Software Support for an Intelligent Terminal, the Beehive B500

Wilm Boerhout
Willem Böhm
Rob Gerth

RUU-CS-79-11

November 1979

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

Software Support for an Intelligent Terminal, the Beehive B500

Wilm Boerhout
Willem Böhm
Rob Gerth

Technical Report RUU-CS-79-11

November 1979

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht, the Netherlands

Index

Software Support for an Intelligent Terminal, the Beehive B500

Abstract

This report documents software on the CDC CYBER and the B500 terminal, that creates an environment for writing, testing and downloading B500 programs.

Keywords and Phrases

Basic software, downloading, intelligent terminal, microprocessor.

Introduction

In the summer of 1978 we got a Beehive B500, an intelligent terminal. The B500 derives its "intelligence" from an INTEL8080 microprocessor. The terminal functions are controlled by a program, residing in ROM, called the Upper Level Terminal program (ULT). Apart from the ULT, "user" programs can be downloaded into the 8080, in this case from a CDC CYBER. In order to use this facility we have created an environment for writing software for the B500 (we think of an editor).

On the CYBER we needed an assembler, an interpreter, a system text mechanism and a linker, to write small pieces of basic software in assembly language. To write bigger programs, such as an editor, we needed a high level language, e.g. PASCAL. On the B500 we needed a monitor program, that performs basic input/output using the ULT, loads and executes programs, disassembles pieces of memory, takes single steps through a program, sends blocks of data to the host, reads blocks of data from the host, and the like. In order to write this monitor, the ULT had to be fully documented.

As far as the CYBER software was concerned, a tape was available containing an assembler, MAC80, an interpreter, INT80, and a PL/M compiler all written in FORTRAN. So "all" we had to do was get it running, write the system text program and the linker, and adapt the download file format to the needs of the B500.

This report documents both the CYBER and the B500 software that we use at the moment, i.e., all programs sketched above except for the PL/M compiler.

Part one of this report documents the CYBER software, part two documents the B500 software.

PART ONE, CYBER SOFTWARE

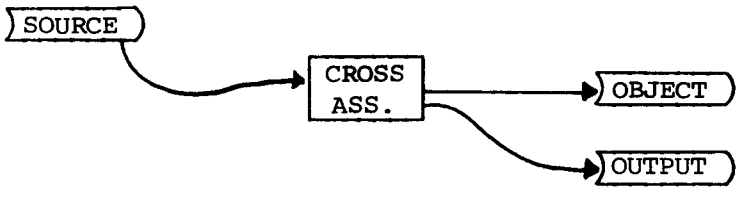
MAC80, an INTEL8080 cross assembler on the CYBER

NOS/BE Interface

To execute the MAC80 assembler, use the following JCL statements:

```
ATTACH,MAC80,ID=INFORMAT.
MAC80,SOURCE,OUTPUT,OBJECT.
```

The given filenames are the default ones. The files are used as follows:



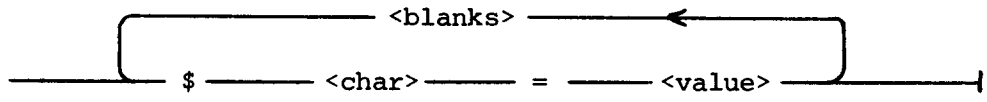
The cross assembler (MAC80) expects its input from SOURCE. MAC80 will read lines from column 1 up to 72. The file OUTPUT will consist of lines of width 72 and will contain the source text, error messages and the like. The line width 72 enables you to put OUTPUT on any console easily. The file OBJECT contains the symbol table, the object code in INTEL's standard hexadecimal object format and two link tables.

Example: Suppose your assembly code is on file SRC.
Then
MAC80,SRC.
will assemble SRC.

The assembly language is described in the INTEL Microcomputer User's Manual 98-135C.

ASSEMBLER SCANNER COMMANDS

Scanner commands are directives which influence the assembly process. Figure 1 shows their syntax.



These commands are recognized only if the line containing them starts with a $-sign in column 1. All information other than scanner commands is ignored on such a line. They can appear everywhere in the source text and may be redefined.

The following scanner commands are defined:

scanner command	value	result
A	0	The symbol table on the object file contains all defined symbols (including the register mnemo's).
	1 (default)	The symbol table contains only the absolute symbols (symbols used as labels or defined through an EQU pseudo).
B	0	Write the binary in BNPF format.
	1 (default)	Write the binary in standard Intel hexa-decimal format.
D	0 (default)	
	1	Every time the (operator precedence) parser decides to reduce, a dump is given on the output file of the working stacks, before and after the reduce action.
F	1 (default)	Issue a form feed every time a page has been completed.
	0	Generate a number of empty lines to simulate a form feed.
L	1...132 (default=1)	Everything to the left of the column position specified is ignored by the assembler.
M	0 (default)	
	1	Every time a macro definition is completed its symbol table entry is displayed.

P	0	Do not write a listing of the assembled source to output.
	1 (default)	Write a listing to file output.
Q	0 (default)	
	1	Every time a symbol table entry is made the symbol table is written to file output.
R	1...132 (default=72)	Everything to the right of the column position specified is ignored by the assembler.
S	1 (default)	Write a symbol table to the list file.
	0	Do not write a symbol table to the list file.
T	1 (default)	Write a symbol table to the object file.
	0	Do not write a symbol table to the object file.
W	72...132 (default=72)	"Width" of the output file. If a print line has a length greater than specified, it will be chopped of from the right.

Furthermore, if the command $$<blank> is issued, all command settings are displayed, if $$<char> is issued, only the setting of command <char> is displayed.

ASSEMBLER ERROR MESSAGES

A Address error: address referenced by a JMP or CALL instruction is not in the range 0 to 65535.

B Balance error: unbalanced parentheses or string quotes.

E Expression error: badly constructed expression (missing operator, missing comma, misspelled opcode).

F Format error: usually caused by a missing or superfluous operand.

I Illegal character: invalid ASCII character in string or digit to big for base in which it occurs.

M Multiple definition: two symbols declared which are identical or not unique in the first five characters.

N Nesting error: IF, ENDIF, MACRO or ENDM pseudo is improperly nested.

P Phase error: the value of an element being defined changed between pass1 and pass2.

Q Questionable syntax.

R Register error: register specified is invalid for this operation.

S Assembler internal stack overflow.

T Assembler internal symbol table overflow (e.g. too lengthy macro definitions).

U Undefined identifier.

V Illegal value: value exceeds range defined for particular operator (e.g. RST 8).

Extensions to the MAC80 assembler

Separate Assembly

When developping new software, one usually does not start from scratch, nor is it usual that the software is conceived and written in one piece and by one person. Normally one will make use of existing routines, and split up the program into smaller modules which are written and tested separately. Given these facts, one is facing the problem of using routines (or accessing information) of other modules, i.e., of using externally defined data or code.

With the original assembler, the only way was to use the absolute addresses, with the inevitable consequence that a local modification which changed the address would evoke changes in all software using it. A parallel can be drawn between this and the use of absolute addressing in routines, which causes difficulties as soon as one plans to insert some code.

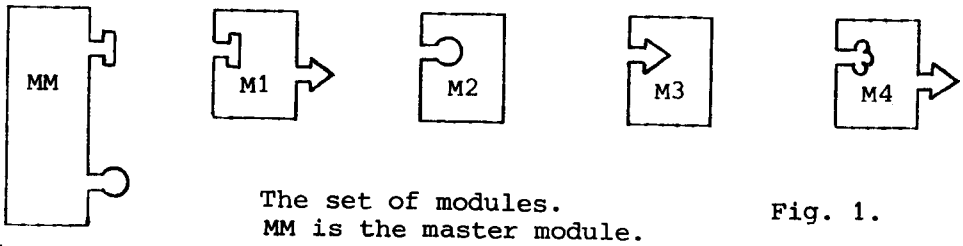
The solution to the latter problem was to use symbolic addressing instead of absolute addressing; the former problem will be tackled in a similar way. The idea is to use symbolic names for the externally defined data or code.

First some terminology:

A reference in one module to a symbol defined in another module is called an external reference (EXT).

A symbol in one module that is defined as being "visible" to the outside world, i.e., that can be referred to from other modules, is called an entry point (ENT).

EXT-s and ENT-s are each others counterparts. The process of linking a set of modules starting from a master module can now be defined as follows: for every EXT in the master module, the set of modules is searched for a matching ENT. If a matching ENT is found the module containing it is added to the master module. If a matching ENT is not found, then an error is reported. This process goes on until all EXT-s are satisfied. Figure 1 pictures this process:

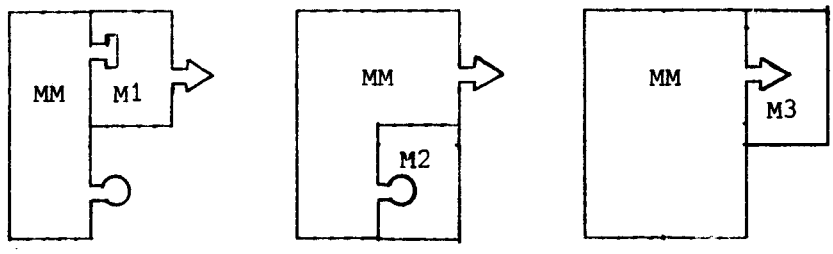


The set of modules.
MM is the master module.

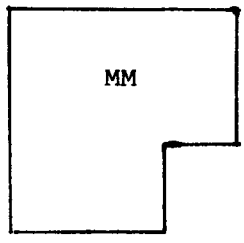
Fig. 1.

The holes on the left hand side of the modules denote ENT-s.
EXT-s are symbolized by the funny things sticking out of the right hand side of the modules.

The stages in the linkage process:



Stage 1 Stage 2 Stage 3



Result

The separate assembly facility is implemented using 2 tables: an entry table and an external table. These are written to the object file at assembly completion. The contents of the entry (external) table, which is described in further detail in the next section, is controlled by the ENTRY (EXT) pseudo instruction, of which figure 2 specifies the syntax.

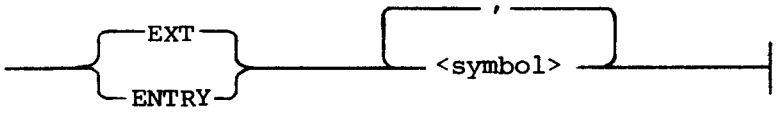


Fig. 2.

The obvious semantics is, that only these symbols specified on the ENTRY (EXT) pseudo will appear in the entry (external) table. These symbols are subject to certain restrictions:

A. Entry symbols must be absolute symbols; i.e., they must either be used as a label or be defined through an EQU pseudo.

B. External symbols, on the contrary, should not have a defining occurence in the module. The use of them is also somewhat more restricted than the use of "ordinary" symbols:

   1. They may only be used as an operand of a DW pseudo or in the (absolute) address field of a machine instruction (3 bytes instruction).
   2. They may not be used in an expression.

   Apart from these restrictions they are treated by the assembler as 16 bit quantities with zero value.

C. The maximum number of entry symbols in a program is 127.
The sum of the number of externals and the number of uses may not exceed 127.
If these limits are exceeded, the assembler will abort with a table overflow.
Violations of all other restrictions will result in Q errors: questionable syntax.

To distinguish entry and external symbols from the ordinary symbols in a program, the former are flagged in the symbol table on the listing: external symbols with a dollar-sign ($), entry symbols with an ampersand (&).

Enhancement of the macro-facility

The macro-facilities of the MAC80 assembler are limited to substitution of symbols. Sometimes, however, it comes in handy to be able to substitute parts of a symbol.
To this end a new (meta) character _ (underline)[1] is introduced, which can be used everywhere in a program. The effect, however, depends on whether it is used outside or inside a macro-definition.
Outside, the assembler is transparent to the underline (UL); i.e. the symbols A_B and AB are the same and the string 'A__PE' is equivalent to 'APE'.
Inside a macro-definition the UL-char does affect the assembler, because here it acts like a delimiter (if it is used outside a string); so the sequence A_B would now be interpreted as 2 symbols (A and B) separated by an UL, rather then as the single symbol AB. This means that if B was a formal parameter of a macro containing A_B, the last character in A_B would be substituted during expansion.
An example will clarify this:
Fig. 1 shows 2 macro's with which a set of machine instructions can be iterated, moreover these iterations can be nested.
The formal parameter loc of the first macro serves a twofold purpose:
1. loc should address a location in which the iteration count can be saved,
2. loc is used to create a label for the 2nd macro to jump to.
The until-macro has 2 parameters.
The second one (n) defines the number of iterations, the first one corresponds with the parameter of the first macro: it addresses the iteration count and it identifies the jump-address.

1) on some keyboards: ←

```
repeat macro loc
         sub a
         sta loc
l_loc:
       endm

until  macro loc,n
         lda loc
         inr a
         sta loc
         cpi n
          jm l_loc
       endm
```

fig. 1.

Fig. 2 shows the relevant parts of a program using the macro's. The 2nd column shows the resulting macro expansions.
Note that the sequence l_loc in the macro-texts are expanded into l_itr1 and l_itr2 because UL acts like a separator and that during the subsequent assembly of the expanded text they are interpreted as the symbols litr1 and litr2 because now the assembler ignores UL's.

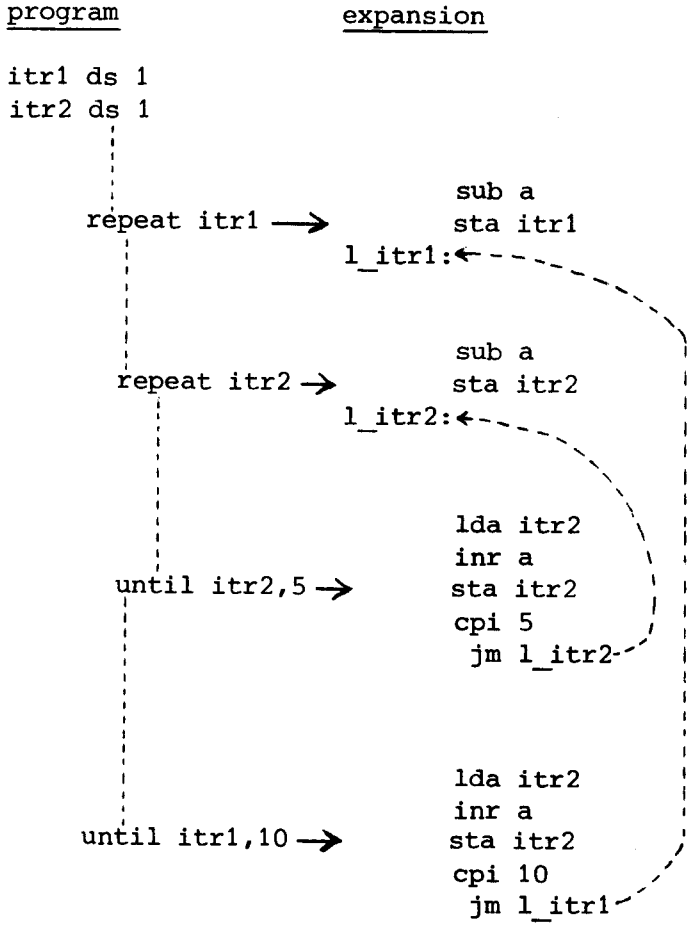


fig. 2.

Format of the object file as produced by the MAC80 cross assembler

The format will be explained using fig. 2 which shows the object produced by assembling the (nonsense) program of fig. 1.
The object consists of 4 tables: the symbol table (A), the load table (B), the external table (C) and the entry table (D).
All tables but the load table are optional.
Now follows a detailed description of each of the tables.

A. The symbol table contains a list of symbols together with their octal value and ends with a $-sign. The contents of this list depends on the $A scanner directive: if $A=1 (the default value) it contains only symbols which are used as labels or which are defined through the EQU pseudo; otherwise ($A=0) it will contain every defined symbol including the register mnemos.
   Note that symbols are truncated to the first 5 characters.

B. The load table consists of load records in hexadecimal form, which start with a colon. Each record consists of 5 parts (see fig. 2):
   1. RECORD LENGTH. This is the number of actual data bytes (part 4) in the record. A zero record length indicates the end of the load table.
   2. LOAD ADDRESS. This is the 4 digit address at which the first data byte of the load record is stored. The remaining data bytes are stored in successive memory locations.
   3. RECORD TYPE. This 2 digit code specifies the type of the record. Currently all load records are of type 0.
   4. DATA. The data is made up of 8 bit bytes represented by 2 hex. numbers.
   5. CHECKSUM. This is the 8 bit 2 complement value (complement +1) of the sum of hex. (2 digit) numbers in a load record starting with the RECORD LENGTH value and ending with the last data byte (last number in part 4).

   The last load record in the load table has a zero record length. The load address on this record specifies the location where the execution should start. The load table is followed by a dollar-sign.

C. The external table is divided into blocks, which start with an asterisk. The link information of each external (where it was used in the program) is collected into these blocks; each external in a separate block. The link information consists of sets of (octal) coordinates which point to the bytes in the load records which must be patched up by the linker. These bytes are characterized by 2 coordinates, the first points to the

load record containing the bytes, the second points to the actual bytes (see fig. 2).

The table, again, is closed with a dollar-sign.

D. The entry table starts with an ampersand and ends with a dollar-sign. It is a subset of the symbol table and contains all entry symbols (defined through the ENTRY-pseudo).

```
8080 MACRO ASSEMBLER, VER 3.1   12/09/79  11.15.22. ERRORS = 0 PAGE 1

  LINE   LOC     CODE
  00010.                                  EXT     READ,LINE
  00020.                                  ENTRY   START
  00030. 0000                             ORG     0000H
  00040. 0000    00010203                 DB      0,1,2,3,4
         0004    04
  00050. 0005    CD0000     ORIGIN:      CALL    READ
  00060. 0008    210000                  LXI     H,LINE
  00070. EEEE                             ORG     0EEEEH
  00080. EEEE    0000CDEF   START:       DW      READ,0EF0H
  00090.                                  END
NO PROGRAM ERRORS
```

Fig. 1.

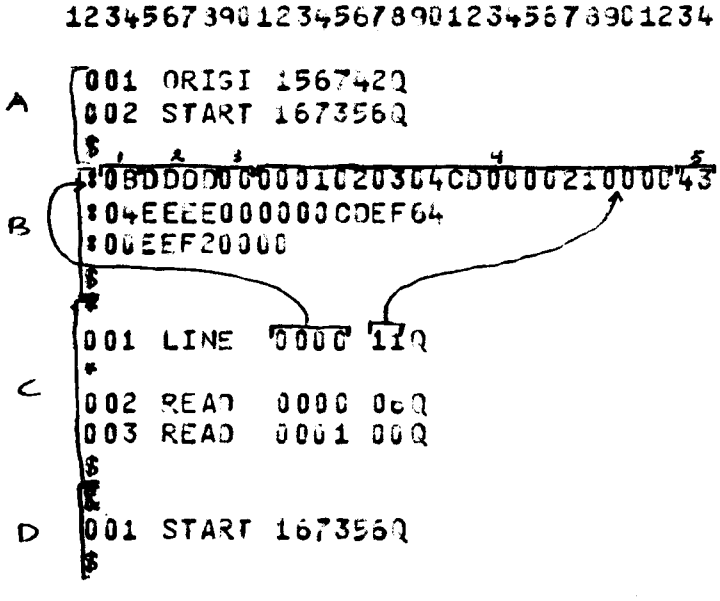


Fig. 2.

The first line is a column-count and is not part of the object.

ASM: a utility to facilitate the use of MAC80.

ASM resides on the permanent file ASM,ID=INFORMAT. It performs the appropriate file manipulations in connection with the MAC80 assembler. In case of a batch job the files OUTPUT and OBJECT are returned before assembly. After assembly, the file OBJECT is rewound.
When errors are detected during assembly, a program MACERR is called. This program processes the file OUTPUT and displays a brief explanation of the assembly errors. This feature is especially usefull when using ASM interactively. The error processor can be switched on and off by the ERR parameter (on=1, off=0) in the ASM procedure call.

Examples:

1. ASM,MYFILE,CODE,LIST,ERR=0.

will rewind and assemble MYFILE. CODE and LIST will contain the object code and the listing, respectively. No error processing will be done.

2. ASM.

will assemble with default options: source on SOURCE, object on OBJECT, listing on OUTPUT. Assembly errors, if any, will be processed by MACERR.

STXT80: A systemtext utility

This utility tries to satisfy all macro calls encountered in an INTEL 8080 assembly program. The program resides on an input file TEXT, the macro definitions reside on files (called systemtexts) STEXT1, STEXT2, etc. Satisfying means including macrodefinitions for which macro calls appear in the program.

The systemtexts are searched in the order in which they are specified by the user. The search is cyclic, that is: after the last systemtext has been searched and macro calls still remain unsatisfied, the search is started again until either all macro calls are satisfied or no calls are satisfied during a search through all systemtexts.
The result is written on the file SOURCE. If macro calls remain unsatisfied, STXT80 will abort, though after abortion SOURCE will contain the assembly program preceded by the macro definitions that were found.

There is, however, one restriction. The assembler allows a macro to be used as a parameter in a macro call. Macros thus implicitly referenced will not be satisfied.

The user can force copying information from a systemtext to the program even though this information is not referred to in the program. This is done by placing the information between a '$COPY' and a '$ENDCOPY' comment line.
E.g., if a system text contains:

```
;$COPY
Include this line
;$ENDCOPY
```

the line 'Include this line' is copied to the source file, regardless whether it satisfies any macro call or not. The copied text will be searched for macro definitions and macro calls.
For instance, by means of this mechanism the problems indicated above can be overcome.
The COPY directives must not appear inside macro definitions, because then they will not be detected (of course). Moreover, it is unclear what they mean in such a context.

The program is called as follows:

```
ATTACH,STXT80,ID=INFORMAT.
STXT80,TEXT,SOURCE,STEXT,STEXT2,STEXT3,...
```

TEXT, SOURCE and STEXT are default so

```
STXT80.
```

is equivalent to

```
STXT80,TEXT,SOURCE,STEXT.
```

LINK80: a linker

LINK80 is a utility which can link several independently assembled programs into an absolute load module.
Linkage is performed through certain tables on the object files, which are constructed by the MAC80-assembler using the symbols defined with the EXT and ENTRY pseudo's.

The linker is called as follows:

```
ATTACH,LINK80,ID=INFORMAT.
LINK80,MASTER,OBJECT,BIN1,BIN2...BIN9,BINA.
```

where MASTER is the name of the master binary
OBJECT is the name of the resulting linked load module (LLM)
BIN1...BINA are the names of up to 10 secundary binaries.

The default names are the same as the ones shown and the call

```
LINK80.
```

is equivalent to LINK80,MASTER,OBJECT.
So LINK80,,,TABLES,.
means that file MASTER contains the master binary, that there are 2 secundary binaries on files TABLES and BIN2 and that the LLM is written to file OBJECT.

The linker discriminates between one master binary, which determines the transfer address (the address at which execution should start) of the resulting LLM and the contents of the symbol table of the LLM (if present) and up to 10 secundary binaries which don't.
The optional symbol table contains the master symbol table, if one was present, together with all the (satisfied) external symbols of the master binary.
The LLM consists of the master binary together with those secundary binaries which were used during linkage; i.e. those binaries which contain an entry point which was referenced by a routine that had already been included in the LLM.
This can however be overridden by prefixing the name of a secundary binary (the master binary is always included) with an asterisk (*) on call.
i.e. LINK80,A,OBJ,*B,C.
means that the binary on file B is included in the LLM regardless whether it was used for satisfying an external reference or not.

LINK80 currently accepts 2 directives:

N - if this one is specified, the LLM will contain no symbol table

C - this one inhibits checksumming the loadrecords of each binary when they are read in.
Normally, a checksum is computed which is checked against the checksum provided by the MAC80-assembler.

These directives are specified on the control card.
They must be separated by a slash (/) from the names of the binaries, and should be the last parameter(s).
So LINK80,,OBJ,BIN/N,C. is a valid control card
whereas LINK80,,OBJ/N,BIN. is not.

Diagnostic (unfatal) errors:

1. UNKNOWN DIRECTIVE-X IGNORED
   This should be clear.
2. UNSATISFIED EXTERNAL REFERENCE-XXXXX
   ON FILE-ZZZZZZZ
   The binary on file ZZZZZZZ contains an external symbol XXXXX for which no matching entry symbol can be found.

Fatal errors:

1. TABLE OVERFLOW
2. STACK OVERFLOW

These messages flag internal table overflow, and indicate that the binaries are too big and/or contain too many external references.

3. ILL FORMATTED SYMBOL TABLE
4. ILL FORMATTED EXTERNAL TABLE
5. ILL FORMATTED ENTRY TABLE

These indicate format errors of the link tables, and should never occur with the use of the MAC80 cross assembler.

6. CHECKSUM ERROR

This indicates a bad checksum on some load file.
This error too should never occur with the use of MAC80.

These messages are followed by a second line.
   ON FILE-XXXXXXX
indicating which binary caused the error.

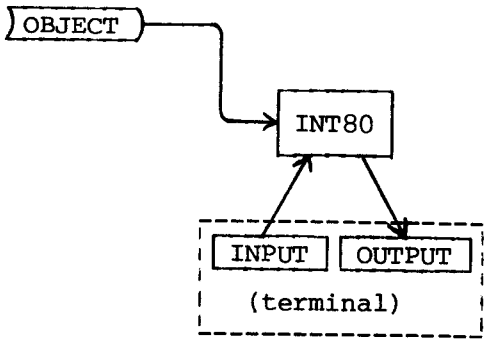
INT80, an INTEL8080 interpreter / debugger on the CYBER

NOS / BE Interface

To execute INT80, use the following JCL-statements:

```
ATTACH,INT80,ID=INFORMAT.
INT80,OBJECT,INPUT,OUTPUT.
```

The given filenames are the default ones. They are used as follows:



OBJECT contains the binary obtained from the assembler (or from the linker). If INT80 is used interactively, it connects INPUT and OUTPUT. INT80 expects commands from INPUT and writes messages to OUTPUT.

Some characteristics of INT80 are:

- linewidth on INPUT: 72
- memory of the INTEL: 10240 bytes (at this moment)
- initially all registers are zero (including the program counter PC)

Description Method and Command Format

The following describes the input commands. The underlined parts of keywords suffice; so you don't have to type

```
LOAD.
```

because

```
LO.
```

is enough.

We use a graph-like formalism (called syntax diagrams) to define the syntax of the commands. The rule of the game is: start left, follow the arrows till you get to the end.

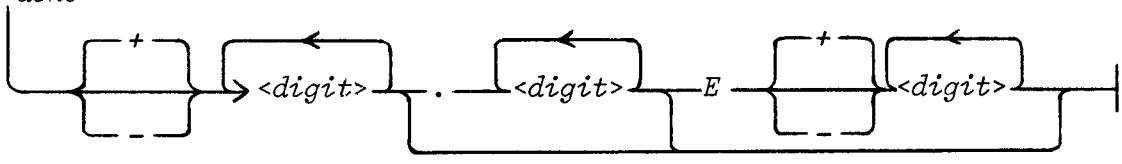
Example:

A real number denotation of the form:

$\underline{\pm}a.f\mathrm{E}\underline{\pm}e$

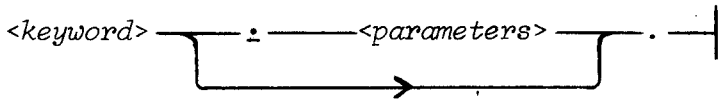
where: a, f and e are integers, is described as follows:

<*real deno*>



Brackets (<>) around a symbol indicate that it is nonterminal.
E.g., <*digit*> stands for the digits: 0,1,2,3,4,5,6,7,8,9 here.

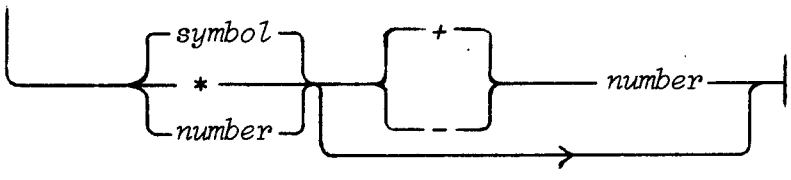
A command has the format:



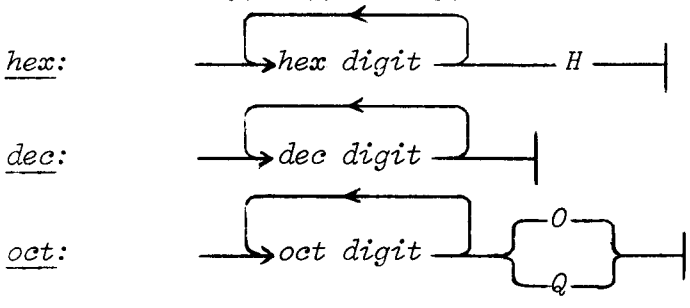
that means: a *keyword*, optionally followed by a *space* (denoted by ␣) and *parameters*, followed by a period.

We identify a command by its *keyword* and give a syntax diagram of its *parameters*, if it has any. We make use of some primitives:

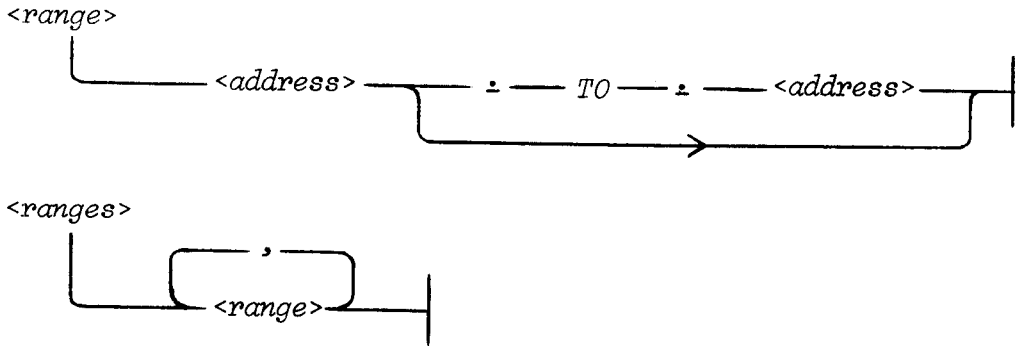
<*address*>



The *symbol* must be present in the symbol table. The *numbers* can be specified in hex, dec and oct:



'*' means: the value of the program counter.



<reg> : *one of the registernames or flags:*
HL, PC, SP, A, B, C, D, E, G, H, L,
CY (for carry), Z, S, P

The parts making up the parameters of a command must be separated by a space, even if it is not stated in the diagrams, so

DSYID.

doesn't make sense, whereas

D SY ID.

is a valid DISPLAY command.

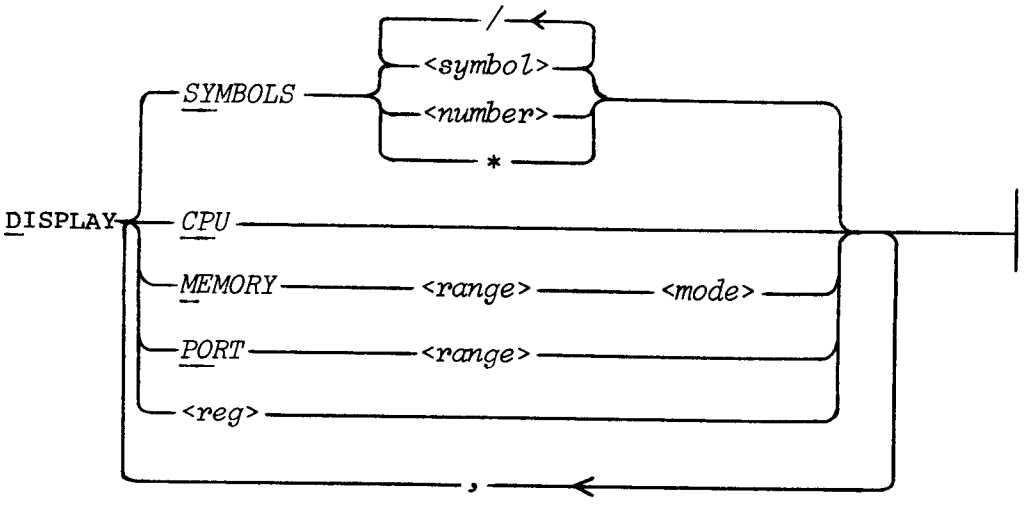
The commands in detail

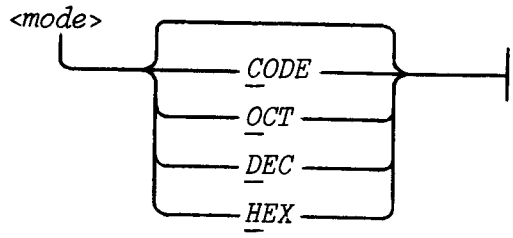
LOAD

Loads the binary (symbol table and object code) from the object file. This will generally be your first command.
If the load succeeds INT80 issues a "XX LOAD OK" message where XX denotes the size of the program in bytes.

CONV <ranges>

The ranges are displayed in binary, octal, decimal and hexadecimal.





SYMBOLS:

<number> : display label closest to the address specified by *<number>*.

<symbol1>/<symbol2>/ ...

: if $symbol_i$ is present in the symbol table its contents is displayed in octal, decimal and hexadecimal.

CAUTION:

The symbol table is scanned only once so the *<symbols>* must be given in the symbol table order! (The symbol table is the first part of your binary file.)

* : The contents of PC is displayed.

<any other char>

: the whole symbol table is displayed.

CPU : The contents of all registers and flags are displayed.

MEMORY : The *<range>* is displayed according to the active base (see BASE command) and, if specified, encoded as determined by *<mode>*:

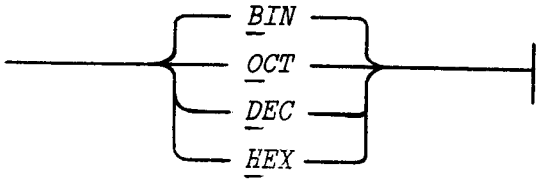
CODE: assembly language
address parts according to active base

BIN
OCT clear
DEC
HEX

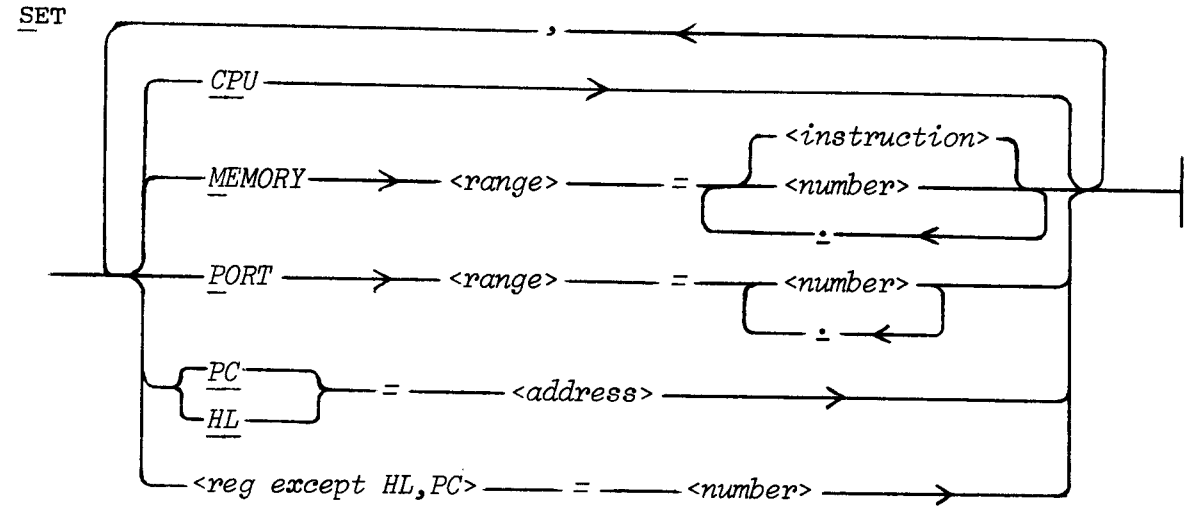
PORT : The ports in *<range>* are displayed.

<reg> : The register, register pair or flag is displayed.

BASE



The active base becomes the specified base.



CPU : Clear all registers and the timer (see TIME command)

MEMORY : The specified <*range*> according to the datalist is reset. If the datalist is shorter than the specified range the first element of the datalist is used as a filler.

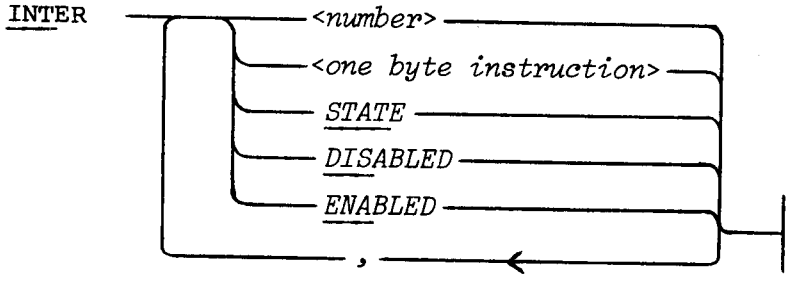
PORT : Ports are handled the same as the bytes in MEMORY.

HL,PC : reset HL or PC to <*address*>.

<*reg except HL,PC*>: reset specified register to <*number*>.

OUTPUT
INPUT
} <*ranges*>

The I- or O-flag of the specified ports are set.
Only if its flag is set the program can read or write data through the port by IN or OUT operations.



DISABLED
ENABLED
} : Disable/enable the interrupt mechanism completely.

STATE : The status (DIS or EN) is displayed.

<number> : *<one byte instruction>*

If the interrupts are enabled the parameter is executed next. In this manner one can simulate interrupts (by inserting an RST-instruction).

TRACE *<range>*

If code in *<range>* is executed, the following information is displayed:

- if a

  $b=0

  scanner command has been given the contents of all registers;
  if a

  $b=1

  scanner command has been given the contents of all modified registers.
- The executed instruction is always displayed.
  Addresses are encoded according to the active base.

NOTRACE *<range>*

Disables tracing for *<range>*.

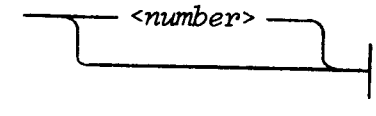
REFER *<ranges>*

Every time a byte in one of the specified *<ranges>* is referred to, a message is printed and, if requested, the nearest program label is displayed.
After that, the user gains control. The request is done by a scanner command:

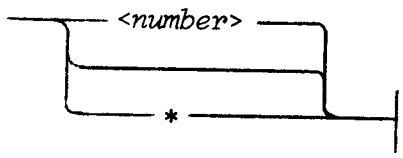
$GENLAB=1 the nearest label is displayed.
$GENLAB=0 it's not.

ALTER *<ranges>*

The same as for REFER, but now only if a byte in the specified *<range>* is changed.

TIME 

If the *<number>* is specified, the timer is reset to it. Otherwise the amount of cycles elapsed since the last reset is displayed.

GO 

Execution is (re)started at (PC). The parameter specifies the number of instructions that will be executed, before control is regained by the user:

<*number*> : clear

* : until termination

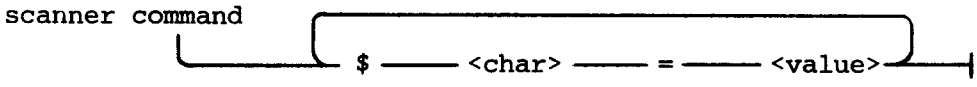
<*unspecified*>: default value, set by a scanner command maxcycle:

$MAXCYCLE = <*number*>

When the user gains control the contents of PC is displayed together with, depending on the scanner command, the closest program label.

Interpreter Scanner Commands

Three useful scanner commands were described ($B, $G, $M). The general syntax of scanner commands is:



where the first dollar sign has to start in column one. All information other than scanner command information is ignored on such a line.

Apart from the ones described above, two new scanner commands are defined:

char	value	description
H	int i (default 5)	print a header every i-th line of the trace.
T	0	do not display a prompting when a new command is to be typed in, to facilitate interactive use.
	1 (default)	display a prompting when a new command is to be typed in, to facilitate interactive use.
$	blanks	display the setting of all commands.
	char	display the setting of the command <char>.

Hints for using INT80

It should be clear that INT80 is a highly interactive program. If one wants to use it in the batch, many of its features will be of no use. If, e.g., INT80 is used in the batch with the following INPUT file:

```
LO.
$B=1
TR 0 TO 2560.
O 0 TO 255.
INP 0 TO 255.
G *.
1          <-
2          <- octal numbers as input
34         <- one per line
56         <-
```

and the program runs wild (after the GO * command) it will hand control back to the user, which means that INT80 will interpret the next input number as a command.
So, if you can, use INT80 from a terminal.

Termination

The scanner command $STOP will end the session.

INTERPRETER ERROR MESSAGES

Execution errors

1 Program counter stack overflow

2 Program counter stack underflow

3 Program counter outside simulated MCS-8 memory

4 Memory reference outside simulated MCS-8 memory

5 Invalid machine code operator

6 End of file while reading port input

7 Invalid port input data (not between 0 and 255)

Command mode errors

1 Reference outside simulated MCS-8 memory

2 Insufficient space remaining in simulated MCS-8 memory

3 End-of-file encountered before expected

4 Input file number stack overflow (max 7 indirect references)

5 Symbol not found in symbol table

6-9 Unused

10 I/O format command error (toggle has value other than 0 or 1)

11 Unused

12 Invalid cascaded labels. Must be of form X/Y/Z.

13 Invalid search parameter in display symbol command (must be symbolic name, address, or *)

14 Display symbols command invalid since no symbol table exists

15 Unused

16 Unrecognized command or invalid format in command mode

17 Missing . or extra characters following command

18 Lower bound exceeds upper bound or is less than zero in range list

19 The format of the symbol table is invalid (must be a sequence of the form N SY AD, where N is an integer, SY is the symbolic name, and AD is the address (in octal))

20 Invalid BNPF tape format (character other than N or P was encountered within the B....F field).

21 Invalid hexadecimal code format (bad hex digit, or missing :T)

22 Unrecognized display element or invalid display format

23 Symbolic name not found in symbol table

24 Invalid address or no symbol table present in display symbol command

25 Output device width too narrow for display memory command (use $WIDTH = N I/O format command to increase width)

26 Invalid radix in memory display command (must be code, bin, oct, or dec)

27 Unrecognized set element in SET command

28 Missing set list in SET command

29 Invalid set list or set value in SET command

30 Missing or misplaced = in SET command

31 Missing program stack element number in SET PS N command

32 Invalid interrupt code specification (either more than one byte, or element exceeds 255, or not a valid 8080 machine instruction)

Installation of the CYBER software

Generation of a new cross-assembler

MACUP is a random update library. By using update define directives for one or more of the symbols NOSYMB, NOCYB, NOLINK, NOB500, NOEDIT and NOMISC the user can determine which version of the cross-assembler MAC80 will be generated. The following section describes which parts of the assembler are controlled by each of the symbols. Note that the OMISSION of a definition for any of the symbols results in the effects stated. If all symbols are defined during generation, the original MAC80 assembler will be recovered.

NOSYMB - The allocated space for the symbol table and the various operator and operand stacks is doubled, thus allowing larger programs to be assembled.

NOCYB - This symbol controls a number of modifications, of which the first 3 will greatly speed up the compiler (the average speed up is 50%).

1. All logical, shift and mask operations are performed through the FTN built-in functions (albeit there remains the calling overhead).
2. Advantage is taken of the internal representation of the characterset, thus eliminating a slow table look up during the translation into internal code in the GNC-routine.
3. During the first pass the source text is packed (10 chars/ word) and stored in central memory thus avoiding disk-access and translations during the second pass.
4. The page header is modified to contain the time and date of assembly.
5. MAC80 identifies itself and prints the number of program errors and the assembly time in the dayfile if the assembler used the batch otherwise it will use the connected file ZZZZZOU, which is returned afterwards.

   The assembler aborts if program errors were detected.

   The assembly time is printed in the listing (following the symbol table).

   Also, messages resulting from erroneous scanner directives are printed in the dayfile or in ZZZZZOU.

   Keeping this symbol undefined during generation of a new assembler, will result in a CDC FTN extended program; moreover use is made of a number of small COMPASS routines which may cause difficulties if the assembler is used under a different operating system than NOS/BE.

NOLINK - 2 new pseudo instructions EXT and ENTRY are defined; the use of these pseudo's creates binaries in a format which allows them to be linked. These linkage symbols will be flagged in the symbol table in the listing.
The block information in the symbol table on the binary is removed.

NOB500 - Creates a binary in the Intel standard hex. format: the last load record contains the transfer address where execution should commence.
(Otherwise the last record will be zero.)

NOEDIT - A metacharacter _ (underline) is introduced which enables one to replace parts of a token during macro expansion.

NOMISC - Some miscellaneous modifications.

1. The default setting of 2 scanner directives is changed:
   $f=1 A form feed is generated at page end on the listing.
   $i=2 The assembler directly starts reading the source file.
2. A new scanner directive $A is defined.
   If $A=1 (the default) the symbol table on the binary will contain only the absolute symbols.
   I.e. those symbols which are either used as tables or are defined through an EQU pseudo.
   Otherwise ($A=0) the symbol table will contain all symbols used in the source text.
   This facilitates the use of the 8080 emulator (INT80) somewhat.
3. A line number (start value 10, increment 10) is printed before every line on the listing contained in the source text, for debugging purposes.
   Also every page contains a sub header denoting the start of the fields containing the line number (LINE), location counter (LOC) and code bytes (CODE).
4. The symbol table on the list file is printed with 55 lines/page.
5. A bug in the assembler.
   Originally the line count was not updated during a dump of the control parameters (scanner directives). This bug has been fixed.

Installation of MAC80

The installation of MAC80 depends on the kind of assembler that has been generated. If the symbol NOCYB was specified the resulting source is a more or less standard FORTRAN program. It can be installed with the following JCL commands:

```
RFL,65000.
FTN,I=COMPILE,OPT=2,UO,LTP=0.
LOAD,LGO.
NOGO,MAC80.
```

If NOCYB was not specified the resulting source contains a number of machine dependent optimizations written in COMPASS. This has two consequences for the installation process. Firstly, the amount of memory needed for compilation is increased to 100000. Secondly, two system texts are needed for assembly of the COMPASS routines. The installation is therefore done as follows:

```
RFL,100000.
FTN,I=COMPILE,OPT=2,UO,LTP=0,S=SYSTEXT,S=CPCTEXT.
LOAD,LGO.
NOGO,MAC80.
```

Installation of STXT80

The source of STXT80 must reside on the file STXT80S. It consists of two records. The first one contains a CCL-procedure which governs the compilation process. The second contains the actual program. To install STXT80 the following command suffices: STXT80S.
This will produce three files:
LIST - the source listing from FORTRAN and COMPASS,
STXTBIN - the relocatable object file,
STXT80 - the absolute object file.

Installation of LINK80

The source of LINK80 must reside on the file LINK80S and has the same structure as STXT80S. To install LINK80 issue the following command: STXT80S.
This, too, will produce three files:
LIST - the source listing,
LINKBIN - the relocatable object file,
LINK80 - the absolute object file.

Installation of ASM

The source of ASM must reside on the file ASMS. To install ASM issue the following command: ASMS. This will produce two files:

LIST - the listing of the programs MACERR and REMARK which perform the error post processing.

ASM - the object file.

Installation of INT80

INT80 is a more or less standard FORTRAN program and can be installed as follows:

```
FTN,I=INT80S,OPT=2,UO,LTP=0.
LOAD,LGO.
NOGO,INT80.
```

PART TWO, B500 SOFTWARE

The Upper Level Terminal Program

The processor communicates with the ourside world by means of input and output ports: the keyboard input port for the keyboard of course, the receiver input (primary receiver) en receiver output (primary transmitter) for the host computer. Our version of the B500 doesn't have AUXPORT hardware so we don't take AUXILIARY input/output into account. Furthermore the screen can be read and written, and the cursor can be positioned.

The following paragraphs shed some light on the way the Upper Level Terminal (ULT) program does input output. This description is meant for the usage of ULT in B500 programs. All routines presented here have no side effects apart from the described ones. All ULT-routine-identifiers are written in capital letters followed by their address in square brackets.

1. Keyboard input.

A keyboard interrupt causes a character to be placed in the keyboard buffer (in the scratchpad). Pointers relating to the keyboard and the screen are updated. The routine GETKD [012C] can be used to get the last character from the buffer into the A-register.

Example

The following routine waits for a keyboard interrupt and puts the character read in into the A-register:

```
KBDIN:  MVI  A,     100B
        OUT  30H  ; masking out non-KBD interrupts
        EI
        HLT
        PUSH H    ; DE & HL worden door
        PUSH D    ; GETKD vernield.
        CALL GETKD
        POP  D
        POP  H
        RET
```

2. The primary receiver.

When the host computer sends a character, this will cause a receiver interrupt. The way ULT handles the receiver interrupt is analoguous to a keyboard interrupt: the character is placed in the input/output buffer in the scratchpad. The parity bit is chopped off (to zero), which means that the host needs two characters to send over an 8 bit quantity. The routine GETRV [0108] can

be used to put the character in the A-register.

Example

The following routine waits for a receiver interrupt, and puts the character in the A-register:

```
RCVIN:  MVI  A,1B
        OUT  30H  ; receiver int. only
        EI
        HLT
        PUSH H
        PUSH B
        CALL GETRV
        POP  B
        POP  H
        RET
```

3. The primary transmitter.

At the moment this report is written the host computer doesn't echo characters sent to it. The ULT routine XMIT [030] waits for a transmitter-ready signal and transmits a character in the B character. This routine can be called by means of an RST 6 instruction (efficient but opaque).

Example

The following routine puts a character from the B-register on the line to the host:

```
SEND:  OUT  4   ; request to send
       PUSH PSW;
       RST  6   ; XMIT
       POP  PSW
       RET
```

4. The cursor.

The display memory of the B500 consists of location 2000 up to 2F9F hexadecimal, which is two pages of 25 lines of 80 characters.

To position the cursor on location 2xxx ($000 \leq xxx \leq F9F$) some memory reference must be made to position 3xxx. This means that location 3000 up to 3F9F cannot be used for other purposes. Throughout the whole ULT program the cursor position is kept in the DE-register pair.

Programs using ULT better stick to this convention!
The ULT routine CURST [018] sets the cursor position and can be called by means of an RST 3 instruction.

Example

The following routine positions the cursor on the location referenced by the DE-register pair:

```
STCUR:  PUSH H
        PUSH PSW
        RST  3  ;  CURST
        POP  PSW
        POP  H
        RET
```

5. Screen input/output.

Because the screen is viewed as just a piece of RAM, screen I/O is very simple. All LOAD and STORE instructions in the 8080 repertoire can be used for that.

Note, however, that the screen administration in the scratchpad, keeping information about the cursorposition, new lines, scrolling etc., must also be updated and that by just writing into RAM some special function associated to (control) characters are not executed. To this end either the routine WRITR [17F] (for all printable characters and almost all control characters) or the entry RAMMG in the subroutine KBD [217] (for escape sequences, line-feed and ETB code) can be used.
WRITR expects the character in the B-register, RAMMG [21B] expects it in the A-register.

The ULT routine DATOT [1A43] puts a string on the screen. The HL register pair is supposed to point at the beginning of the string. The string must be delimited by FF hexadecimal. The DE register pair must contain the screen location as usual. DATOT does not update the screen administration or execute function codes. The routine SFMSG does. SFMSG [137B] expects the same parameters as DATOT.

6. Upload and download.

6.1. Upload.

Apart from character-wise transput as described in paragraphes 2 and 3 it is also possible to send and receive blocks of data to and from the host. ULT offers

the following facilities for this.
By means of the functionkeys F1...F8 a chunk of RAM can be associated with a functionkey. (See the operators manual for details.) The chunk can then be uploaded by means of keying " (esc) d". The chunk is sent up in INTEL's "standard file format", which means that the host has to convert this format itself.

Example

you are ON LINE, type:

CONNECT,INPUT

COPYSBF,INPUT,FILE

go OFF LINE by means of RESET

now type: (esc)d

ULT will send the information associated with F1 to F8 to the host. If all goes well, ULT writes "END THE FILE" on the screen and goes ON LINE.

now type:

%EOF

so that COPYSBF will finish and INTERCOM will regain control.

Note that with this upload-function only "functionkey-RAM" can be sent to the host. If one wants to send an arbitrary chunk of RAM, the host has to be brought in the same condition as described above (it has to expect a file from the terminal) and a routine has to be called, which does the following:

```
PUSH <display address>
HL := <start address for upload>
DE := <number of locations>
A  := <upload stock size>; A=0 means: default block size (=18 hex)
JMP RAMU1 [1948]
```

6.2. Download.

Pushing a program from the host into the memory of the Bee is called downloading. The program file must have the standard INTEL download format as described in the B500 programming manual (table 2-4).
The software residing on the host (the assembler and the linker) generates files in this format.
A download file consists of a number of records, each containing (amongst others) a record length field, a load/address field, data, and a checksum.

ULT will put the data on the specified address while checking the sum. If a checksum error occurs it will ignore the rest of the file and put an error message on the screen. If the whole file has been read in without checksum errors ULT will "beep".

There is a little timing problem when downloading a file:

(i) The B500 has to be told not to act as a terminal anymore but as a computer receiving a file from its host.

(ii) The host has to be told to send a file to the B500.

To do (ii) we need the B500 as a terminal. But in order to be in time for the reception of the file (i) has to be done first.
This timing problem is solved by the following trick: you don't tell the host to send a file to the B500, but you tell the B500 to tell the host to send that file, and to go into computer mode immediately afterwards.
In our situation (ii) looks like:

```
COPYSBF,FILE,LOADF←
```

(← is return in "prog entry"-mode).
So what must be done is the following.
Get your files ready on the host, e.g.:

```
REWIND,FILE
CONNECT,LOADF
```

Go OFF LINE by means of RESET.
Put at the beginning of the screen message (ii), e.g.:

```
COPYSBF,FILE,LOADF←
```

Push the PROG LOAD button, which will cause ULT to send the message written on the screen to the host and wait for a download file.

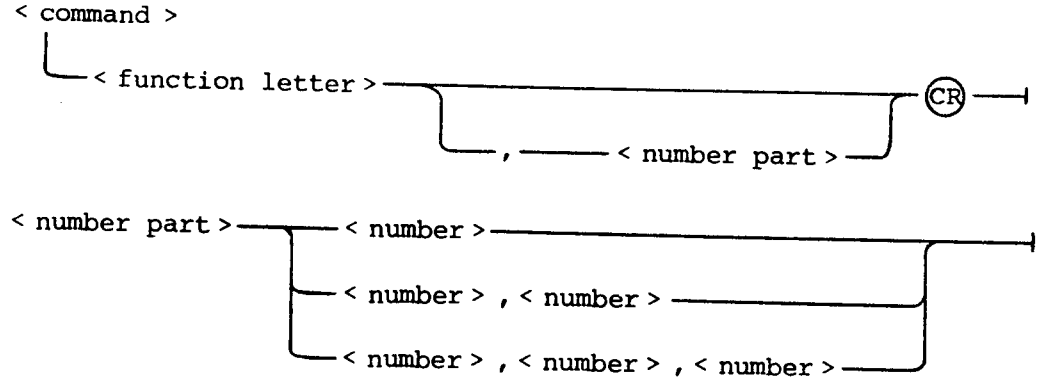
The Monitor.

The biggest B500 program written up till now is a monitor, which does all kinds of nice things for testing and debugging other programs. Within the monitor a user can

- run his program
- single step through his program
- upload files
- disassemble pieces of memory
- examine and modify pieces of memory
- move pieces of memory around

(and a few things more).

After the monitor has been downloaded and started (by means of " (esc)o"), commands can be entered.



<number> is a number of up to four hexadecimal digits.

In case of typing errors the monitor is reset by typing a RUBOUT.
The monitor knows one error message: "?", if that is what you call a message.

The following functions are implemented:

M (move) move a piece of RAM/ROM to some RAM area.

parameters 1: move-from start address
2: move-to start address
3: number of bytes to be moved

example: M,4000,2050,800
shows the function-key-area on the screen.

X (examine and modify)

parameters 1: start address

A memory location and its contents are showed on the screen. The contents can be changed by typing two hexadecimal digits followed by a carriage return. When a comma is typed in, the next location is displayed. RUBOUT gets you back to the monitor.

T (translate: disassemble a piece of memory)

parameters 1: start address
2: number of instructions to be disassembled.

S (single step trace)

parameters 1: start address
2: stop address

After executing one instruction, the CPU status, the next instruction and the top three words of the stack are displayed. The monitor waits for a space to execute the next instruction. To get back to monitor mode type a carriage return.

R (run)

parameters 1: start address

A program is executed starting at the indicated address. If the program ends with a return instruction, the monitor regains control.

U (upload a file)

parameters 1: start address
2: number of bytes

The host has to be waiting for a file. The monitor does not regain control. The B500 goes ON LINE so that the file on the host can be handled.

C (clean the screen)

W (wait)

Every three minutes a dummy-command is sent to the host to make sure the terminal doesn't get logged out.

Acknowledgement

We wish to thank Hans van den Engel for helping us through the first stage of getting the assembler running.

References

FORTRAN EXTENDED VERSION 4 REFERENCE MANUAL
Control Data Corporation

COMPASS VERSION 3 REFERENCE MANUAL
Control Data Corporation

NOS/BE VERSION 1 REFERENCE MANUAL
Control Data Corporation

INTEL Microcomputer User's Manual 98-135C

B500 PROGRAMMING MANUAL
Beehive International

B500 OPERATOR MANUAL
Beehive International