

NexGen自体は一昔前から仕事をしていたベンチャーなのだが、最近になって、いきなり、IBMがでてきたのには驚いた。(ASCIIは前から出資していた)

Windowsはまともに動かなくて、OS/2 warpとAIXのための機械とかになるんじゃないだろうな？？:-P

にしても、わけわからんアーキテクチャが、IBMなんかに応援されるのはいいことだ。:-)

また、AMDがK5というRISC技術とOut of Order処理を採用した、Pentium互換プロセッサを作る事を表明しているが、RISCでPentiumをやるということは、マイクロ・コードが一杯走るんじゃないの？ということで、どこか、NexGenと似ている部分があるんではないかなぁ、と、思うのだが…

ちなみに、CyrixもSuperScalarのPentium互換プロセッサを言っていたが、Cyrixには、あれは作れまい。(100年後には可能だろうけど)

3. SH-2

SEGAの次世代(って、もう売り出されるから、今世代?)ゲーム機Saturnに積まれて、アホでも知ってるMPU。

3DOに載ってるARMは知らないのに、SH-2を知ってるなんて、なんか変。

SH-2は、日立が地道に、H-8, H-16, とやってきた、Hシリーズの最新版とも言える。

H-8もH-16も、68000の様な馬鹿げたレジスタ構成をとらない、素直なマシン。なかなかにタチが良い。

SH-2は、組み込み用には珍しく、積和演算をちゃんと流すように設計したらしい。で、Saturnのポリゴン処理は、実は、ほとんどがSH-2のソフトウェアで処理されていて、SONY PlayStationとは、異なった味を持つ。

SH-2は、シンクロナスDRAMを使うように設計されているが、実はSH-2(Saturn)用のDRAMには、日立のメモリ屋さんがSH-2の特注モードをいれていて、MPUが効率良くメモリ・アクセスできるようになっている。(さすがに、この不景気から脱出するには、ゲーム機だ！と、言われるだけのことはある)

外部キャッシュを持たなくても、かなりな性能が出せるようなので、なかなかに結構なMPUである。

SH-2は、公称25Mipsだし、大量産されるので、みんな、値段も安くなると見ており、組み込み屋さんにも、楽しみなMPUとなっている。

さらにどうでもいい、うんちく:
上記、全部うんちくなので、他には何もない。日立は、big-endianが好きだ。わかってない。日立は、HPのPA-RISCもやっているはずだが、実体は見た事ない。

4. HP PA-RISC

PA-RISCの最上位チップは、１日ラインを動かして１個しか取れない歩どまりだ、と聞いた事がある。
でも、PA-RISCの最下位ワークステーションは30万円ぐらいで売られているので、まぁ、量産もされているのであろう。
現行のPA-RISCよりも、将来1000Mipsになるという、口からでまかせの将来版PA-RISCの方が気になる。
VLIWで、1000Mipsを実現すると言うけれど、一体VLIW命令も、内側の演算器を満たすためのデータもどこから出し入れするというのだろう。今の常識的な１チップのLSIのピン数では、とても足りないと思うのだが…1999年頃には、解決できているだろうか？(可能性は十分ある)

さらにどうでもいい、うんちく:
ちょっと昔、HPはApolloを買収した。Apolloは社運をかけて、PrismというRISCアーキテクチャを開発していた。もとのHPのRISCはかなりショボかったが、PA-RISCは結構良くなった。PA-RISCにどこまで、Prismの影響があったのかは知らない。
VLIWは誰が見てもコンパイラの勝負である。HPがどんなコンパイル技術で、幅広いVLIW命令フィールドを埋めるのか、楽しみである。が、1999年頃には、誰でも、それなりのVLIWコンパイラを作れそうな気がする。:-)

5. PowerPC

PowerPCは、今じゃ、Macintoshに採用されて、Pentiumの次にミーハーなMPUなので、どうでもいいや。
PowerPCはよく考えられている。
Motorolaは大馬鹿だが、IBMはやっぱり偉い。PowerStationも速いぞ。
セグメントもあるし、Byteはbig-endianだし、bitはlittle-endian(MSBがbit0)だし、IBMのメインフレームみたいでかっこいいぞ。
相対ジャンプの番地計算が、ジャンプ命令そのものを起点にしているのも、なんだかおしゃれ(かも知れないが…謎 :-))。

さらにどうでもいい、うんちく:
AIXは、UNIXのくせに、Multics流のシングル・レベル・ストアという、単一アドレス空間を採用している。セグメントもあるし、IBMのこだわりも、なかなか、意地があっていいね。(若もんも、Multicsぐらいは勉強せんといかんぞ)
ちなみに、PowerPC版WindowsNTはLittle-Endianで動いているらしい。
DOS/Vマガジン12月号には、PowerPC採用システムの広告が載った。
わがAXEはPowerPCについても、ちょっと詳しい。

6. DEC Alpha

こんなの良い子のみんなは知らんだろ。DECっていう会社は、この間まで、IBMに継いで、世界第2位のコンピュータ会社だった。
UNIXが初めて動いたPDP-11もDECのミニコンだし、仮想記憶で有名なVAXも DECのだし、BSDで仮想記憶が最初に実装されたのもVAXだ。ついでに、人工知能研究によく使われたDEC20X0もDECの大型ミニコンだ。
という、DECも、今じゃ、素人にEpsonのIBM互換機 販売会社？などと、言われるしまつ。
DEC AlphaはDECの技術屋がMipsに負けないように、心機一転、高速マシンを目指して作っているMPUだ。
AlphaとPowerPCをみていると、もう、バラバラの部品では、高速な計算

機は作れない時代だな、と、しみじみ思う。
Alphaは100MHz, 150MHz, 250MHzと、そのクロックをどんどん電波の領域に持っていっている。こんなクロック、引き回すだけでも大変だ。もはや、チップの外に出して引き回す信号ではない。
ということで、計算機を作りたいみんなは、LSIの設計をマスターしようね。
Alphaのアーキテクチャは、2000年まで持つものだそうで、アーキテクチャ・マニュアルを読んでも、当たり触りないことしか書いてないので、あんまり面白くない。

さらにどうでもいい、うんちく:
Alpha(α)というネーミングは、DEC内で別にOmega(Ω)という計画があったから、単にその反対で付けただけとの事。
WindowsNTを作ったのは、DECのPDP-11のOSであるRSX-11を作り、VAX/VMSを作った人である。
DECはMipsにも肩入れしていたので、WindowsNTがMipsでもAlphaでも動くのは、もっともなことだ。
ちなみに、DOS/Vマガジン11月号には、Alpha採用システムの広告が載った。

7. 終りに

Sparcは嫌いだ。Register Windowというのは、SuperScalarの足を引っ張る、最悪な代物だ。近々、滅びるであろう。
DOS/Vマガジン12月号には、Mips採用システムの広告が載った。Sparcなんか滅んで、Mipsになればいいんだ。ちなみに、Sparc用のWindowsNTもどこかの会社がやっているそうだが、SparcはBig-Endianのみなので、移植が難しいのかも知れない。
PowerPCは、Macのおかげで、商売的にも成功できるかもしれない。まともなものが、ハヤルのは、なかなかにいいことだ。
あたしは高級言語計算機であるとか、Tagアーキテクチャとかは、好きなので、1990年代後半には復活して貰いたいものだ。(チャンスは十分あるはず)

全然関係ないが、あたしはDECが大好きなので、DEC Alphaを応援している。でも、それでどうなる訳でもない。:-)
DEC Alphaが西暦2000年になるまでに、世界の終りが来なければいいのだが…
(その前にDECの終りが来ない事を祈るべきかも知れないが… :-P)

(1994/Nov/19 たけおか しょうぞう AXE,Inc)

Free UNIXでのWWW導入

中村 豊

1. はじめのはじめ

8月頃の私
今年の学祭何しようかな～。
やっぱり、今流行のマルチメディアか？
ほんだら、いっちょうMosaicでもインストールしてみるか。

先輩Fさんとの会話
私：Mosaicのソースってありませんか？
F：Mosaicすんの～？MosaicってMotifがいんねんで～。
私：え～そんなんないやんけ～。Motifいなんやつでmosaicみたいなもんってないんかな～。
F：さぁ。知らんわ。

9月頃の私
先輩S1さんとの会話
私：学祭でMosaicしよーと思ってるんですけど、MosaicってMotifがいるでしょ。Motifがいなんやつでmosaicみたいなんってないですか？
S1：あるで。chimeraっていうのがそれや。これはMotifいなんで。ここは、chimeraで、キメラな…
私：しらしら～……まぁ、とりあえずそれのソースってありますか？
S1：あるよん。ここにあるし持っていったら？
私：わーい。
S1：でも、それインストールするんやったらhttpdをインストールせんなあかんし。まぁそれのソースもあるし持っていったら？
私：はいはい。
S1：chimeraは日本語パッチもあるし結構いいで。
私：わーい。やったね。

2. 導入の始まり

こうして、私はＳ１氏のおかげでWWWの入口となるchimeraのソースを手に入れることができた。しかし、これが怒涛の２０日間の入口になろうとは誰が予想しえただろうか。。。

ところで、今回導入の対象にしたシステムは、

CPU　Am486DX2processor 66MHz
Memory　8Mbytes
HDD　540MBytes SCSI
OS　FreeBSD 1.1.5.1-RELEASE

というものである。

Ｓ１氏にもらったソースは次の通りだった。

chimera-1.53.tar.gz	chimera本体
chimera-1.53-special.patch	スペシャルパッチ
chimera-1.53-kanji.patch	漢字パッチ
NCSAhttpd-1.3.tar.gz	httpdのソース

さて、chimeraのインストールの前にhttpdをインストールしなくてはならない。というわけで、早速インストールを行なった。詳細は、UNIX USER１０月号にのっているので、そちらを見ていただきたい。これは結構簡単にインストールできた。

次に、chimera本体のインストールである。最終的に日本語の表示ができなくてはならないが、とりあえず始めは、英語環境でインストールを行なった。これは、附属のINSTALLというファイルに詳細がのっている。

ここまではソースをもらってから２日で終わった。結構サクッと終わるのでは？とたかをくくっているとトンでもない問題に遭遇するのであった。

3. 怒涛の２週間

さて、英語環境ができたので、次は日本語環境である。日本語を表示させるには、パッチを当てなくてはならない。これは非常に簡単で、パッチを当てる順番さえ間違えなかったらＯＫである。スペシャルパッチ・漢字

パッチの順番にパッチを当てればもうそれでおしまいである。
この後、英語環境と同じ手順でインストールを行なった。全く問題がないように見えた。

さぁ、実行して日本語でも見てみようかな〜。と、胸踊らせていると、なんかメッセージが出てるやんけ。まぁ、とりあえず無視して日本語のファイルを見てみたら。げげーん。文字化けしてるやんけー。こらあかん。
さっきのメッセージを見てみたら、なになに？

Cannot setlocale to ja_JP.ujis

なんのこっちゃ、さっぱりわからん。

このことをＳ１さんに質問にいった。

私：こんなメッセージが出て日本語がでーへんやけど。
Ｓ１：なぁーるほど。これってlocaleディレクトリの設定が間違ってんちゃうか。どこにlocaleディレクトリがあるん？
私：さぁ。(適当に探す)
私：あった。ここや。/usr/share/locale や。
Ｓ１：ほんだらそこの下のJa_JP.EUCに変えて設定してみたら？
私：おお、メッセージがなくなった。と思ったら、別のエラーメッセージが出て止まってもーた。
Ｓ１：んん？なんじゃ？こんなエラーメッセージ見たことないぞ。
私：げげーん。どーしたらいいんじゃー。
Ｓ１：うーん。これはどうやらFontの指定が間違ってるようやし、自分の持ってるFontに変えて設定してみたら？
私：うーむ。ほんだらてきとうに変えて、、、ふー、やっぱりあかん。
Ｓ１：うーん。これは根が深そうやなぁ。

これからchimeraの解析が始まった。
とりあえずFontよりは、localeの方が怪しそうだった。というのも、漢字パッチの中に自前でsetlocaleという関数があった。それで、そいつを生かすようにしても駄目だった。
ふーむ、じゃあlocaleの中のLC_CTYPEというファイルを作るコマンドがあるし(mklocale)、それで、自前でlocaleを作ったろか。と思って、SunOSの

localeのソースファイルの真似をしてLC_CTYPEの元になるファイルを作った。するとどうだ？SunOSには、国際化機能がどうとかと書いてあるのに、FreeBにはその部分がないやんけー。

まぁ、それでもとりあえず作ったlocaleで試してみた。でも、やっぱり駄目だった。

次に、組み込み関数のsetlocaleは何をやっているんだ？と思って、組み込みのsetlocaleの関数の中をずっとたどっていった。するとどういうわけだ？Ja_JP.EUCという単語が出てこないじゃないか。これはもしかして……

man　setlocaleとして、じっくり眺めていると、

"setlocaleという関数のLC_CTYPEは、現在CとPOSIXしかサポートしていません。"

と書いてある。なんじゃとー！関数の中にも、

"いつかは、デフォルトのCの設定をリセットする必要があるだろう。でも今はそれらをリセットする方法がない。"

とコメントがしてあった。

へなへな～。要するに、日本語環境をFreeBSDはサポートしてないゆーことか。

この事実に到達するのに、約１０日かかった。

4. 徹夜の２日間

上の事実の報告と、その対策を一緒に考えようとＳ１さんのところに行った。

私：どうやらFreeBSDでは、日本語環境をサポートしとらんようですは。
Ｓ１：なにー。それって、かなり辛いんとちゃう。どうやったらいいんやろ。
私：ktermの真似して自前でパッチ当てるしかないんかなー。でも、これはあまりに厳し過ぎる。
Ｓ１：それしかないんかなー。でも、もしそれやるにしても今からやった

って間に合わんと違うか？そんなん１ヶ月やそこらでできへんぞ。
私：確かにその通り。
Ｓ２：何の話ししてるん？
Ｓ１：localeが日本語をサポートしてないＯＳで、日本語出すのってどうしたらいいんでしょうか。
Ｓ２：Xのバージョンは？
私：R6です。Ｆさんから、バイナリキットをもらったんです。
Ｓ２：X_LOCALEとかXI18Nとか定義した？
私：は？
Ｓ２：それをやらんと日本語出んやろ。
私：はぁ
Ｓ１：OSがサポートしてなくてもいいんですか？
Ｓ２：R6は、自前でlocaleを持ってるからいけるで。
Ｓ１：へぇ～
私：ふーん
Ｓ２：まぁ、X_LOCALEとかXI18Nを定義してXをコンパイルしなおすことですな。
私：はぁ。でも、ソースがないんですけど。
Ｓ２：どっかからとってきたろか？
私：お願いします。
Ｓ２：あと、ほら、このニュースに日本語環境について書いてあるやろ。これも持っていき。
私：どうもありがとうございます。

こういうわけで、chimeraの日本語を表示するという目的のためXwindow-systemのコンパイルを行なわなくてはならなくなった。なんだかな～。

さて、Ｓ２さんにもらったニュースの記事と、附属のREADMEを見ながらXのインストール作業を開始した。
ニュースの記事を読んでいると、DX2/66で、コンパイルするのに３時間半から４時間かかると書いてあった。げろげろ。なんじゃそらー。

そういうわけだから、コンパイルを始めてほったらかしににて寝ていた。するとどうだ？なんか様子が変だ。げげーんＯＳが落ちてるやんけー。一体どうなってるんや。もう一回やり直しや。

1回目失敗・約5時間

コンパイルを再開して、漫画を読んでいた。あれ？また止まってる！

2回目失敗・約2時間半

メモリ不足かなぁ。とりあえずいなんサーバーとデーモンと切ってコンパイルを始める。

3回目コンパイルは成功

ふう、やっと終わったか。make installしてと。あれ？エラー続出やんけ。コンパイル中のエラーメッセージのところで引っかっかとる。くっそー。もう一回やり直したる！

インストールでこける・約5時間

エラーの出ていたdefineを切ってコンパイルを始める。ちゃんといってくれよーと思いつつ昼飯を食いに行く。帰ってきたら……へろへろ～

4回目失敗・約2時間半

なんでやねん！(すでに半泣き状態)もう一回や！

5回目失敗・約2時間半

かんべんしてくれよ～。たのむで～。(泣きかけ)

6回目コンパイル成功・インストールも成功

ただ一つフォントのインストールに失敗したが、バイナリパッケージからコピーした。・約6時間

Xをコンパイルしている間に、OSが4回も落ちた。原因は良くわからなかった。上にもかいたが、メモリ不足の可能性を考えて、いらないサーバーやデーモンを切った。それしか原因が思いつかなかった。

この作業の後新しく作ったXを立ち上げて、chimeraをインストールしなおしたら、何かメッセージが出たけど、とりあえず日本語が表示された。

感動や。まさに感動や。(これが夜中の3時から4時ぐらいだった)

5. おわりに

結局、今回は、なんとかFreeUNIXにWWWを導入することはできたが、問題点も残った。

(1) X11R6のコンパイルを行なったがこれがどういうわけか、一回目の起動では立ち上がらないが、二回目ではちゃんと起動する。これはちゃんと直したいが、時間がなかった。
(2) WWWと言ってはいるが、インターネットに継っていないのはやはり淋しい。
(3) ハイパーテキストを書く時間がどれだけあるか。ここが一番重要なところなのに、時間をかけることができなかった。

(1)と(3)は時間をかければ良くなるだろうが、(2)は学校の事情があるので、実現するのはかなり厳しいということを感じた。

イベントシステム

武田雅也

０．始めに

あるプログラムで、例えばＲＰＧ（ロールプレイングゲーム）などで、シナリオが大きくなればなるほど、実行プログラムが大きくなるようでは困る。そこで、シナリオなどの部分は実行プログラムと分離してしまおうというときに、このイベントシステムは利用できる。こうすることにより、システムは基本的にシナリオの大きさに依存せず、ただ別に用意されているシナリオファイルを制御するだけでよいことになり、その大きさは変化しない。その他、いろいろな使い方もあると思われる。

１．仕様

基本データサイズは３２ビットで、アドレス空間は0000:0000～FFFF:FFFFとなる。

２．event命令

このシステムの特徴として、event命令がある。実際の使用例は、

＜call命令の場合＞

```
 call  func_Nothing
 halt
func_Nothing:
 nop
 ret
```

＜event命令の場合＞

```
public  func_Nothing
 event  str_func_Nothing
 halt
str_func_Nothing:
 db  "func_Nothing", 0
func_Nothing:
 nop
 ret
```

以上、上の２つのプログラムは同じ動作をする。このように、event命令はラベル名が格納されているアドレスをオペランドに持つ。よって、そのラベルは外部参照可能ラベル（public）でなければならない。これは別モジュールにあってもよい。下の例である。

```
; main.asm
 event  str_func_Nothing
 halt
str_func_Nothing:
 db  "func_Nothing", 0
; sub.asm
public  func_Nothing
func_Nothing:
 nop
 ret
```

実行時にラベルを検索するので、もし存在しない場合は実行時にエラーとなる。

また、イベントシステムに存在しなくても実際のプログラムで登録することもできる。例えばＣ言語で例を示す。

```
; main.c
void Nothing(void){}
void assign(void){
 EV_Assign("func_Nothing", Nothing);
}
; main.asm
 event  str_func_Nothing
 halt
str_func_Nothing:
 db  "func_Nothing", 0
```

これが、event命令の使い方である。

３．セグメント

すべての命令およびデータ（実際、命令とデータの区別はない。）は、どれかのセグメント内に存在する。

セグメントには、セグメントの種類とセグメント番号がある。

セグメントの種類には、命令があるcode、定数があるconst、変数があるdataに分かれる。

セグメント番号は、0以上の整数である。

実行ファイルはセグメント番号0から各セグメントcode、const、dataの順で構成される。

ただし、別にcodeに変数領域を確保しても、constに命令をおいても全く問題はない。

4．命令セット

イベントシステムの命令セットは、下のように分けられる。

制御：（halt、nopなど）。

転送：（mov、push、popなど）。

算術：（add、sub、mul、div、cmpなど）。

論理：（and、or、xor、not、shl、shrなど）。

分岐：（jmp、条件jmp、call、event、retなど）。

5．プログラミング言語とのインタフェイス

ここではC言語を使って、説明する。

typedef void (*EV_Function)(void)：　イベント関数のインタフェイス。

void EV_Init(void)：　イベントシステムの初期化。

void EV_Free(void)：　イベントシステムの終了。

void EV_Open(char *filename)：　実行ファイルのオープン。

void EV_Close(void)：　実行ファイルのクローズ。

void EV_Segment_Load(unsigned short no)：　セグメントのロード。

void EV_Segment_Free(unsigned short no)：　セグメントの解放。

void EV_Assign(char *name, EV_Function func)：　イベント関数の登録。

void EV_Erase(char *name)：　イベント関数の削除。

unsigned long EV_SymbolAddress(char *name)：　publicシンボルのアドレスを得る。

void *EV_RealAddress(unsigned long offset)：　実際のアドレスに変換する。

unsigned long EV_Reg_Read(unsigned short no)：　レジスタの値を得る。

void EV_Reg_Write(unsigned short no, unsigned long data)：　レジスタの値の設定。

void EV_Exec(unsigned long start)：　イベントシステムの実行。

6．実行ファイルの作成

今回は、ソースファイル（.asm）をアセンブリ言語で作成し、それからアセンブラでオブ

ジェクトファイル（.o）を作成し、最後にリンカで実行ファイル（.ev）を作成するという場合について考えていく。

7．実行ファイル（.ev）のフォーマット

スタックサイズ
セグメントの数
セグメント列
　＜セグメントの実際＞
　セグメントサイズ
　publicシンボルテーブル
　　＜publicシンボルテーブルの実際＞
　　publicシンボル数
　　（publicシンボル名、オフセット）の列
　code、const、dataサイズ
　セグメント本体（code、const、dataの順）
　再配置アドレステーブル（セグメント数だけある）
　　＜再配置アドレステーブルの実際＞
　　再配置アドレス数
　　（オフセット）の列

8．オブジェクトファイル（.o）のフォーマット

スタックサイズ
セグメント数
セグメント列
　＜セグメントの実際＞
　セグメントサイズ
　セグメントの種類
　セグメント番号
　publicシンボルテーブル
　　＜publicシンボルテーブルの実際＞
　　publicシンボル数
　　（publicシンボル名、オフセット）の列

セグメント本体サイズ
セグメント本体
再配置アドレステーブル
＜再配置アドレステーブルの実際＞
再配置アドレス数
（オフセット）の列
外部参照アドレステーブル
＜外部参照アドレステーブルの実際＞
外部参照アドレス数
（外部参照シンボル名、オフセット）

9．ソースファイル（.asm）のフォーマット

以下に、擬似命令を示す。

end：　ソースファイルの終了を示す。（別になくてもよい）

（例）　end

stack size：　スタックサイズを+sizeする。

（例）　stack 256

segment type no：　セグメントの開始を示す。typeにはcode、const、dataがあり、noは0以上の整数である。

（例）　segment code 0

ends：　セグメントの終了を示す。

（例）　ends

public name：　publicシンボルの宣言。nameはこの命令のセグメント内のラベルである。

（例）　public func_Nothing

extern name：　externシンボルの宣言。nameはpublic宣言されているラベルである。

（例）　extren func_Nothing

db data{, data}：　byte型データ領域の確保。byte型とは１バイトである。

（例）　db [100], "func_Nothing", 0

ds data{, data}：　short型データ領域の確保。short型とは２バイトである。

（例）　ds [20], 4096

dl data{, data}：　long型データ領域の確保。long型とは４バイトである。

（例）　dl [8], func_Nothing, 128000

次に、１行のフォーマットを示す。

[ラベル:] [命令又は擬似命令] [オペランド列] [;コメント]

オペランドには下の８種類が許される。

type 0：　レジスタ

（例）　r0, sp

type 1：　値

（例）　123, 0x12fe95, func_Nothing

type 2：　[レジスタ]

（例）　[r7], [bp]

type 3：　[値]

（例）　[256000], [0x400], [func_Nothing]

type 4：　[レジスタ+レジスタ]

（例）　[r3+r4]

type 5：　[レジスタ+値]

（例）　[bp+6]

type 6：　[レジスタ+レジスタ+レジスタ]

（例）　[sp+r1+r2]

type 7：　[レジスタ+レジスタ+値]

（例）　[bp+r0+100]

また、オペランドは３種類の型がある。byte、short、long型である。オペランドの前に明示的に、型宣言することにより指定が可能である。省略するとlong型である。

（例）　(byte)100, (short)[bp+12], (long)0

レジスタの型宣言はレジスタ番号の後ろにつけるアルファベットで可能である。

（例）　r0:long型, r0h or r0l:short型, r0a or r0b or r0c or r0d:byte型

ただし、r0=r0h*0x10000+r0l、r0h=r0a*0x100+r0b、r0l=r0c*0x100+r0dである。

以下に、ソースファイルの例を示す。

```
stack 100
segment code 0
 push 100
```

```
 event str_func_Print
 add sp, 4
 halt
str_func_Print:
 db "func_Print", 0
ends
end
```

１０．アセンブラ

アセンブラの仕事は、１つのソースファイル（.asm）から１つのオブジェクトファイル（.o）を作成することである。

その主な仕事は、命令の翻訳と、シンボルテーブルの作成である。

命令の翻訳は、必要ならばオペランドも含めて、１対１に変換していくだけである。

シンボルテーブルの作成は、publicシンボルや再配置アドレス、外部参照シンボルなどのテーブルの作成である。

public宣言されたラベルは外部参照されたときのため、そのセグメント内でのオフセットを求めなければならない。よって、public宣言したラベルがそのセグメント内に存在しなければそれはエラーである。

オペランドにラベルを使用している場合、それをアドレスに置き換えなければならないが、そのラベルがそのセグメント内に存在しなければ、extern宣言されているラベルかどうかを調べる。もしも、そうならばそのアドレスがわかった時に書き込むということにしておく。よって、もしも他の全セグメントのpublic宣言されたラベルの中にもないのなら、それはエラーである。

セグメント内のすべてのアドレス部は、仮にそのセグメントの先頭が、アドレス0000:0000から始まっているものとしている。よって、それらは実際そのセグメントの先頭アドレスがわかったときにすべて書き換える必要がある。それを再配置アドレスという。

以上がアセンブラの仕事である。

１１．リンカ

リンカの仕事は、１つまたは複数のオブジェクトファイル（.o）から１つの実行ファイル（.ev）を作成することである。

その主な仕事は、セグメント番号の同じセグメントを集め、セグメントの種類の同じセグ

メントを連結し、それをcode、const、dataの順に連結するが、そのときに外部参照しているところは、それらのアドレスがわかるはずなので書き換える。publicシンボル、再配置アドレステーブルはそのセグメントの先頭アドレスを0000:0000と仮定して書き換える。

できあがる実行ファイルは、セグメントの集まりで、各セグメント内の全アドレス部はあるセグメントの先頭アドレス0000:0000からのオフセットで表されているので、ロード時には参照しているセグメントの先頭アドレスによって、すべて書き換えなければならない。
そのときそのセグメントの先頭アドレスがわからなければ（まだロードされてない）、それはエラーである。

以上がリンカの仕事である。

１２．ロード

実行ファイル内の任意の番号のセグメントがロードできればよい。ただし、前の節でも述べたが、そのセグメントが他のセグメントを参照している場合、そのセグメントはすでにロードされていなければならない。

ロード開始時には、そのセグメントの先頭アドレスは決定しているので、それを元にpublicシンボルのオフセット値を書き換える。そしてセグメント本体をロードしたら、再配置アドレステーブルを元に、必要となるセグメントの先頭アドレスにより、全アドレス部を書き換える。それで、ロード終了である。

１３．システムの実行

セグメントをロードし、プログラムカウンタに値を設定し、命令の実行を仕様通り繰り返していけばよい。今回は、その実現方法の説明は省く。

１４．最後に

このシステムにより、このシステムの実現可能なすべての環境で、実行ファイルはその動作環境に依存せず、動作可能である。

今回の場合、実際このシステムを構築したのは、ＭＳ－ＤＯＳ上だったのでちょっとめんどくさい部分があった。３２ビット基本のＯＳ上でなら、容易にこのシステムの実現が可能であろうとおもわれる。

また現段階では、ソースファイルにアセンブリ言語を用いてるが、最終的には、例えばＣ言語のようなソースファイルをコンパイラにかけてアセンブラファイルを出力し、それから実行ファイルを作るようにしたい。

*Visual Basic*による *Windows*プログラミング

樫東 清貴

０．はじめに

これまで*Windows*用のプログラミング言語といえば、ＣやＣ＋＋といった言語が主流でした。しかしこれらは日曜大工のように*Windows*プログラミングを書くといった気軽な作業には不向きです。しかし「*Visual Basic*」(以下ＶＢ)という名前からしていかにもとっつきやすそうな、誰にでも簡単に*Windows*アプリケーションがつくれるんじゃないかと思わせてくれるようなプログラミング言語の登場によって、実際非常に簡単に*Windows*アプリケーションを作れるようになりました。

ここでは簡単なデジタル時計を作りながら、*Windows*アプリケーションが最低限備えるべき機能と併せて具体的に見ていくことにします。

１．*Windows*プログラミングの難しさとは

ところでなぜ「*Windows*プログラミングは難しい。」といわれるのでしょうか。その理由の１つとして*Windows*が擬似的ではありますが複数のアプリケーションを同時に起動することができる「マルチタスク」であることがあげられます。*MS-DOS*のような一時に１つのアプリケーションしか起動できないようなシングルタスク環境では、画面表示もキーボード入力も全てその時動作している１つのアプリケーションが独り占めする事ができます。しかし、*Windows*では複数のアプリケーションが共同で１つのハードウェアを利用するため、一定の規則が必要になってきます。つまり*Windows*アプリケーションではこの共同生活のための規則に従わなければならないという制約が出てきたことが、これまで１人で勝手気ままにくらすことのできた*DOS*アプリケーションとの大きな違いであり、*DOS*アプリケーションを作っていた人たちをして「*Windows*は難しい」と言わしめているのです。しかし実際は特にそう難しいわけではありません。

2．*Visual Basic* による *Windows* アプリケーション制作の手順

①ユーザーインターフェイスのデザイン

フォームと呼ばれる土台にコントロールと呼ばれるコマンドボタン、ラベルなどを画面を見ながら配置してユーザーインターフェイス部分を作成します。

②プロパティの設定

配置したユーザーインターフェイス部分の詳細な設定をします。たとえばラベルを作ったら、キャプションはどうするのか、フォントのサイズや種類はどうするのかといったことです。これらの属性情報をＶＢではプロパティと呼んでいます。

③ *BASIC* コードの記述

BASIC コードでプログラミングします。ただしビジュアル要素のプログラミングは①～②で終わっているのでそれ以外の部分を記述します。

①～③のうち、ＶＢの特徴的な部分が①～②です。多くのプログラミング言語ではユーザーインターフェイスなどのビジュアル要素についてもすべて各言語のコードで記述する必要があります。しかしそれは暗闇の中で積み木を積み上げるような作業であり、思ったようにははかどりません。ＶＢなら、ユーザーインターフェイスについて完成イメージを確認しながら作成することができます。しかも、配置できるコントロールは *Windows* の規格品なので見た目も操作も自然に統一されるというメリットがあるのです。

3．デジタル時計の制作

(1) *Windows* プログラミングにおけるマナーとイベント

まず、デジタル時計の仕様を考えていくことにします。*Windows* プログラムはイベントドリブン(イベント駆動)が基本ですから、自分がループをしてＣＰＵを独占して、他のプログラムが動くチャンスをなくしてしまうようなコードを書かないようにしなくてはいけません。イベントとはプログラムが動作するためのきっかけです。例えばマウスをクリックしたりキー入力をしたりするといった動作にあたり、その操作に対応したイベントはアプリケー

ションに通知され、その通知されたイベントに対応したアプリケーションの動作によってユーザに知らされます。

ではデジタル時計を作る上で、一体どんなイベントが必要でしょうか。実際に至る所にあるデジタル時計を見てみると、その動作を確認するには１秒ごとに秒数の表示が変わったり、秒表示のない時計では「：」が点滅したりするのを見るのではないでしょうか。つまり実際には常に動作しているデジタル時計でも外からは１秒ごとにしか動作しているようにしか見えません。１秒ごとにイベントを発生すれば常に時刻を管理するようなループを作る必要がなくなることがわかります。幸いパソコン自身が現在の時刻を所有しているので、「１秒ごとにパソコンが所有している現在の時刻を表示する」ことにします。

(2)　基本部分の設定

それでは２．①ユーザーインターフェースのデザインをします。これはツールボックスにあるコントロール・オブジェクトをフォームに張り付けることによって行います。「オブジェクト」とは、*Windows* アプリケーションを構成する部品のことです。ここではアプリケーション実行時には変更できない固定された文字列を表示する「ラベル」と定期的に同じ処理を繰り返す時に使う「タイマー」の２つを張り付けます。

次に２．②プロパティの設定です。張り付けた段階ではフォーム、ラベル、タイマーにはそれぞれ「*Form1*」「*Label1*」「*Timer1*」という名前が付いています。今はコントロールが１種類につき１つしかありませんからこのままでも問題はないのですが、１つのフォーム上に同じコントロールが２つ、３つとなってくると区別しにくくなってくるので、これらのコントロールの名前を変えておきます。これはプロパティウィンドウで変更するのですが、フォームとラベルに関してはこれだけでは説明不足です。フォームならデフォルトで「*Form1*」と表示されているのは「*Caption*」プロパティと「*Name*」プロパティの２ヶ所あるためです。２つの違いは *Caption* プロパティはフォームならタイトルバーまたはアイコン化したときのアイコンの下に、コントロールならコントロール上に表示するテキストを示し、*Name* プロパティはオブジェクトそのものの名前を示します。ここではフォームの *Caption* プロパティを”デジタル時計”、*Name* プロパティを” *DigitalClock* ”、タイマーの *Name* プロパティを” *NowTime* ”、ラベルの *Name* プロパティを” *SecTimer* ”とします。あと、ラベルの *Caption* プロパティを変更しないのは実行時に時

刻の表示に使うからです。

あとは表示を見やすくするためにフォントの大きさと種類を変更します。ウィンドウの大きさにもよりますが、ここでは大きさは75、種類は”*Times New Roman*”にします。また、表示を大きくしたいならフォントの種類にはアウトラインフォントを使った方がきれいに表示されます。

最後にタイマーイベントを1秒ごとに起こす必要があるので*SecTimer*の*Interval*プロパティを*1000*にします。

これでユーザーインターフェースは完成しました。あとはコーディングです。

図1　　ユーザーインターフェースの作成

(3)　コーディング

基本の最後は2．③*BASIC*コードによるプログラミングです。このデジタル時計に関してはほとんどコーディングする部分はありません。*SecTimer*の*Timer*イベントに

NowTime.Caption = Time$

と1行書くだけです。

「*Time$*」は現在時刻を*hh:mm:ss*の8バイト文字で表現した文字列を返す関数で、この返り値を*NowTime*ラベルの*Caption*プロパティに代入して画面に現在の時刻を表示しているのです。

これで十分に*Windows*アプリケーションとして動作する時計が完成しました。ただまだ少し欠点があります。実際に動作させてみると分かるのですが、

起動させてから最初のタイマーイベントが起こるまで「*Label1*」と表示されているのです。ですから起動時から時間を表示するように、*DigitalClock* の *Load* イベントに *SecTimer* の *Timer* イベントと同じ１行を加えます。

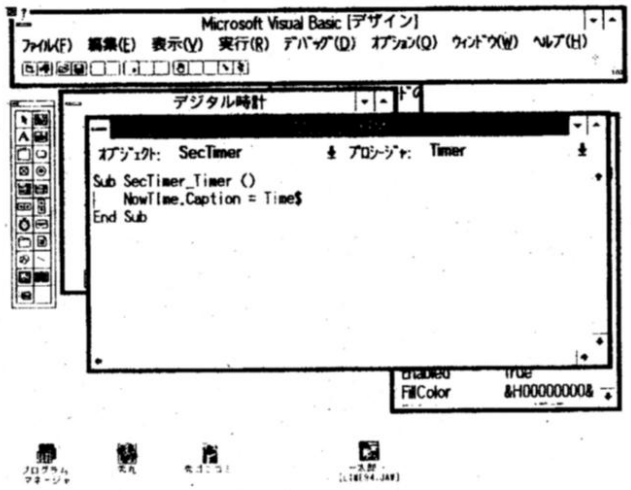

図２　　コードの記入

(4)メニューの作成とバージョン情報の表示

ここからは基本的なアプリケーションスタイルとして必要な機能を足していくことにします。(3)までで十分に時計としての機能を果たすのですが、*Windows* アプリケーションとしてはまだ不十分なところがあります。その中の１つとして一般の *Windows* アプリケーションにはプログラムについての簡単な説明を表示させるメニューが付いています。そこでこのデジタル時計にもメニューをつけることにします。メニューの作成方法は対象フォームをアクティブにしたあとでメニュー[ウィンドウ(W)]-[メニューデザイン(M)]を選択し、キャプションに”バージョン情報(&A&)...”、名前に”*MnuVer*”と入力し、＜ＯＫ＞ボタンを押すと時計のフォームにメニューが追加されます。

次に呼び出される側のフォームを作成します。メニュー[ファイル(F)]-[新規フォーム　モジュール(F)]を選択すると新しいフォームが表示されます。まずフォームの *Caption* プロパティを”バージョン情報”に、*Name* プロパティを”*Abouts*”に変更しておきます。このフォームにはアイコンをイメージボックスコントロールで、プログラム名、バージョン、著作権表示をラベルコントロール、そして確認ボタンをコマンドボタンコントロールで表示し

ます。イメージボックスにアイコンを表示するには *Picture* プロパティを設定します。ここではＶＢについている *clock01.ico* を指定します。ボタンの *Name* プロパティは”*Ok_Button*”に、*Caption* プロパティは”確認”や”ＯＫ”などにしておきましょう。

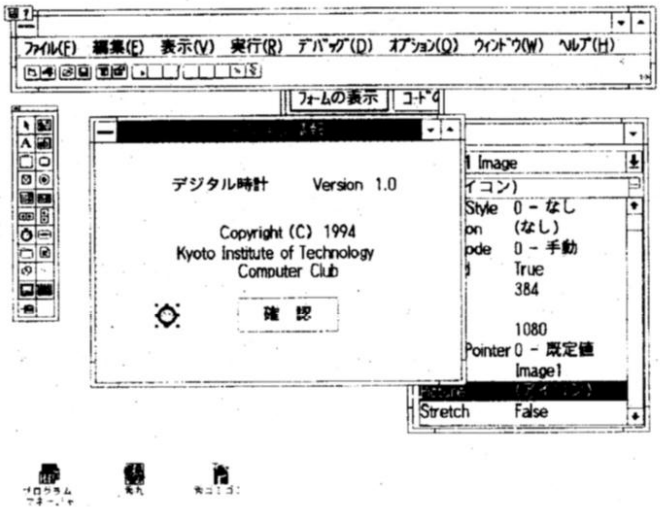

図３　　Abouts フォーム

確認ボタンを押すとこの「バージョン情報」フォームを閉じるようにしたいのでボタンの *Click* イベントのコードは以下のようになります。

Unroad Abouts

それでは今度は逆に、先ほど作ったメニューをクリックするとフォームを表示する処理を考えます。別のフォームを表示するには *Show* メソッドを使用するのですが、ここで表示するフォームが、”モーダル”か、”モードレス”かを指定します。モードレスでは呼び出したフォームと呼び出されたフォームの両方が操作可能になり、モーダルでは呼び出されたフォームが非表示かアンロードされるまで呼び出されたフォームのみが操作可能となります。普通はモーダルに設定します。そしてメニューをクリックしたときのコードは

Abouts.Show 1

となります。

最後に一般のバージョン情報やメッセージボックスは、ロードされたとき画面の中央に表示されます。そこでこのバーション情報もロードされたときは中央に表示するようにしましょう。フォームの表示位置は *Top*、*Left*、サイズは *Windth*、*Height* の各プロパティにより設定されます、通常は作成時の位置が設定されます。以下のコードは *Form* の *Load* イベントに追加します。

Abouts.Top = (Screen.Height - Abouts.Height) / 2

Abouts.Left = (Screen.Width - Abouts.Width) / 2

この方法は画面の解像度に関係なく画面中央にフォームが表示することができます。

(4)とりあえず完成

まだいくつか足りない機能が残ってたりします(*INI*ファイルのことや多重起動の防止など)が、そこまでしようとするとＡＰＩ(*Application Programming Interface*)の話が出てきたりしてしまいます。ＡＰＩに関しては本人がまだよく分かっていないのもありますし・・・。でもごく限られた知識で作られた割にはそれなりにみれるようになっているのではないでしょうか。

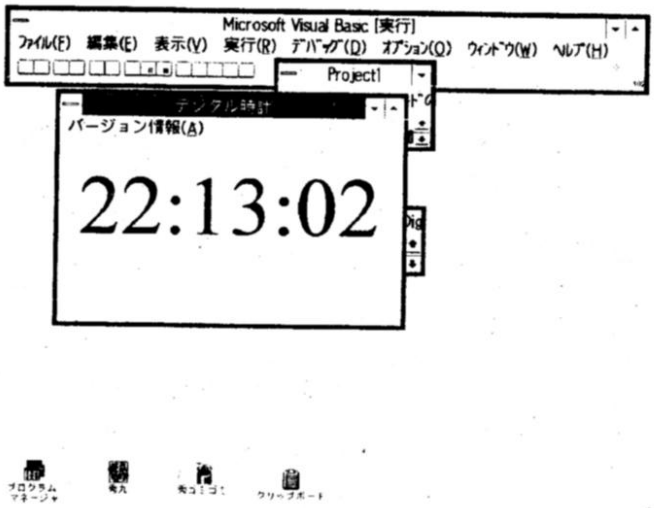

図４　　デジタル時計の実行

４．最後に

ＶＢような高度な環境となると、あらかじめ用意された機能以外のことを実現するにはＡＰＩの知識が必要になります。さらにそのＡＰＩを知っていたとしても５００種類を超えるものの中から必要なものを見つけだすのは不可能です。だからといってそれはＣなどのほかの言語で書けばいいというような問題ではありません。結局はＡＰＩの呼び出しがあることに変わりはな

いのですから。ならば、取っつきやすいＶＢから始めるのはいい選択肢１つではないでしょうか。

これを読んでＶＢを興味を持った人は、書籍などに付いている体験版なりで十分ですからすぐにやってみることをお勧めします。「こんなに簡単に*Windows*アプリケーションをつくれるのか」と思うのは間違いないでしょう。