IMPROVING EFFICIENCY OF DEDICATED REAL TIME MICROCOMPUTER BASED SYSTEMS

by

Z. Barzilai, Z. Segall, E. Strasbourger

Technical Report #159
August 1979
ABSTRACT

Efficient table processing is essential for dedicated real time microcomputer based systems. Both hardware and software may be tailored to enhance programming and processing efficiency. A higher level assembly language (Sr.1AL) is presented as a software tool, and address mapping is presented as a hardware tool for efficiency improvement. Use of these tools is described to implement a microcomputer based PABX. The approach saved programming time, memory space, and improved execution throughput.

Key words: real time, memory mapping, table processing, structured programming.
I. TABLE PROCESSING PROBLEMS IN MICROCOMPUTER BASED SYSTEMS

An essential problem in the expanding applications of microprocessors is the need to improve table handling. Real time tasks for example, use tables to characterize equipment states and to communicate between tasks of differing time constraints. These tables are a natural result of top down design of application programs and data structuring. Advantages of table organization are:

1. good visibility of equipment and program status
2. easy system maintainability
3. obvious focal point for communication between tasks.

The advantages of table organization are offset by software and hardware limitations of microcomputers.

1.1 Software Problem

An implementation language must be chosen among a high level language which is interpreted or compiled and a low level (assembly) language. With the time and space constraints of real time problems, generally a low level language is chosen. Often a foreground-background system is designed. The foreground, written in assembly language, implements the severe time constrained tasks; while the background, written in a high level language, handles tasks with relaxed time constraints. This solution necessitates a two language system with language interfacing and maintainability problems.

Table processing in low level languages has the problems of poor symbolic table addressing and difficulty of understanding and maintaining table addressing constructs in these languages.
1.2 Hardware Problem

Table processing demands extensive address calculation capabilities. Generally in microprocessors the data bus is smaller than the address bus, and the size of internal registers matches the data bus size. This mismatch in sizes tends to make address calculation inefficient in microprocessors.

1.3 Improving Table Processing in Microcomputer-Based Systems

These table processing problems demand improving symbolic table addressing in low level languages (or improving time/space efficiency of high level languages) and improving address calculation efficiency in microcomputers. Our approach is to create an assembly language with high level syntax to improve symbolic table addressing, and to incorporate address mapping in microcomputers to improve address calculation efficiency.

1.4 Software Tools

To improve symbolic table addressing we have developed a general purpose high level Standard Microcomputer Assembly Language (SMAL) [1] for several microprocessors: M-6800, Z-80 and PDP-8 (see Appendix). SMAL uses unified self-explanatory English-language instructions. SMAL has been demonstrated to produce programs which, compared with normal mnemonic assembly language:

* take much less time to write
* are much less prone to coding and logic mistakes
* are easier modified, especially if the structuring facilities of SMAL are used
* are more portable
* are better maintainable
* will produce more efficient code
SMAL uses unified, consistent, self-explanatory instructions.

Examples:

<table>
<thead>
<tr>
<th>TYPE OF MICROCOMPUTER</th>
<th>MNEMONIC</th>
<th>SMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-6800</td>
<td>LDS #ARRAY</td>
<td>MOVE ADDRESS OF ARRAY TO SP</td>
</tr>
<tr>
<td>Z-80</td>
<td>LD SP, ARRAY</td>
<td>MOVE SP ADDRESS TO SP</td>
</tr>
<tr>
<td>8080</td>
<td>LXI SP, ARRAY</td>
<td>MOVE SP ADDRESS TO SP</td>
</tr>
<tr>
<td>M-6800</td>
<td>LDA A MEM</td>
<td>MOVE A TO MEM</td>
</tr>
<tr>
<td>Z-80</td>
<td>LD A, (MEM)</td>
<td>MOVE A TO (MEM)</td>
</tr>
<tr>
<td>8080</td>
<td>LDA MEM</td>
<td>MOVE A TO (MEM)</td>
</tr>
<tr>
<td>M-6800</td>
<td>STA B, X</td>
<td>MOVE B TO (X)</td>
</tr>
<tr>
<td>Z-80</td>
<td>LD (IX), B</td>
<td>MOVE B TO (X)</td>
</tr>
<tr>
<td>M-6800</td>
<td>CBA</td>
<td>IF B &gt; A THEN GO TO LOOP</td>
</tr>
<tr>
<td></td>
<td>BLT LOOP</td>
<td>IF B &lt; A THEN GO TO LOOP</td>
</tr>
<tr>
<td>M-6800</td>
<td>CPX #0</td>
<td>IF X IS NONZERO THEN GO TO LOOP</td>
</tr>
<tr>
<td></td>
<td>BNE LOOP</td>
<td>IF X IS NOT ZERO THEN GO TO LOOP</td>
</tr>
</tbody>
</table>

Compared with high-level microcomputer programming languages (e.g. PL/M) SMAL produces machine code which is more efficient by an order of magnitude. On the other hand, SMAL has many of the advantages of high-level languages, such as structured organization and control. Especially, SMAL includes:

* IF-THEN-ELSE statements
* nestable DO-blocks, to any depth
* declaration and procedure facilities.
As opposed to PL/M-like languages, any machine instruction is conveniently expressable in SMAL.

The SMAL-assembler is single-pass, and is also being used interactively. Errors are indicated line-by-line and may be immediately corrected.

To illustrate the improvement of SMAL over mnemonic assembly languages for table processing, consider a typical table operation using the M-6800 microcomputer (see Appendix for M-6800 SMAL definition). The operation is to transfer a one byte field NEWFIELD of each item in the table to a different one byte field OLDFIELD, as shown by the arrows in Figure 1.

<table>
<thead>
<tr>
<th>TABLE</th>
<th>name</th>
<th>age</th>
<th>old</th>
<th>number</th>
<th>new</th>
<th>field</th>
<th>field</th>
<th>date</th>
<th>field</th>
<th>last field</th>
</tr>
</thead>
<tbody>
<tr>
<td>first item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>second item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>last item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: A table operation example

Figure 2 shows a routine to implement this table operation. Compare the SMAL program (Fig. 2b) and its equivalent mnemonic program (Fig. 2a) for symbolic table addressability.
### Figure 2: Table processing example

(a) Mnemonc (assembler) program

(b) SMAL program
1.5 Hardware Improvement

Table address calculations may be accomplished using efficient data bus width calculations if address mapping is employed to increase the effective data bus width sufficiently to cover the required address space. Other advantages of address mapping improve the efficiency of specific microprocessors.

As an example two M-6800 deficiencies while working on tables are: lack of full-indexing, and lack of ability to add to the index register. These deficiencies may be overcome by adding address mapping registers. With address mapping all items in a table appear to occupy the same fixed location in page 0 memory. This approach enables an increment memory mapping register instruction to replace the 5 or 6 instructions required for index address calculation. Additionally, the X register is freed for double-byte moves and tests. Also, usage of page 0 "direct" addressing substitutes 2 byte 3 cycle page-zero instructions in place of 2 byte 5 cycle indexing and 3 byte 4 cycle extended addressing instructions.

A Mapping Implementation Example for the M-6800 Microprocessor

Mapping pages the address space as follows:

<table>
<thead>
<tr>
<th>8 bits</th>
<th>3 bits</th>
<th>5 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>area</td>
<td>offset</td>
</tr>
</tbody>
</table>

8-bit mapped area

If the page (most significant byte of the address) is not zero, then no mapping occurs. If the page is zero ("direct" addressing) then the most significant 3 remaining address bits (the area bits) address one of 8 mapping registers. The contents of the selected mapping register then
supply an 8 bit mapped area. This mapped area together with the 5 bit offset are supplied to the address bus as the effective address.

This mapping scheme partitions memory into 32 byte areas. At any given time 8 of these areas may be mapped onto the eight 32 byte areas in page 0. For our application, 32 byte tables are convenient, and 8K of memory is sufficient (i.e. 256 items in all tables combined). Therefore, we chose a 5 bit offset and 8 bit mapping registers. The eight mapping registers are currently located at addresses FFF0 through FFF7, and may be read or altered as regular memory. Note that real location 0100 for example may be accessed either directly at 0100 or at location 0000 if the contents of mapping register 0 is 8 or at location 0000 if the contents of mapping register 7 is 8, etc. Also note that page 0 locations themselves may only be accessed through a mapping register (where the mapping register is set to 0 through 7).

Implementation of address mapping demands high speed mapping registers so as not to show down the address bus. Thus, we are using 35 ns access time 3101 integrated circuits in the design.

To illustrate the advantages of memory mapping, the table operation of Figure 1 is programmed in Figure 2 with the mapping implementation just described.

Comparing the implementation of the table operation of Figure 1 with address mapping (as in Figure 3) against the implementation without mapping (as in Figure 2) reveal the following efficiency gains of address mapping for this typical table process:

Without address mapping requires 29 bytes with an execution time of 41 µsec per item (see Figure 2).
DECLARE TABLE AREA AT HEX 80,
    END OF TABLE AREA AT HEX CO,
    ITEM REGISTER AT HEX FFO,
    NEWFIELD AS 5,
    OLDFIELD AS 3

PROCEDURE MOVE NEWFIELD TO OLDFIELD
    MOVE # TABLE AREA VIA A TO ITEM REGISTER
    DO .EACH ITEM FOREVER
        MOVE NEWFIELD VIA A TO OLDFIELD
        INCREMENT ITEM REGISTER
        IF ITEM REGISTER VIA A = # END OF TABLE AREA
        THEN UNDO .EACH ITEM
    .END EACH ITEM
    END MOVE NEWFIELD TO OLDFIELD

**Figure 3:** Table processing example using address mapping.

With address mapping requires 19 bytes and 23 μsec per item (see Figure 3), a savings of 34% memory and 44% time.

If the fields to be transferred (NEWFIELD and OLDFIELD) are double bytes rather than single bytes, then without mapping takes an additional 4 bytes and 11 μsec, while with mapping requires no extra memory and an extra 2 μsec. Thus a savings of 42% memory and 52% time is realized by using address mapping.
II. IMPLEMENTATION OF A PABX USING IMPROVED TABLE PROCESSING

2.1 PABX Model

A Private Automated Branch eXchange (PABX) has been designed and implemented using the language and address mapping approach described in the previous section. This design uses a top-down structured approach partially discussed in [2-3,4] on a complex problem (PABX) requiring extensive table handling.

To illustrate the improved table processing techniques, a simplified model of the PABX will be presented. Figure 4 shows the hardware and software structure of a typical PABX. PABX hardware consists of:

1. line circuits to individual telephones
2. trunk circuits to other exchanges
3. switching network
4. junctor circuits to connect between lines and/or trunks
5. generator to produce ringing and busy and waiting signals
6. a microprocessor controller.

PABX software consists of time constrained signal analysis and synthesis device handlers (foreground) and time relaxed interconnection control and supervision and maintenance (background).

The foreground detects changes in line and trunk signal levels to ascertain on-hook/off-hook conditions and dialing status, as well as producing time dependent signals such as ringing. These changes of conditions are conveyed to background for execution of less urgent decisions, by means of changing entries in global tables. In the other direction, if a background task needs to initiate a time constrained function, the necessary information is passed to foreground by updating global tables.
SOFTWARE

HIGH LEVEL FUNCTIONS
- CALL PROCESSING
- MAINTENANCE AND ADMINISTRATION
- SUPERVISION

OPERATING SYSTEM
- SCHEDULING
- DATA RESOURCE HANDLING
- PRIMITIVE SIGNALLING
- TIMING

DEVICE HANDLERS

EXCHANGE HARDWARE
- LINE CIRCUITS
- INTERCONNECTING NETWORK
- JUNCTORS
- TRUNK CIRCUITS

ACCESS DEVICES

Figure 4: PABX structure
2.2 Hardware Configuration and Data Structure

The simplified model PABX must control up to 48 telephone lines with a maximum of 24 simultaneous conversations (junctors). The PABX has been implemented on a M-6800 with 8K of RAM and 8K of ROM. Complete line control is incorporated including conference calls, call forward, do not disturb, abbreviated dialing and many other features.

Memory is organized as in Fig. 5 with 8K of tables, 8K of programs, and hardware control and sensing locations. Line table organization is shown in Fig. 6 and junctor table organization in Fig. 7.

2.3 Program Example

Consider the situation that a "caller" telephone line has just finished dialing a "called" telephone line. If the called line is busy, the PABX must send a busy tone to the caller line. Otherwise, the PABX must find a free junctor. If no junctor is free then a special tone must be sent to the caller. If a junctor is free, then the PABX must instruct a free junctor to connect the caller line to the called line and also must send a ring to the called line and a ringing tone to the caller line.

Figure 8 shows a straightforward M-6800 program to accomplish this mission.

To gain efficiency in time and memory, this mission was programmed using address mapping. Eight mapping registers are used as in Figure 9.

To illustrate the relationship between the mapping registers and the memory, consider a situation where the 18-th junctor is used to connect a caller on the 7-th line with the 15-th line which he requested.
EVERY TABLE CONSISTS OF 32 BYTES

Figure 5: The Memory Organization
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LINE STATE</td>
</tr>
<tr>
<td>1</td>
<td>FUNCTION</td>
</tr>
<tr>
<td>2</td>
<td>HIGH OF LINE ADDRESS</td>
</tr>
<tr>
<td>3</td>
<td>LOW OF LINE ADDRESS</td>
</tr>
<tr>
<td>4</td>
<td>INDICATION</td>
</tr>
<tr>
<td>5</td>
<td>JUNCTOR</td>
</tr>
<tr>
<td>6</td>
<td>HOT LINE 1</td>
</tr>
<tr>
<td>7</td>
<td>HOT LINE 2</td>
</tr>
<tr>
<td>8</td>
<td>HOT LINE 3</td>
</tr>
<tr>
<td>9</td>
<td>HOT LINE 4</td>
</tr>
<tr>
<td>10</td>
<td>FIELD 1</td>
</tr>
<tr>
<td>11</td>
<td>FIELD 2</td>
</tr>
<tr>
<td>12</td>
<td>FIELD 3</td>
</tr>
<tr>
<td>13</td>
<td>FIELD 4</td>
</tr>
<tr>
<td>14</td>
<td>COARSE TIMER</td>
</tr>
<tr>
<td>15</td>
<td>FINE TIMER</td>
</tr>
<tr>
<td>16</td>
<td>BUSY OVERRIDE</td>
</tr>
<tr>
<td>17</td>
<td>PAST JUNCTOR</td>
</tr>
<tr>
<td>18</td>
<td>SPECIAL FIELD 1</td>
</tr>
<tr>
<td>19</td>
<td>SPECIAL FIELD 2</td>
</tr>
<tr>
<td>20</td>
<td>HIGH OF CONTROL</td>
</tr>
<tr>
<td>21</td>
<td>LOW OF CONTROL</td>
</tr>
<tr>
<td>22</td>
<td>HOLD DETECTED 1</td>
</tr>
<tr>
<td>23</td>
<td>HOLD DETECTED 2</td>
</tr>
<tr>
<td>24</td>
<td>CLOSED</td>
</tr>
<tr>
<td>25</td>
<td>OFF HOOK</td>
</tr>
<tr>
<td>26</td>
<td>CALL BACK</td>
</tr>
<tr>
<td>27</td>
<td>SECRETARY MANAGER</td>
</tr>
<tr>
<td>28</td>
<td>CALL FORWARD</td>
</tr>
</tbody>
</table>

**Figure 6:** Line Table
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HIGH OF JUNCTOR STATE</td>
</tr>
<tr>
<td>1</td>
<td>LOW OF JUNCTOR STATE</td>
</tr>
<tr>
<td>2</td>
<td>HIGH OF JUNCTOR ADDRESS</td>
</tr>
<tr>
<td>3</td>
<td>LOW OF JUNCTOR ADDRESS</td>
</tr>
<tr>
<td>4</td>
<td>JUNCTOR CALLER</td>
</tr>
<tr>
<td>5</td>
<td>JUNCTOR CALLED</td>
</tr>
<tr>
<td>6</td>
<td>JUNCTOR THIRD PARTY</td>
</tr>
<tr>
<td>7</td>
<td>JUNCTOR FOURTH PARTY</td>
</tr>
<tr>
<td>8</td>
<td>JUNCTOR FIFTH PARTY</td>
</tr>
<tr>
<td>9</td>
<td>JUNCTOR POSITION</td>
</tr>
<tr>
<td>10</td>
<td>JUNCTOR LINK OR TRUNK</td>
</tr>
<tr>
<td>11</td>
<td>JUNCTOR PRIVATE TRUNK</td>
</tr>
<tr>
<td>12</td>
<td>JUNCTOR PABX</td>
</tr>
<tr>
<td>13</td>
<td>JUNCTOR BUSY</td>
</tr>
<tr>
<td>14</td>
<td>JUNCTOR TRUNK COARSE TIMER</td>
</tr>
<tr>
<td>15</td>
<td>JUNCTOR CALL BACK TO ATTENDANT</td>
</tr>
<tr>
<td>16</td>
<td>JUNCTOR CAMP ON BUSY</td>
</tr>
<tr>
<td>17</td>
<td>JUNCTOR SPECIAL 1</td>
</tr>
<tr>
<td>18</td>
<td>JUNCTOR SPECIAL 2</td>
</tr>
<tr>
<td>19</td>
<td>JUNCTOR SPECIAL 3</td>
</tr>
<tr>
<td>20</td>
<td>JUNCTOR SPECIAL 4</td>
</tr>
<tr>
<td>21</td>
<td>JUNCTOR SPECIAL 5</td>
</tr>
<tr>
<td>22</td>
<td>JUNCTOR SPECIAL 6</td>
</tr>
<tr>
<td>23</td>
<td>JUNCTOR SPECIAL 7</td>
</tr>
</tbody>
</table>

Figure 7: Junctor Table
DECLARE CURRENT, JUNCTORS DOUBLEBYTE.

LD X: MOVE CAILRED, NUMBER TO A.

CLR 2 CLEAR B.

ASL 2 SHIFT A LEFT.

ROL 2 ROTATE B LEFT.

ASL 2 SHIFT A LEFT.

ROL 2 ROTATE B LEFT.

ASL 2 SHIFT A LEFT.

ROL 2 ROTATE B LEFT.

ASL 2 SHIFT A LEFT.

ROL 2 ROTATE B LEFT.

ADD 2 ADD LOW OF ADDRESS OF LINE TABLE TO A.

BCC 4 IF CARRY THEN INCREMENT B.

INC 2 INCREASE B.

STX 4 MOVE A TO LOW OF CALLED.

STA 4 MOVE B TO HIGH OF CALLED.

Figure 8: Program without Mapping
I

Figure 8 (cont'd): Program without Mapping
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFF0</td>
<td>DIAL.REGISTER</td>
</tr>
<tr>
<td>FFF1</td>
<td>THIS.DIAL.REGISTER</td>
</tr>
<tr>
<td>FFF2</td>
<td>EXTERNAL.ABBREVIATED.NUMBER.EXPANSION.REG.</td>
</tr>
<tr>
<td>FFF3</td>
<td>CALLER.REGISTER</td>
</tr>
<tr>
<td>FFF4</td>
<td>CALLED.REGISTER</td>
</tr>
<tr>
<td>FFF5</td>
<td>HOLD.REGISTER</td>
</tr>
<tr>
<td>FFF6</td>
<td>JUNCTOR.REGISTER</td>
</tr>
<tr>
<td>FFF7</td>
<td>DIAL.JUNCTOR.REGISTER</td>
</tr>
</tbody>
</table>

Figure 9: Mapping - Registers
Figure 10: Dynamic Mapping Mechanism
Figure 10 shows the contents of the mapping registers 3, 4, and 6, and in memory the 7-th line table, the 15-th line table, and the 18-th junctor table, with the junctor field (the fifth field), of the line tables, indicated as well as the caller field (the fourth field) of the 18-th junctor table. All items are accessed by the program in page zero locations, which are mapped into physical memory by concatenation with a mapping register. For example, the 18-th junctor table is in the 74-th area of memory (from location 0940 to 095F) where $74_{10} = 10010102$. The caller field is field 4 of the junctor table where $4_{10} = 001002$. Thus the 18-th junctor caller is contained in address 0944 of memory where $0944_{16} = 1001010001002 = 10010102$ concatenated with $001002$.

Figure 11 shows a M-6800 program using address mapping, which accomplishes the connection mission. Comparing the programs with and without mapping shows a 32% memory savings with mapping (77 bytes compared with 113 bytes without mapping) and a 35% time savings (415 microseconds compared with 640 microseconds) for the situation where the first 12 junctors are busy.
CONNECTED CALLER TO CALLED WITH MAPPING

DECLARE FIRST, JUNCTOR, AREA AS 56,
END OF JUNCTOR, AREA AS 80,
FIRST LINE, AREA AS 7,
JUNCTOR, REGISTER AS HEX FFF6,
JUNCTOR, STATE AS HEX CO,
JUNCTOR, BUSY AS 1,
JUNCTOR, CALLER AS HEX C4,
JUNCTOR, CALLED AS HEX C5,
CALLER, REGISTER AS HEX FFF3,
CALLER, STATE AS HEX 60,
CALLER, JUNCTOR AS HEX 65,
CALLED, REGISTER AS HEX FFF4,
CALLED, STATE AS HEX 80,
CALLED, JUNCTOR AS HEX 85,
RING AS 13,
RINGING, TONE AS 11,
BUSY, TONE AS 16,
SPECIAL, TONE AS 17,
CALLER, NUMBER,
CALLED, NUMBER,
THERE, IS A FREE, JUNCTOR

Figure 11: Program with Mapping
Program 11 (cont'd): Program with Mapping
CONCLUSIONS

Software and hardware tools have been presented to improve table processing in microcomputer based systems. These tools are useful for increasing programming and execution efficiency of real time systems such as a microprocessor based telephone exchange. A higher level Standard Microcomputer Assembly Language (SMAL) increases programming reliability and program flexibility and portability while decreasing programming time.

Address mapping decreases execution time and memory space, resulting in a system with greater throughput or larger capability. Using these table processing improvement tools, a PABX has been designed and implemented, confirming the viability and advantages of the concepts.
REFERENCES


SMAL - Standard Microcomputer Assembly Language

SMAL is a high-level assembly language, standardized for a variety of microprocessors. It is also useful in defining a given microcomputer from a software viewpoint. SMAL instructions for three computers are presented in this report: M-6800, Z-80, and PDP-8. Structured, self-documenting assembly language programs may be developed in SMAL. Experiments are being defined to evaluate SMAL as a tool for converting machine language programs between different types of microcomputers. For instance, a running program on the M-6800 may be automatically converted to M-6800 SMAL and then automatically or semi-automatically to Z-80 SMAL and then automatically to a Z-80 running program.

SMAL Syntax

* A lower case word is a non-terminal symbol, i.e. is explained on this page. Upper case letters and special characters are terminal symbols.

* Brackets [ ] denote a choice of one of the included options, unless there is only one option in which case the brackets mean to either include or to omit the option as desired.

* Triple dots ... means an indefinite repetition of the preceding option.

* A "comment" is any characters in a line following a semicolon.

* A "register" is one of a subset of the available registers. For example, in the M-6800 a "register" is one of a subset of \{A,B,X,XP,SP,CC\}, often the subset \{A,B\}, optionally preceded by the word REGISTER.

* A "number" may be decimal such as 25, or hexadecimal such as HEX 25 = 37, or octal such as OCTAL 25 = 21, or binary such as BINARY 1101 = 25.
* A "symbol" is a string of continuous alphanumerics which must begin with an alphabetic and may include periods as concatenators such as GET.NEXT_SYMBOL.

* A "label" is a symbol immediately followed by a colon such as NEXT.WORD;

* A "memory" is a location expressed as a symbol such as M, or a number such as HEX 25, or a displacement from an index value such as (X)-3, or an indirect reference such as '(M) or ((X)+5).

* A "character" is a single ASCII or EBCDIC character such as 'T' or '■'. Default is ASCII unless overridden by an EBCDIC instruction.

* A "string" is a string of characters surrounded by quotes such as "SMAL ACTIVE".

* A "line" of SMAL code is [label] [instruction] [comment] which may be blank or may contain just a label or just an instruction or just a comment or any two parts or all three parts. Only the IF instruction may be split into two lines with the THEN part appearing on one line and the ELSE part appearing on the next line.

```
  register  
character

  NUMBER

  ADDRESS OF

  CONTENTS OF

  number

  memory

where a number is a value unless preceded by CONTENTS OF and a

a symbol is a location unless preceded by NUMBER or ADDRESS OF.

  flag  [BIT] [IS] [SET]

  CLEAR

  NOT

A "condition" is

  RESULT test

  WHEN [INCREMENTED] [DECREMENTED] [By number] [IS] [test comparison]

  MATCHES [ANYWHERE] [NOWHERE] [source]
```
M-6800 INSTRUCTIONS

ADD [DECIMAL source] TO register [WITH CARRY]
AND source TO register
[CALL] address or MONITOR
CLEAR flag or register or address
COMPLEMENT flag or register or address
DATA LOCATION address
DECIMAL ADJUST A
DECLARE [name: number][BYTE[S] or DOUBLEBYTE[S]][INL implication or AT source],
[DATA : (].string or source ...[])]...
DECREMENT register or address [BY number]
DISABLE INTERRUPT
DO [name] [FOREVER]
    UNDO [name]
    REDO [name]
    END [name]
ENABLE INTERRUPT
EOR source TO register
GO TO address
IF condition THEN any instruction other than IF
    [ELSE any instruction other than IF]
INCREMENT register or address [BY number]
INPUT [FROM] address TO register
MOVE [FROM] source TO register or address
NEGATE register or address
OR source TO register
OUTPUT [FROM] register TO address
POP [INTO] register
PROCEDURE name
    END name
[PROGRAM] LOCATION address
PUSH register
REGARDLESS IF condition
RETURN [FROM name or MONITOR or INTERRUPT or HIM!]

ROTATE register or address LEFT or RIGHT [number BITS]

SET flag or register or address

SHIFT register or address LEFT or RIGHT [ARITHMETICALLY] [number BITS]

SUBTRACT source FROM register [WITH DORRÖN]

WAIT [FOR INTERRUPT]
Z-80 INSTRUCTIONS

ADD [DECIMAL] source TO register [WITH CARRY]
AND source TO A
[CALL] address
CLEAR BIT number OF destination
COMPLEMENT [A CARRY]
DATA LOCATION [number

DECIMAL ADJUST A

DECLARE [name [number]
DECREMEN destination
DISABLE INTERRUPT
DO [name][FOREVER]
UNDO [name]
REDO [name]
END [name]

[ENABLE] INTERRUPT [MODE [8080 CALL HEX INDIRECT CALL]

EXCHANGE [DE WITH HL
AF
BC DE HL
[HL X WITH STACK

FIND (HL) = A [INITIAL FORWARD BLOCK BACKWARD]
GO TO address
HALT

IF condition THEN any instruction other than IF
[ELSE any instruction other than IF]

INCREMENT destination

INPUT [INITIAL [FORWARD BLOCK BACKWARD]] source TO destination

MOVE [INITIAL [FORWARD BLOCK BACKWARD]] source TO destination

NEGATE A

OR source TO A

OUTPUT [INITIAL [FORWARD BLOCK BACKWARD]] source TO destination

POP [INTO] double register
Z-80 INSTRUCTIONS (continued)

PROCEDURE name 
    END [name]

PROGRAM LOCATION [number] [name]

PUSH douberegister
RExARDLESS IF condition

RETURN [FROM [name [NONMASKABLE] INTERRUPT]]

ROTATE destination [LEFT RIGHT] [DECIMAL][number BITS][BYPASSING CARRY]

SET [CARRY BIT number OF destination]

SHIFT destination [LEFT RIGHT] [ARITHMETICALLY][number BITS]

SUBTRACT [DECIMAL] source FROM register [WITH BORROW]

XOR source to A.
ADD address TO AC
AND address TO AC
[CALL] address
CLEAR AC or LINK or device FLAG
COMPLEMENT AC or LINK

DECLARE [name [number] [WORDS]] [INLINE [number] [AT name]] [DATA [number] [name] ...] ... 

DISABLE INTERRUPT
DO [name] [FOREVER]
UNDO [name]
REDO [name]
END [name]

ENABLE INTERRUPT

GO TO address
HALT

address WHEN INCREMENTED IS [ZERO NONZERO]
AC is [ZERO or NONZERO or POSITIVE or NEGATIVE or NONPOSITIVE or NONNEGATIVE]

IF device IS [HOT READY]
LINK IS [ZERO NONZERO]
THEN any instruction other than IF
[ELSE any instruction other than IF]

INCREMENT AC or address

INPUT [CHARACTER] [FROM] device TO AC
  address TO AC
  AC TO address
  SR TO AC
  device TO AC
  AC TO device
  LINK TO AC

NEGATE AC

OR [FROM] [device] TO AC

OUTPUT [CHARACTER] [FROM] AC TO device

PROCEDURE name
  END [name]

RETURN [FROM [INTERRUPT SUBROUTINE]]

ROTATE AC [LEFT [2 BITS]]

SET AC or LINK

WAIT UNTIL device IS READY

device may be KEYBOARD or TELEPRINTER or ITY