MZ-80K/C/1200/700/1500
MZ-80B/2000/2200,X1

特集2　クリーンコンピュータと言語

T.T.L INTERPRETER

マイコン黎明期，4K BASIC，8K BASICなどという整数型のインタプリタが数々あったそうな。多少不便ではあったが，メモリにゆとりのない当時のマニアにとって，それはそれはステキな言語だったそうな。超巨大な，超肥大化したBASICやゲームが当たり前のように思っている現代の人々に，あの当時のマニアたちの楽しみをもう一度…。

```
** TTL VERSION 1.1 (C)MIKA **
#READY
A=&H8000
EXEC &H8000
#READY
A=0
EXEC &H8000
#READY
 1000 '-------------------------------
 1010 'G
 1020 "THIS IS A TINY TINY LANGUAGE"//
 1030 " FOR MZ-80K/C,1200,700,1500"/
 1040 "    & MZ-80B,2000,2200"/
 1050 "    & X1 SERIESE"///
 0

 1000 '-------------------------------
 1010 'G
 1020 "THIS IS A TINY TINY LANGUAGE"//
 1030 " FOR MZ-80K/C,1200,700,1500"/
 1040 "    & MZ-80B,2000,2200"/
 1050 "    & X1 SERIESE"///
#READY
```

開発・本文　YOU & 美香

タイニータイニーランゲージ

1．はじめに

TTL（Tiny Tiny Language）は基本的には整数型BASICを記号化してインタプリタ作成の負担を軽くしたものです。このような整数型の言語の生い立ちは，メモリが非常に高価であったころにさかのぼります。峰岸氏の連載でも書かれていますが，当時のマイコンマニアたちの夢は何よりもまず，自分の作った（そのころはまだ，マイコンの完成品などは売っていませんでした）マイコンで高級言語を走らせることでした。大学の研究所などで，お金を湯水のごとく注ぎ込めるような恵まれた環境の人を別にすると，当時のマニアに許されるメモリ容量はどんなに頑張ったとしても数Kバイトに過ぎませんでした。

この程度の容量ではもちろん，浮動小数点演算などを行うわけにはいきません。そこで生まれてきたのが，扱う数値を整数に限定し，命令も基本的なもの以外を取り払った，いわゆるTINY言語だったのです。

ダンプリストをもとに解読し，自分のマイコン用に書き換え，メモリが足りないと自分で機能を削ったり，1バイトでも短くできるところはないかと必死の思いで探したりして，みな，自分のコンピュータで動かす喜びに浸っていたものです。

メモリも安価になり，マイコンが商売になるとみた大手メーカーが台頭し，実数型のBASICインタプリタ＋32K以上のRAMの搭載が当たり前になるにつれてこのような言語を作っていたマニアたちもしだいに姿を消していき，TINY-BASICなどというものがこの世に存在していたことさえ忘れられてしまいました。メモリも安価になり，本格的な言語が動くのが当たり前になった現在，なにも時代の流れに逆らって整数型の言語を作成する意味があるのか疑問視する向きもあると思いますが，果たしてそうでしょうか。私個人の考えかもしれませんが，少なくともテープベースのクリーンコンピュータでのTINY言語にはTINYなりの良さがあると思うのです。

BASICでは数値の演算は通常，実数演算を行いますので，整数のみの演算に使う場合には無駄が多くなります。実数演算は学術計算には絶対に必要なものですが，ゲームやパズルを解く場合などでは使用頻度は非常に低いものでしょう。また，自分で作成したハードウェアのテストなどを行う場合にはメモリやI/Oの操作能力が問題となりますが，多くのBASICではPEEK関数，INP関数，POKE命令，OUT命令によって1バイトの受け渡しが許されているのみで，非常に貧弱なものです。また，Z80のようにメインメモリの空間の狭いCPUを使用する場合にはメモリ効率も問題となり，無用な実数演算ルーチンのためにメモリをくわれるのはあまり望ましいものではありません。

ここで気を付けて欲しいのは，このようなことから「だからBASICは駄目だ」と判断しないようにということです。BASICとTTLでFOR～NEXTループのようなものを組み，「TTLのほうが2倍ぐらい速い」と単純に考えるのは大きな間違いであるということです。たとえば，「そのループの中で，浮動小数点演算を行ってみなさい」と言われたらどうですか。BASICではあっさり追加できますが，TTLでは考え込んでしまいます。プログラムの完成までの時間はまさに桁違いです。同じことが機械語についてもいえます。I/Oやメモリの操作を機械語で置き換えるとBASICとは比べものにならないくらいの動き方をする上，マイコンマニアが触れるプログラムのほとんどがリアルタイムのゲームであるということも手伝って，「機械語は速い」とか「機械語で組むのが上級者だ」，「機械語のプログラムは高級だ」といった間違いをしがちなのでしょう。少なくともこの記事をお読みの皆さんはこのような間違いがないようにしてください。

2．入力について

インタプリタのサイズはワークエリアを含めて8Kバイトとなっていますがその3分の2程度が演算用の第1スタックとサブルーチンコールやループ制御用の第2スタック用のエリアとして使われていますのでインタプリタ本体は約3Kバイトです。したがって，GOSUBやREPEATループのネスティングレベルをそれほど深くしないのであれば，4Kバイトでも動作は可能です。

私が最初に作成したものは8Kバージョンですが，アセンブラのORGを書き換えて$C000_H$からアセンブルし直せば，4Kバージョンにすることもできます（この他のアドレスに配置するのはソースリストを入力した後，ORGや，ワークエリアなどを設定し直してから再アセンブルすることで可能です）。

入力はモニタから，ダンプリストどおりに$B000_H$～$BBFF_H$まで打ち込みます。終わったら，スタートアドレス$B000_H$，エンドアドレス$BBFF_H$，ジャンプアドレス$B002_H$として，テープにセーブしておきましょう。TTLのコールドスタートは$B002_H$番地，ホットスタートは$B000_H$番地になっています。いずれの場合も，テキストエリアのクリアは行いませんので，プログラムを入力する前には%＝0（BASICのNEWに相当する命令）を実行しておく必要があります。

セーブが終わったら，試しに動かしてみましょう。8Kバージョンでは$B002_H$にジャンプさせます。画面がクリアされて画面の一番上から

```
** TTL VERSION 1.1(C)MIKA **
*READY
■
```

のようにタイトルが出力されて，入力待ちになります。これがTTLがコールドスタートした状態です。続いて%＝0によってNEWを行ってからテキストの入力を行います。

```
1000 "DUMP FROM ? " ADRS=?
1010 ??=A ":"
1020 I=0 ,=8
1030  ?$=<A:I) " "
1040 +I @=I
1050 / ;=!=0 A=A+8 #=1010
1060 #=-1
```

入力が終わったら0 CR によっていったんリストをとってみます。今，入力したテキストがそのまま出力されます。間違いがなければ走らせてみましょう。実行は#＝1で行います。

"DUMP FROM?"と聞いてきますので，**$B000**を入力しますと，TTLインタプリタの機械語リストが出力され，何かキーを押すと終了します。きちんと出力されていれば，正常に動いたと考えて良いでしょう。あとでいくつかサンプルを載せますので動かしてみてください。

リファレンス

1．TTLの特徴

TTLはBASICよりもメモリやI/Oの操作が強力であり，フリーエリアも8Kバージョンでも40Kバイトと非常に広くなっているために，これまで機械語に頼っていたような仕事をTTLで行わせることもある程度可能となっています。特にメモリ操作においては操作が2バイト単位で行えるために，BASICに比べて非常に簡潔な表現ができます。ひとつ例を挙げましょう。次のプログラムはV，およびV＋1番地に書かれているデータを2バイトの符号なしのデータと見なして取り込み，200を加えたあと元のアドレスに戻すものです。

```
BASIC:
 1000 INPUT "ADDRESS=";V
 1010 A=PEEK(V+1)*256+PEEK(V)
 1020 A=A+200
 1030 POKE V+1,INT(A/256)
 1040 POKE V,A MOD 256

TTL (1):
 1000 "ADDRESS="V=?
 1010 A=<V:1)*256+<V:0)
 1020 A=A+200
 1030 <V:1)=A/256
 1040 <V:0)=¥

TTL (2):
 1000 "ADDRESS="V=?
 1010 <V(0)=<V(0)+200
```

TTL(1)はBASICに忠実に書いてみたもので，TTL(2)はTTLの2バイト型メモリアクセス機能をフルに使用してみたものです。最初のBASICで書いたものに比べ，非常に簡潔になっているのがわかります。これはかなり極端な例ですが，いずれにしても書き込みに2バイト型が使用できることはかなりの効果があると思います。

TTLの命令セットはHuBASICはおろか，S-BASICと比べても貧弱なものではありますが，用途によってはこのようにTTLがBASICを圧倒する場合もあるのです（逆に多次元配列や文字列処理などではTTLがBASICに圧倒されてしまいます)。TTLは機械語に非常に近いBASICとも言えるでしょう。このため，TTLを使ってシステムプログラムを書くことも可能です。実際，TTLのアルゴリズムはベースとなったGAMEインタプリタを使って開発したのですから，言ってみればTTLはTTL自身によって開発されたということになるでしょう。

2. サンプルプログラム

TTLを使ってサンプルプログラムを作ってみました。TTLによるプログラミングの参考にしてください。なお，プログラム中↑となっているものは，MZ-80B/2000/2200，X1ユーザーの方は＾と書き換えてください。

(1) Oh！MZ式チェックサム

```
 900 ,=1 @=#! "PRINTER ON ? (Y/N)"/
1000 .=0 ,=1 N=! @=N ;=N="Y" .=1
1010 "START ADRS = " S=?
1020 " END  ADRS = " E=?
1030 "** OH! MZ CHECK SUM SYSTEM V1.2 **"/
1040 "ADR. +0 +1 +2 +3 +4 +5 +6 +7 :SUM."/
1050 A=0 B=0 K=0 ,=1
1060   PAGESUM=0
1070   J=0 ,=8
1080     ??=S " " LINESUM=0
1090     I=0 ,=8
1100       DATA=<S:I) ?$=D " " L=L+D
1110     +I @=I
1120     ":" ?(1)=$L.$F "." ?$=L /
1130     P=P+L S=S+8
1140   +J @=J
1150   "--------------------------------"
1160   ??=P / ;=A>(A+P) +B
1170   A=A+P
1180 ;=! ,=1 @=#!
1190 @=(S>E)
1200 ??=B ??=A /
1210 .=0
```

(2) 再帰的関数表現

TTLの特徴のひとつである，再帰的な関数の表現のサンプルを作ってみました。再帰を使うと，漸化式をそのまま使用することができるようになります。サンプルは，ある自然数Nを与えたときに1からNまでの自然数の和を返すものです。たとえばNが2ならば3，Nが5ならば15を返してきます。これを漸化式を使って書くと次のようになります。

$$y = f(x) = \begin{cases} 1 & \cdots\cdots \text{if} \quad x = 1 \\ f(x-1) + x & \cdots\cdots \text{if} \quad x > 1 \end{cases}$$

これをTTLで記述すると次のようになります。

```
1000 "A=" A=?
1010 :=2000,A
1020 ?=Z
1030 #=-1
1999-----------------------------
2000 ;=A=1 Z=1 ↑
2010 :=2000,A-1 Z=A+Z ↑
```

2000行が漸化式で言うと f(x)＝1の部分に相当し，次の2010行が f(x－1)＋x に相当します。TTLではPASCAL のように関数が値を返す機能がないために変数Zを使って値の受け渡しを行っています。つまり，2000行を：=2000，A という形で呼び出すと1からAまでの自然数の和を変数Zに代入してくるわけです。

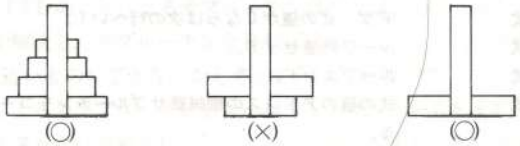
(3) ハノイの塔

ハノイの塔というのはかなり古典的なパズルのひとつですが知らない方もいらっしゃると思いますので，簡単に説明しておきましょう。

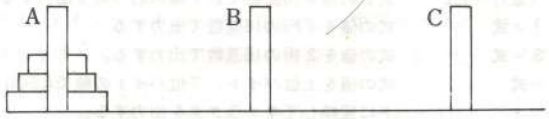
A，B，Cと3つの柱があり，柱Aに大きい円板から順次小さい円板が重なっています。この円板を次の規則に従って円板全部をAからCに動かすのを目的とします。

①円板は1度には1枚しか動かせません。

②大きい円板を小さい円板の上に重ねることは許されません。



例として円板が3つの場合の回答を示しましょう。



例ではAに3枚のプレートが重ねられています。これをCに移すにはA→C　A→B　C→B　A→C　B→A　B→C　A→Cとなります。わかりにくい人は紙を切って重ねて動かしてみるとよいでしょう。

```
1000------ TOWER OF HANOI ------
1010 "HOW MANY PLATES ? " N=?
1020 :=2000,1,2,3,N
1030 /"FINISH!"/
1040 #=-1
1999-----------------------------
2000 ;=D<2 #=2100
2010 :=2000,A,C,B,D-1
2020 $=$40+A "->" $=$40+C " "
2030 :=2000,B,A,C,D-1
2040 ↑
2100 $=$40+A "->" $=$40+C " "
2110 ↑
```

TTLではローカル変数を6つまで持つことが可能ですから，パラメータを6つ持ち，多数のデータを返すような関数まで記述することが可能となっています。ここまで来ると言語を作った私自身にも，どのように使ったものか，考え込んでしまいます。みなさんのアイデアで使いこなしてください。

3. 文法について

TTLの設計にあたり，GAMEとの互換性をある程度意識していましたので，GAMEをご存じの方にはすぐにのみこめると思いますが，さすがに二十数回にわたってバージョン変更を行っていると，自分自身にも記憶しかねる部分が出て来てしまったので，リファレンスマニュアルを作ってみました。もともと個人用に使うつもりだったのですが，考えてみると，TTLを使うのは私だけではなくなるのですから，やはり公開することにしました。一般に公開するということで，かなり書きたしてはみたのですが，もとが自分用に作ったものですからややわかりにくい個所があるかもしれません。各コマンドの詳細については次の章の解説を参照してください。

ここで**項**というのは定数，変数，およびカッコでくくられた式のことを指し，**式**というのは少なくとも1つ以上の項からなり，項が2つ以上の場合には項と項の間に演算子が入ります。

(項) A，1234，$ABCD，!，"ABC"，(123＋4＊6)，(2＊(3＋2))，etc.

(式) A，12345，A＋B，12＋(34＊3)，!＊2，etc.

〈コマンド一覧〉

＃＝式	式で指示される行番号へジャンプする。式の値が32768以上であればプログラムの実行を終了する（BASICのENDと同じ）。
.	
!＝式	式で指示される行番号のサブルーチンをコールする。
]	サブルーチンからのリターン。
:＝式	値の引渡しと変数の退避を行うサブルーチンコール。
↑(^)	:＝に対応するリターン。
;＝式	IF文。式の値が0ならば次の行へいく。
,＝式	ループ終値セット。
@＝式	ループエンド。
＞＝式	式の値のアドレスの機械語サブルーチンをコールする。
?＝式	式の値を10進数の5桁の右づめで出力する。
?(式1)＝式2	式2の値を10進数で式1幅の右づめで出力する。
??＝式	式の値を4桁の16進数で出力する。
?$＝式	式の値を2桁の16進数で出力する。
$＝式	式の値を上位バイト、下位バイトの順でASCIIコードに変換してキャラクタを出力する。
"文字列"	文字列を出力する。
／	改行する。
'数字列'	各数字に対応するディスプレイコントロールを順に実行する。
＋ 変数	変数の値を1増やす。
－ 変数	変数の値を1減ずる。
＊ 変数	変数の値の上位バイトと下位バイトを交換する。
10進定数 ／	定数で指示される行番号の行からプログラムリストを出力する。ダイレクトモード時のみ有効（プログラム中では使用不可能）。

〈変数・定数〉

A～Z	単純変数。冗長形が許される。
¥	最後に行った除算の余りを返す。
<項：式)	メモリ変数（1バイト）
<項（式)	メモリ変数（2バイト）
[項：式)	I/O変数（1バイト）
[項（式)	I/O変数（2バイト）
"文字列"	文字列の右から2文字をASCIIコードと見てデータにする（ex. "ABC"→4243H）。
!	キーボードを見て押されていなければ0，押されていれば該当するキーのASCIIコードを返す。
?	キーボードから式を1行入力し，その値をとる。
π	カセットからファイルを読む時の先頭アドレスを格納する（ただしπが0ならばセーブ時と同じアドレスから読み込まれる）。
. （ピリオド）	出力制御用変数。プリンタの出力のON-OFF（第0ビットが1ならON）や画面コントロールコードの有効，無効（第1ビットが1なら無効），および画面出力のON-OFF（第2ビットが1ならOFF）を決定する。
&	テキスト開始番地を格納する。
%	テキストの終了番地を格納する。プログラムの実行が終了し，"＊READY"が出力された時点で自動的に文末探査が行われ，変数%の値も変更される。ただし，%＝0はBASICのNEWに相当する。

※10進定数は通常使用するとおり（ex. 1234）。取り得る値は0から65535までの整数。負の数は補数表現であり，－1と65535が等しくなるため，たとえば－1＞2は真となる。

※16進定数は最初の文字を$にする（ex. $12AF）。取り得る値は$0000から$FFFFである。

〈二項演算子〉

＋，－，＊	加算，減算，乗算
／	除算(余りは自動的に変数¥に格納される)
. （ピリオド）	論理積（AND）
;	論理和（OR）
!	排他的論理和（XOR）
＞，＜，＝，＃	大小比較（＃……等しくない）

〈単項演算子〉

＃	項の値が0ならば1，それ以外の場合は0をかえす。
／	項の値の行番号の行の先頭のアドレスをあたえる。
－	0から項の値を引いた値にする。
＊	項の値の上位バイトと下位バイトを交換する。

※行番号の直後にスペースをあけない場合，その行は注釈文と見なされ，読み飛ばされます。

※プログラムの入力方法は基本的にはBASICとほとんど変わりありません。RENUMやDELETEのようなコマンドもありませんので，やや不自由に感じることがあるかも知れませんが何せTINY言語ですので我慢してください。

※プログラムのSAVEルーチンはBB00Hから，LOADルーチンはBB70Hから設けてあります。＞＝$BB00や＞＝$BB70のようにしてCALLするとSAVE，LOADを行うことができます。

マニュアル

次にTTLの文法をコマンド1つひとつについて説明していきましょう。なお，｛　｝はそれが省略可能であることを示しています。

1. コマンド

(1)＃＝式

式で指定される行番号にジャンプします。BASICのGOTO文と同様に考えることができます。飛び先の指定に式が許されていますので，たとえば＃＝Aのようなことも許可されますが，プログラムが見ずらくなりますから，あまり使用しないほうがよいでしょう。該当する行番号がない場合には式の値を越える最小の行番号にジャンプします。

```
1000 #=2000
1999 "L1999"
2001 "L2001"
```

1000行で2000行へのジャンプを指定していますが，プログラム中には2000行は存在しません。このプログラムを実行すると"L2001"と表示されます。BASICで同様のことを行うと"UNDEFINEND LINE NUMBER"などのエラーが出ます。TTLを作るとき，エラーを出すか否か迷ったのですが，0で除算を行ってしまったといったような場合と異なり，プログラムの実行が不可能になるようなエラーではない場合が多いので，エラーは出さないようにしました。

プログラム中で使用されている最大の行番号よりも大きい値を指定すすと，プログラムの実行は打ち切られ，最後に実行した行を表示してコマンド待ちに戻ります。行番号として許されるのは1から$7FFFまで，つまり32767までですから，それ以上の値を指定すると必ず実行が終了することになります。TTLで扱う数値は基本的には符号なしの2バイトの整数ですからたとえば－1は65535と同じになります。したがって，#＝－1とするとBASICのENDと同等になります。

(2)！＝式1｛：式2｝および］

！＝は式1で表される行番号から始まるサブルーチンをコールし，サブルーチンからは］によって元に戻ります。BASICのGOSUB～RETURN文と同様です。該当する行がないときは#＝のときと同様，式の値を越える最小の行をコールします。式の後にコロン(：)をつけて式2を書くとその値から始まるテキスト中のサブルーチンをコールします。TTLではメモリの許す限り，複数のテキストをメモリ上に置くことが可能です。TTLをコールドスタートさせた直後にはテキストは$7000からとなっていますが，後述する&＝によって変更することが可能です。このことを頭にいれて，次のようなプログラムを実行してみましょう。まず，**&＝$A000 %＝0**（CRをお忘れなく）とやって，$A000番地からテキストを格納するようにしてから次のプログラム(サブルーチン)を入力します。

```
1000--- SUB ROUTINE ---
2000 "THERE IS $A000 "
2010 ]
```

次に**&＝7000 %＝0**とやって，$7000番地から次のプログラム（メインルーチン）を入力します。

```
1000 --- MAIN ROUTINE ---
1010 !=2000:$A000
1020 #=-1
2000 "THERE IS $7000"
2010 ]
```

入力が終了したら#＝1によって実行を開始させます。"THERE IS $A000"と表示されたでしょう。$A000から入力したプログラムの2000行からのサブルーチンがコールされたわけです。次に1010行の：**$A000**を削除して再び実行してみます。今度は"THERE IS $7000"が出力されたでしょう。これはBASICのGOSUB文と同じものとみなせるでしょう。TTLにはBASICのMERGEに相当するようなコマンドは存在しませんが，この例のようにプログラムを分割して互いに相手を呼び出すことが可能なため，プログラムを各モジュールごとに開発していって最後にそれをまとめあげていくといったプログラム作成を行うことで，それほどの不自由はないと思うのですがいかがでしょうか。

(3)：＝式｛，式，……｝｛：式｝および↑

（↑はMZ-80B/2200/2000, X1では＾。以下同様に読み替えてください。）

BASICにはTTLの：＝に相当する命令はありません。あえて類似を探せば，！＝を拡張したものということが可能かもしれません。

BASICを使用していて困ることのひとつにサブルーチン内でのみ有効なローカル変数がないということがあげられると思います。次のようなサブルーチンを組んだとしましょう。

```
1000 REM--- CALC SUM(A)---
1010 S=0
1020 IF A<0 THEN 1060
1030 FOR B=1 TO A
1040  S=S+B
1050 NEXT B
1060 RETURN
```

このサブルーチンは1からAまでの自然数の和を求めるものです。このサブルーチンが正常に動作することは疑いがないところでしょう。ところが，このサブルーチンを次のように使うと問題が発生します。

```
100 REM--- MAIN ROUTINE ---
110 FOR B=1 TO 10
120  FOR C=0 TO 9
130   A=B*10+C
140   GOSUB 1000
150   PRINT A:S
160  NEXT C
170  PRINT
180 NEXT B
```

この例のようにメインルーチンでサブルーチンと同じ変数を使用していると，アルゴリズムとしては正しいにもかかわらず，プログラムとしては正常に動作しないという事態が発生します。ここであげた例はごく短いプログラムなのでかなりわざとらしい感じがするかもしれませんが，少し長いプログラムを組んだことがある方でしたら経験のあるところでしょう。TTLの：＝によるサブルーチンコールはこのようなトラブルをなんとか解決し，サブルーチンをPASCALで言うところのPROCEDUREに近いものにできないかと考えて作った命令です。

：＝によってサブルーチンをコールするとき，AからFまでの6つの変数はスタックに退避され，↑によってリターンするときに復帰されます。したがって，AからFまでの変数はサブルーチン内で値が変化してしまっても，メインルーチンに戻ったときに，サブルーチンを呼び出す前の値に戻りますから，お互いに変数の重複を気にすることなくプログラムを組むことができます。

さらに，：＝式の後に，式の形を続けて書いておくと(6つまでです)，変数の退避を行ったあと，式の値が変数AからFまで順に代入されていきます。次のプログラムを実行させるとよくわかるでしょう。

```
1000 A=1 B=2 C=3 Z=7
1010 :=2000,A+B "MAIN: "
1020 " A=" ?(1)=A " B=" ?(1)=B
1030 " C=" ?(1)=C " Z=" ?(1)=Z
1040 // #=-1
1999-------------------------
2000 B=5 Z=0 " SUB: "
2010 " A=" ?(1)=A " B=" ?(1)=B
2020 " C=" ?(1)=C " Z=" ?(1)=Z
2030 / ↑
```

少し込み入っていますが，順に見ていって下さい。1010行でサブルーチンコールを行い，AにA＋B(＝3)を代入しています。サブルーチン内ではBを5に，Zを0にしています。ここのPRINT文の結果からもわかるようにAは3(：＝で引き渡された値)になり，BとZは2000行で設定された値，Cはメインの最初で設定した値そのままになっています。↑によってメインに戻った時点ではAからCまでの変数は1010行の：＝を行う前と同じ値になっており，ZはBASICでGOSUBを行ったときと同様にサブルーチン内で変更されたままの値を持って来ていることが

わかります。

AからFまでの変数はこのように特殊な変数であり，サブルーチンの中のループカウンタなどにはこれらの変数のみを使うようにすることで，プログラムをブロックごとに開発して，動作が確認されたものについてはその内部で使用されている変数名などに気を使うことなく，ブラックボックス的に扱うことが可能になります。

さらに変数の退避や復帰をスタックで行い，ローカル変数への値の受け渡しが行えるために，簡単な再帰を使ったようなプログラムもTTLで扱うことができます(具体的な例はサンプルプログラムを見てください)。

この命令の場合にも！＝と同様に，：に続いて式を書き加えることで，他のアドレスから始まるテキスト中のサブルーチンを呼び出すことが可能です。

(4)；＝式

式の値が0ならば次の行へ，それ以外の場合は次の文へいきます。BASICのIF文と同じものと考えられます。

(5)，＝式，＠＝式

ループ制御です。，＝はループのリミットを設定するものです。ループの終わりは＠＝です。もし，＠＝の後の式の値が，，＝で設定した値以上になればループは終了し，それ未満であれば再びループを繰り返します。，＝1としておいて，＠＝**条件式**とすれば，BASICのREPEAT～UNTIL文と同じことになります。

```
1000---- SAMPLE OF LOOP ----
1010 I=0 ,=100
1020  ?(4)=I
1030 +I @=I
```

(6)＞＝式

式の値で示されるアドレスから始まる機械語のサブルーチンをコールします。BASICのCALL文に相当します。

(7)？＝式

式の値を10進数で5桁幅で右づめで出力します。

(8)？(式1)＝式2

式2の値を10進数で式1の桁幅の右づめで出力します。

(9)？＄＝式

式の値を16進数に変換し，下位バイト(16進数2桁)を出力します。

(10)？？＝式

式の値を16進数の4桁で出力します。

(11)＄＝式

式の値を16進数に変換し，上位バイト，下位バイトをそれぞれASCIIコードとみなし，その順序に出力します。たとえばAが＄4142のときに**？＄＝A**を実行すると"AB"と出力されます。

(12)"文字列"

文字列をそのまま出力します。BASICのPRINT"文字列"；と同じです。文字列出力後の改行は行われませんので，改行が必要な場合は次に出てくる／による改行を付加します。

(13)／

改行する。後で説明する，．＝によって画面コントロールを禁止しても，この命令による改行は実行されます。改行の禁止が可能になるようにしたい場合には，**＄＝＄D**によって改行を行ってください。

(14)'数字列'

各数字に対応する画面コントロールを順に実行します。数字は続けて記述できます。各機種のコントロールコードが異なりますので，よく使うコントロールを1～6の数字に割り当てました。移植を考えると，**＄＝式**や**"コントロールキャラクタ"**よりも**'数字列'**を使ったほうがよいと思われます。コードとコントロールの対応は以下のとおりです。

1……カーソルを下へ移動
2……カーソルを上へ移動
3……カーソルを右へ移動
4……カーソルを左へ移動
5……カーソルをホームポジションへ移動
6……画面クリア

たとえば，**'633311'**を実行すると，画面クリアしたあと，カーソルをホームポジションから右へ3つ，下へ2つ移動させます。

(15)＋変数

変数の値を1増やす。オーバーフローのチェックは行っていませんので，変数が＄FFFF(＝65535。ただし，変数が1バイト型の場合は＄FF＝255)であるときに＋**変数**を行うと変数の値は0になります。

(16)－変数

変数の値を1減ずる。＋**変数**の場合と同様に，変数の値が0のときにこの命令を実行すると変数の値は＄FFFF(変数が1バイト型の場合には＄FF＝255)になります。

(17)＊変数

変数の値を16進数とみなし，上位バイトと下位バイトを交換します。たとえばAが＄0102(＝256＋2＝258)であるときに＊Aを実行するとAの値は＄0201(＝256＊2＋1＝513)になります。1バイト型の変数(メモリやI/Oの1バイト型の指定型)の場合には上位4ビットと下位4ビットが交換されます。

(18)その他

※NEWは**%**＝0，LISTは0 CR です。ある行からのLISTを出力したいときは**行番号**／によって行うことができます。この場合の行番号指定に使えるのは10進定数のみで，16進数や変数，式などを使うことは許されていません。

※プログラムの実行は＃＝1によって行います。結局はGOTO文と同じですから変数のクリアは行われません。

2．変数，定数

(1)<項：式)，<項(式)(メモリ変数)

TTLではメモリと普通の変数(AやBなど)とを区別せずに使用することができます。項と書いてあるところは，変数，定数，または(＄D000+40)のようにカッコでくくられた式のいずれかがきます。

メモリ変数には1バイト型と2バイト型があり，項と式の間が：で区切られているのが1バイト型，(で区切られているのが2バイト型です。1バイト型の場合にはメモリの(項+式)の値のアドレスが，2バイト型の場合には(項＋2＊式)を下位バイト(項＋2＊式＋1)を上位バイトとしてアクセスされます。メモリ変数の内容を読み出すことはBASICのPEEKに相当し，メモリ変数に代入することはBASICのPOKEに相当します。たとえば，**<＄D000：B)＝A**はBASICでいうとPOKE ＄D000＋B，Aに相当し，**A＝<＄D000(B)**はBASICで書くとA＝PEEK(＄D000＋2＊B)＋256＊PEEK(＄D000＋2＊B＋1)になります。

※2バイト型のメモリへのアクセスはアドレスの小さい方(下位バイト)，大きい方(上位バイト)の順序で行っています。タイミング的にまずい場合には1バイト型を2回使用して書き込むようにしてください。

(2)[項：式)，[項(式)(I/O変数)

TTLではメモリと同時にI/Oを扱うことができ，メモリと同様に1バイト型と2バイト型をもちます。I/O操作はTTL内部ではCレジスタ間接モードで行っていますので，X1シリーズのようにI/O空間にV-RAMを割り付けたようなマシンの場合にも機械語のお世話にならずにI/O空間をアクセスすることが可能です。2バイト型の場合の読み書

きの順序はアドレスの小さい方（下位バイト），大きい方（上位バイト）の順序で行っています。

A＝[B：0）はBASICのA＝INP（B），[B(1)＝AはBASICのOUT B＋2，A MOD256：OUT B＋3，INT(A/256)に相当します。

(3)&と%　（テキスト開始番地，終了番地）

TTLではテキスト（プログラム）はメモリ上のどこにあってもかまいませんので，テキストの開始番地は任意に決定することができます。テキストの開始番地を格納するのが変数&で，テキストの終わりを格納しているのが変数%です。？？＝&や，？？＝%とすることで現在のテキストの開始番地やテキストの最後がどこになっているかを知ることができます。また，&＝式としてテキストの開始番地を変更しながら，テキストを書くことで複数のプログラムを同時にメモリ上に存在させることができますから，BASICのようにいちいちカセットにセーブしてからNEWを行って再び入力するといった手間をかける必要もほとんどなくなります。

&と%はこのような用途のほか，セーブのときのセーブエリアの指定にも使用されます。＞＝$BB00によってセーブルーチンをコールすると，&から%までのエリアがセーブされます。TTLでは"＊READY"が出力されるごとに自動的にプログラムエンド探査が行われ，%の値が更新されていますので，メモリの任意のエリアをセーブしたい場合には&＝先頭アドレス　%＝エンドアドレス　＞＝$BB00と1行にズラズラとかいて一度に実行することで行います。

例外としては%＝0があります。%＝0はBASICのNEWに相当するものです。もっともNEWとは言っても，やっていることはテキストの先頭にFF_H，00_Hを書き込み%ポインタを&ポインタと等しくするだけですから，たとえばA＝先頭行番号　＊A　＜&（0）＝Aとでもやれば簡単に復活することができますので，%＝0をうっかり行ってしまったと言って嘆く必要はありません。

(4)π　（ロードアドレス）

TTLでは一度作ったプログラムはメモリ上のどこに配置しても動作します。一度作ったプログラムをアドレスが合わないというだけの理由で再びキーインし直すというのは非常に馬鹿らしいことです。TTLでは一度セーブしておいたプログラムをロードするときに任意のアドレスから読み込むことができるようにしてあります。変数πがカセットからロードするときの先頭アドレスを指定するものです。例外として，πが0のときにはセーブしたときと同じアドレスにロードされますから，通常はπ＝0を実行してからロードするようにしましょう（ロードは＞＝$BB70によって行います）。

(5)！

BASICで言うとINKEY$でしょうか。押されているキーがあればそのキーのASCIIコードを，押されているキーがなければ0を返します。2つ以上のキーが押されていても1つしか判別できません。また，画面へのエコーバックも行われません。

(6)？

キーボードから1行分，式を入力し，その値を返します。文字列に関してはダブルクォーテーションで囲むことで2文字（2バイト分）までは入力することができます。これ以上の文字列の入力が必要な場合には1行入力バッファ（LNBUF 1）を利用してください。

(7)"文字列"

文字列の右から2文字をASCIIコードとみなし，左側の文字を上位バイト，右側を下位バイトとして値を返します。ちょうど$＝式の逆を行うと思えば良いでしょう。もし，文字列が1文字しかないときは上位バイトは0に設定され，文字のASCIIコードが下位バイトに設定されます。また，文字列がヌルストリングのときは下位バイトも0になります。

3．二項演算子

二項演算子は2×3のように次に書かれている項との間で行う演算を指定するものです。これに対して次の項の値のみに対して演算を行うのは単項演算子と言い，BASICでいうとLOG（X）とかINT（X），RND（X）などがそれにあたります。

(1)四則演算（＋，－，＊，／）

それぞれ，加算，減算，乗算，除算です。いずれの場合にもオーバーフロー（0から65535の間にあるか否か）のチェックは行っていませんので，たとえば？＝$100＊$100をやると0に，　？＝300＊400とすると54464といった具合に妙なことになります。

除算を行ったときの余りは変数¥に代入されます。

(2)論理演算（．；！）

それぞれ，AND，OR，XORです。TTLでは16ビット単位で処理される点が違うぐらいで機械語の論理演算と似たようなものです。ここで論理演算についてよくご存じでない方のために少し説明を加えておきましょう。たとえばA＝$4E，B＝$7Cとします。

①論理演算は2進数で行われます。

A＝$4E＝01001110

B＝$7C＝01111100

②両者の同じ位置にあるビットを比べ，演算を行います。

ANDは，両者のビットが共に1の場合にのみ1，その他の場合には0にします。ビット単位での掛算のようなものということで，論理積とも言われます。AとBのANDをとると，01001100＝$4Cとなります。

ORは，少なくともどちらか一方のビットが1であれば1になり，どちらも0なら0になります。論理和ともいわれます。AとBのORをとると，0111110＝$7Eとなります。

XORは両者のビットが同じであれば0に，違っていれば1になります。一般に，排他的論理和と呼ばれています。AとBのXORをとると，00110010＝$32となります。

(3)比較（＜，＞，＝，＃）

大小関係を判別するもので，関係式が成立すれば（真ならば）1，成立しなければ（偽ならば）0となります。＃は両者が等しくないという条件です。たとえば，1＝2の値は0，1＃2は1になります。比較は2バイトの符号なしの場合の値で行われますので，－1は65535とみなされます。したがって，－1＞2は真となってしまいますから注意してください。

4．単項演算子

単項演算子は次にくる項に対して演算を行うもので，二項演算よりも優先的に処理されます。

(1)＃

どちらかというと，算術演算よりも論理演算用の単項演算子で，意味としてはNOTのつもりで作ってみました。次の項の値が0ならば1，それ以外ならば0になります。＃に続けて論理演算式をカッコでくくって記述すると，論理演算式の否定がえられます。；＝（BASICのIF文に相当）文で使用することが多くなるでしょう。

```
;=#(A=B) "NOT EQUAL!"
```

（AがBと等しくなければ"NOT EQUAL！"と出力する。）

(2)－

TTLは基本的には負の数は持っていませんが，これではV-RAMのアクセスのときなどにやや不便を感じるので，単項演算子として用意することで不自然なく利用することができるようにしました。10進の定数が

きたときの形は－1といったふうになるので一見演算子でないようにみえますが，TTLの内部では演算子として扱われています。－はその前に0があるのと同じ値を返します（たとえば－1は65535，－2は65534といった具合になります）。

(3)／

次の項で与えられる値の行番号を探し出してそのアドレスを返します。該当する行番号がないときには項の値を越える最小の行番号のあるアドレスになります（＃＝などの飛び先と同じことです）。返ってきたアドレスと次のアドレスに格納されている2バイトの値が行番号になっています。

```
1004 A=0 B=1 C=3
1010 W=/1000 Z=#<W(0) ?=Z
```

（＃＝1000を行ったときの飛び先，この場合には1000行がないので1004行を見つけ，その行番号を表示します。）

(4)＊

項の値を16進数4桁に直し，その上位2桁と下位2桁を交換した値にします。コマンドの＊と同じようなことを行う演算子ですがコマンドが単体で用いられるのに対し，演算子の＊は必ず式の中で使用される点が異なります。

```
1000 A=$1234
1010 ??=A / *A ??=A / ?=*A //
1020 B=5**A ?=B
```

（二項演算子の＊とは式の中で置かれる位置が異なるので判別できます。）

おわりに

移植について

TTLに限らず，ほとんどのシステムソフトウェアは，使用するCPUさえ同じならば入出力ルーチンを書き換えるのみで動作することはCP/Mなどを使用している方ならばよくわかることでしょう。CP/M上で動作するプログラムは機種に依存せず，たとえばPCで動いているPASCALをそのままMZのメモリ上にもってくると（ディスクのフォーマットを変えたりするだけで），そのまま何の手直しも行わずに動かすことができます。世の中にはパソコンと称するものが数多く出回り，それぞれがそれなりの特徴を持っています。このことは，ユーザー側には目的に合う物を選択できることにもなり，好ましいことではありますが，反面，あまり機種の特性に依存しないプログラム，たとえば，アセンブラなどのシステムソフトなどでは非常にムダな労力を払うことにもなります。

CPUが同じで，同じ機械語のプログラムが同じアドレスにあれば，同じことをやるのは当然ですが，外から見ているユーザー（オペレータ）にとっては同じことにはなりません。たとえばMZ-700でD000番地に1を書くと画面の左上にAと表示されますが，PCで同じことを行ってみると，そのようなことはまったく起こりません。CPUにとっては同じ「D000番地に1を書く」ことであっても，システム全体では同じことにはならないことになります。

このようにどうしても機種に依存してしまうのはその機種独自の入出力部だけです。逆にいえば，機種に依存している入出力を書き換えれば，他はまったくいじらずに移植が完了することになります。CP/Mではこのような入出力関係のサブルーチンのエントリポイント（入口）や入力条件（パラメータの与え方）などが決められており，それを使うようにする限り，まったく機種に依存しないプログラムになります。

TTLを作るときにこのようなことも頭の片隅においていました。MZの場合，CP/Mなどのようなグローバルなオペレーティングシステムは標準で付いてこないのですが，幸いクリーン思想を正しく堅持してきたシャープさんがMZ-80KのP-ROMタイプ（試作段階）以来，最新のMZ-1500にいたるまでROMの主要サブルーチンのエントリポイントやワークエリアをまったく変更することなく，またカセットのフォーマットなどについても共通のまま，バージョンアップをしてきてくれたおかげで，ROMがMZ-80K/C/K2/K2E，1200，711/721/731，1500のすべてを標準化するキーになっています。ROM内のサブルーチンを使う限り，シャープの1号機でも最新の1500でもまったく同じプログラムが使えるということは驚異でもあります。某社のようにコンパチビリティをうたってこそいませんが，MZは統一された設計思想に基づいて作られた素晴らしいシリーズであるといえます。

話をTTLに戻しましょう。TTLはMZ-700（MZ-731）で作成したことはすでに述べましたが，このとき，700の特徴であったファンクションキーやカラーコントロールは無視し，MZ-80Kシリーズと共通なROM内ルーチンのみ使用しました。このため，TTLは一度カセットを作っておけば，MZ-80KからMZ-1500に至るまでのMZ-80K系統の機種すべてでロードし，実行することができます。もちろん，コマンドや動作にもまったく変化はありません。機種別の特徴は多少そこなわれることはいなめませんが，機種に依存しないことのメリットを考えれば当然ともいえるでしょう。

TTLを他機種に移植する場合は少し手間がかかります。幸い，編集室にX1とMZ-80B，2000に詳しい方がおられますので，MZ-80B，2000，およびX1への移植は完了していますが，その他の機種や自作の機械に移植したい方のために簡単に移植の方法を書いておきましょう。まず，TTLのソースリストをすべて入力し，入出力部の書き換えを行い，さらに必要があれば，ORGやワークエリアを変更します。書き換えが終了したら，アセンブルしなおしてオブジェクトプログラムを得ることになります。

ソースを入力するのが嫌な場合にはパッチをあてるという方法もあります。0番地からがRAMになるモードにして，MZ-700のROMと同じエントリポイントから同じことを行うプログラムを組み込み（ワークエリアもそろえておきます），TTLのオブジェクトはそのまま入力して使います。

どちらの方法でも動作するようになりますが，機械語のプログラムのソースというものは持っていると何かと便利なものですから，MZ-80K系統以外の方はもちろん，MZ-80K系統の機械を使う場合でもできればソースを入力しておくほうが良いと思います。次にTTLで使用したROM内ルーチンとその動作を説明しておきましょう。

ラベル名	アドレス	動作
??CR (AF以外は保存)	0006	改行する。
??NL (AF以外は保存)	0009	同上。ただしカーソル位置が左端なら何もしない。
??PRNT (AF以外は保存)	0012	Accの内容を画面出力。
?ADCN (AF以外は保存)	0BB9	Accのアスキーコードをディスプレイコードに変換する。
?PRNTX (AF以外は保存)	0DB5	Accのディスプレイコードを出力。
?BRK (AF以外は保存)	001E	SHIFT＋BREAKが押されているかを検出する。押されていればZフラグをたてる。
?GETKY (AF以外は保存)	001B	キーボードを読み，押されていればそのASCIIコードを，押されていなければ00をAccに収めてくる。エコーバックは行わない。

?GETL (全レジスタ保存)	0003	キーボードから1行入力を行う。入力した行はDEレジスタで指定されるアドレスから格納される。行の終わりには0D(CRコード)がセットされる。入力文字数は0Dを含めて最大80文字。SHIFT+BREAKが押されたら先頭に0B(BREAKコード)次に0Dがセットされる。これらのコードは機種により異なることがあるので，ソース中では，INPEND，BRKEYというラベル名で定義してある。
DSPXY	1171	モニタのワークエリア。現在のカーソルのX座標が格納されている。INPUT文のときにキーから入力した部分のみをとりだすときと，?NLルーチン(??NLではない)の処理で使用される。

その他，システムに依存するのはプリンタへの出力ルーチンです(LPTと書いてあるところ)。これはソースをみると最後のところにおいてありますので，必要に応じて書き換えてください。仕様としてはAccに収められたASCIIコードをプリンタに出力するだけのものを組めばよいようにしてあります。

カセット関係のルーチンはうしろに付加しただけになっていますので(SAVEとLOADルーチン)，各自のシステムに応じてカセット，フロッピーディスク，QD用に書き換えても良いでしょう。

出力関係で，ASCIIコードを与えるだけで出力するルーチンがあるのに，ディスプレイコードへの変換を行うルーチン，ディスプレイコードを出力するルーチンを使っていることを不思議に思う方もおられるでしょう。ほとんどのキャラクタはどちらを使用しても同じ結果になるのですが，問題はコントロールコードです。MZではカーソルの移動や画面のクリアを行うキャラクタ(矢印やH，Cの反転文字)を持っています。プログラムの実行時にはコントロールコードとして動作して良いのですが，LISTをとるときにカーソルコントロールをやられたのでは使いものになりません。そこで，CALL ?ADCNに続いてCALL ?PRNTXを行うとコントロールキャラクタは反転文字で表示され，画面のコントロールは行われないのを利用してLIST出力などを行うのです。

したがって，テキストの中にコントロールコードが入らないシステムでは?ADCN ?PRNTXは不要で，?PRNTで書き換えて構いません。

SHIFT+BREAKは他の機種では単にBREAK，STOP，ABORTなどとよばれていると思いますが，いずれにせよプログラムの実行を打ち切るキーをセンスするようにしてください。

●ちょっと注意していただきたいこと

ここではTTLと機械語を併用するときに必要な情報を書いておきましょう。

(1)変数(AからZまで)はワークエリアのVARTから2バイトずつ，A，B，C，D……Zの値を格納してあります。アドレスの小さいほうが下位バイトとなっていますので，例えば，LD HL,(VART)を実行すれば，HLレジスタに変数Aの値が転送されます。また，LD(VART+2),HLなどとすれば，TTLの変数(この場合は変数B)に値を引き渡すことができます。

(2)TTLの1行入力バッファは2つあります(LNBUF1のLNBUF2)。ひとつはテキストの入力のためのもので(LNBUF2)，もうひとつはA=?などを行うときの1行入力用のバッファです(LNBUF1)。最初，気付かずに両者を兼用していたら，ダイレクトモードでA=?をやったときにおかしなことになってしまいました(考えてみれば，あたりまえのことで，実行している行を壊してしまったのです)。ダイレクトモードで，1行入力を行うということをしないのであれば，LNBUF1とLNBUF2に同じ値を与えてもかまいません。

(3)TTLのスタックは3本あります(CPUのスタック，およびSTACK1とSTACK2)。TTLがスタートすると，CPUのスタックポインタはD000Hに設定されます。この値はソース中ではCPUSTKで表わされています。TTLでは一部，サブルーチンの中やスタックに値を退避させた状態から強引にエラー処理に飛んだりしていますので，スタックのイニシャライズを行う所は5個所もあります。ダンプを直接書き換える人はソースをよく見てください。

STACK1は演算式で，カッコの処理を行うときに使用されるもので，STACK2はループやサブルーチンコールの際に，ループの終値，返り先の番地や変数の退避などを行うのに使用されます。どちらもリミットチェックが行われており，スタックに積みすぎたり，掘りすぎたりすると，〔?STACK1〕，〔?STACK2〕のエラーメッセージが出力されます。

STACK1はこのほか，1行演算が終了した時点で，もとの値に戻っているか，つまり，(の数と)の数が同じであったかどうかのチェックも行われています。

●メモリマップ

機械語とTTLを併用する場合に便利なようにTTLの簡単なメモリマップを書いておきます。編集室のご好意によって，ソースリストが掲載できるようになったので，細かなサブルーチンの配置などについてはそちらを見てください。

B000	ホットスタート	TTL本体
B002	コールドスタート	
	行実行ルーチン	
B8FF	テキストエディタ	
BA60	入出力ブロック	
BB00	カセット出力ルーチン	
BB70	カセット入力ルーチン	
BBE6	ワークエリア	TTLワーク
BC4B	ラインバッファ1	
BD00 CDF7	第2スタック	
CE00 CEFF	第1スタック	
CF10 CFFF	ラインバッファ2 CPUスタック