| | | | |
|---|---|---|---|
| bcc | title<br>MICRO Reference and User Manual | prefix/class-number.revision<br>MICRO/M-8 | |
| checked<br>R.R.V.S. | authors<br>Bo Lewendal<br>Bo Lewendal | approval date<br>9/2/69 | revision date |
| checked | | classification<br>Manual | |
| approved<br>Mel | | distribution<br>Company Private | pages<br>50 |

## ABSTRACT and CONTENTS

Reference and user manual for MICRO, the programming and reference language for the BCC microprocessors. The syntax and semantics of the language are defined and explained. An informative appendix is also included to aid the user in generating microprograms.

# Table of Contents

## 1.0 Introduction

MICRO is a special purpose programming language designed for use in writing code for the BCC microprocessors. The language is very machine dependent. Therefore it is mandatory that the prospective user have a good understanding of the functional characteristics of the microprocessors. Such an understanding may be acquired by reading conscientiously the various hardware documents on the subject.

It is possible to write microcode by simply assigning appropriate values to the various bits and fields of the microprocessor word. This however, is hard to do and produces very unreadable code. MICRO is designed to serve two purposes. The first is that of providing a convenient way of coding microprograms, and the second is that of providing a readable reference language for the communication of microprograms between mere people.

## 2.0 Declarations

The declaration logic of MICRO is present mainly for the convenience of the user. None of it, except possibly the ORG and RORG statements, is necessary for the coding of a program. We will proceed now to describe each type of declaration in detail.

2.1 Macros

MICRO macros have the following capabilities and incapabilities:

1. Can be used anyplace except as a subset of a name.
2. Can have up to nine arguments.
3. Concatenation is allowed.
4. Constant expressions may be evaluated and converted to strings during macro expansion.
5. Macros may expand to anything including strings representing more than one statement.
6. Nested and recursive macro calls are allowed so long as the user makes sure that the uppermost call will be finished. In other words, infinite recursion is not detected.
7. There is no conditional macro expansion machinery.
8. There are no repeat statements as in NARP.
9. Macro arguments may be null, but the number of arguments may not exceed the number specified implicitly by the macro declaration.

2.1.1 The syntax of a macro declaration is:

```
macro = "MACRO" mname "←" $(string[arg])  ";"
string = $(-";" (character / "↑;"))
arg = "*" no "*"
mname = word
```

2.1.2 The semantics is as follows:

1. The macro name must be alphanumeric and start with a letter and may be of any reasonable length.
2. If the macro name was previously defined then the

previous definition is lost and a message to that effect is elicited.

3. A macro may expand to the null string.

   (MACRO GOOM←;)

4. "↑" is an escape character allowing macros to expand to several statements:

   MACRO DOOM ← statement ↑; statement ↑; statement;

5. An argument indicator (arg = "*" no "*") may not have imbedded blanks. These are args: *1*, *4*, *9*. These are not args: * 3 *, *6 *, * 8*.

6. The number of arguments of a macro is defined as the value of the largest digit of the args, or zero if there are no args. The number of arguments of the following macro is seven:

   MACRO FOOM ← XX*2* XXX*7**7*; However, only the second and seventh arguments are used during the expansion process.

7. A macro may have no arguments:

   MACRO MO ← MACRO;

8. The following macro declaration has the effect of destroying the macro declaration facility:

   MACRO MACRO ← FOOL;

2.1.3 The syntax of a macro call is:

```
macro:call=mname["("[cstring["$"]cstring$(","cstring["$"]
    cstring)]")"]
cstring=$(-(";"/","/"("/")")(character/"↑;"/"↑,"/"↑("/"↑)"))
const=xconst $(("+"/"-"/"*"/"/")xconst)
```

xconst=["-"](digit$digit[("B"/"D")digit]/pname/"@"skname/
"*"/lname)

pname=parameter as defined in section 2.5.

"@" skname=the address of the scratch pad register skname.

"*" is the value of the location counter. lname=label.

2.1.4 The semantics is as follows:

1. A macro call need not have any arguments and if it does it may have missing arguments. "↑" is again an escape character.

Consider this macro: MACRO X ←*1*ZAP*3*;

X expands to "ZAP".

X() expands to "ZAP".

X(WOV) expands to "WOVZAP".

X(,COW, A HORSE) expands to "ZAP A HORSE".

X(A,B,C,F) does not expand, but elicits
"TOO MANY MACRO ARGUMENTS" from the compiler.

X X(FIE ,BLAH,)X(,,PY) expands to "ZAP FIE ZAPX(,,PY)"

X(X) expands to "XZAP".

X(X )X expands to "ZAP ZAPX".

X(X↑;,↑),X) expands to "ZAP;ZAPX".

2. A constant expression (<u>const</u>) is a sequence of constants (<u>xconst</u>) separated by any of the allowable arithmetic operators ("+","-","*","/").
The first constant may be preceded by a minus sign.
Evaluation of the expression is strictly from left to right, and parentheses are not allowed.

3. The "$" operator signals the macro expander to look for a constant expression which it then evaluates and converts to a string. A string of characters may be interposed between the "$" and the constant expression with the effect that the converted expression will be appended to the string. Consider the following macros:

```
MACRO FARM←ADD(*1*,*2*)**3*;
MACRO ADD←*1* PLUS *2*;
MACRO PLUS←+;
```

FARM expands to " + *".

FARM(7,8,9) expands to "7 + 8*9".

FARM(7+9*9) expands to "7+8*9 + *".

Assume SOX=6 and LOX=3.

FARM($7+8*9,Ø,3) expands to "135 + Ø*3".

FARM(BOX$SOX,LOX,$LOX) expands to "BOX6 + LOX*3".

FARM($BOX SOX,LOX,$LOX) expands to "BOX 6 + LOX*3".

ADD(LOX$SOX,3+$SOX/LOX) expands to "LOX6 + 3+2".

ADD($LOX$SOX,3+$SOX/$LOX) expands to "36 + 3+2".

ADD($ADD(3,6),$LOX) does not work because there is a limitation in MICRO which does not allow a macro call during the conversion of a constant expression to a string.

ADD(ADD(3,6),$LOX) does work and expands to "3 + 6 + 3".

## 2.2 Register declarations

Most of the microprocessor registers may be given symbolic names. These registers are: M, Q, Z, RØ to R6, SKØ to SK63, SKZ, OS, E1, and E2. SKZ is, strictly, not a register but that is beside the point. The upper registers M, Q, and Z are usually not given symbolic names.

In addition to defining the symbolic name internally in MICRO, the name is passed on to DDT so that the name may be used while debugging with the simulator.

The syntax of a register definition is:

```
    rdef = "DEFINE""REGISTER" prim "AS" rname";"
```

rname is the symbolic name to be associated with the prim which may be one of the registers mentioned above or another rname.

A convenient macro for defining registers is:

```
    MACRO REG←DEFINE REGISTER *2* AS *1*;
```

REG(SAVE,R5) would define "SAVE" as being the symbolic name for holding register R5.

## 2.3 Special condition declarations

Special conditions, of which there is a list in the appendix, may be defined using the declaration given by the following syntax:

```
    sdef = "DEFINE" "SCONDITION" sname "←" const ","
        "(" opcode ")" ["," "NOVCY"]";"
    sname = word
```

sname is the symbolic name for the special condition and const is its value which may range from Ø to 77B. opcode is a NARP opcode which will be executed by the simulator when the special condition in question is called in the program. It may be null, though it is normally a subroutine call. The optional part of the declaration ("," "NOVCY") is used to tell the compiler that VCY should not be set for that special condition. It normally is set. Here again the symbolic name is passed on to the simulator.

Following is a convenient macro for special condition declarations:

```
    MACRO SC←DEFINE SCONDITION *1*←*2*,(*3*)*4*;
```

SC(JAM,37B,SBRM SPAM) would define "JAM" as being the symbolic name for special condition 37B. Subroutine "SPAM" would be called when the simulator encounters the special condition. Also, VCY is set. If we did not want VCY set, then the following macro call would be used: SC(JAM,37B,SBRM SPAM,↑, NOVCY). However, in this case, one might as well say:

```
DEFINE SCONDITION JAM←37B,(SBRM SPAM),NOVCY.
```

There are a number of special conditions which are already defined in MICRO. They are the left cycle operations, scratchpad address from Z flag, and the memory operations. A special condition name may be redefined, but a message to that effect will be output by MICRO.

## 2.4 Branch condition declarations

Branch conditions are defined almost in the same way as special conditions. A list of branch conditions is in the appendix. The syntax is as follows:

```
    bdef = "DEFINE" "BCONDITION" bname [","bname1] "←"
        const ","" (" opcode ")" [","  "NOVCY"]";"
    bname = 1$(-(","/";"/" ") character)
    bname1 = word
```

The semantics is the same as that for special condition declarations except for bname and bname1.

bname may consist of any characters besides ",", ";", and " ". This means that branch condition names such as "RØ<=Ø" are possible. However, whenever such a non-alphanumeric name is used the alphanumeric name bname1 must be supplied. bname1 is then the name passed on to the simulator since it must have an alphanumeric name.

A macro for defining branch conditions would look essentially the same as the one for special conditions except for the possibility of a second name. Redefining a branch condition name again elicits a message from MICRO.

Several branch conditions are defined internally in MICRO and will be discussed in the section on branch instructions.

## 2.5 Parameter definitions

Parameters exist in MICRO solely for the convenience of the coder. Of course, it is almost a necessary convenience if the user's program is to be changed frequently.

The syntax of a parameter declaration is:

```
    pdef = "DEFINE" "PARAMETER" pname "←" const ";"
```

A parameter pname may be used anyplace where a const may be used. A parameter may be redefined as a parameter without eliciting a message from the compiler. This is so in order that computations involving constants may be done during compile time. Examples follow:

```
    MACRO PM←DEFINE PARAMETER *1*←*2*;
    PM(RAP,Ø) sets RAP to Ø
    PM(RAP,RAP+1) increments RAP ; in fact
    MACRO INC←PM(*1*,*1*+1) allows one to say INC(RAP).
    PM(RAP,@SKNAME+BASE) sets RAP to the sum of parameter
        BASE and the address of scratch pad register SKNAME.
```

A good example of the use of parameters is the package designed to implement field logic in MICRO. These macros were designed by Bob Van Tuyl and are described in the appendix.

A constant, const, has the following syntax:

```
    const  = xconst $(("+"/"-"/"*"/"/") xconst)
    xconst = ["-"](digit$digit[("B"/"D")[digit]] / pname /
             lname / "@" skname / "*")
```

```
skname = rname / "SKØ" / "SK1" / ... / "SK63"
```

## 2.6 Origin relocation

Normally, the location counter points to the word which is being, or is about to be, assembled. Two commands exist in MICRO which allow the user to modify the location counter: ORG and RORG. The special symbol "* ", incidentally, has as its value the value of the location counter, and is treated as a constant.

### 2.6.1 ORG statement

An ORG statement looks like: "ORG" const";".
It has the effect of setting the location counter to the value of the constant <u>const</u>.

### 2.6.2 RORG statement

A RORG statement takes no argument. It simply resets the location counter to the value it had before the last ORG statement. The argument of an ORG statement does not get stacked, so one may not have an ORG statement between another ORG and a RORG statement.

## 2.7 Labels

Labels, though they are a part of executable instructions, are still declarations. A label declaration has the following syntax:

```
    ldef = [$(lname":")] inst ";"
    lname = word
    inst = executable instruction; to be defined later
```

Note that comments and other declarations may not be inserted between the labels and the instruction. An instruction may have any number of labels. Each of the labels becomes defined as the value of the location counter and may thereafter be used just as a constant.

Only the last label of an instruction is output to the simulator. It is possible to redefine a label, but of course the compiler will output a message that this has been done.

## 3.Ø Instructions

Each instruction of the user program, terminated by a semi-colon, is scanned from left to right and compiled into bits of 9Ø-bit microprocessor word. The location counter is incremented <u>after</u> an instruction is compiled.

The syntax for an instruction is as follows:

```
    inst = partial:exp $("," partial:exp)
    partial:exp = branch / memory:op / special:cond / assn /
        field
```

Normally, the order of the partial expressions does not matter. There are a few exceptions, however, and these will be covered in the specific sections describing each of the five types of partial expressions.

## 3.1 Assignment instructions

The assignment instruction is the main and most complex type of partial expression. Its syntax is:

```
    assn = [ref la] exp
    la = "<-"[("Y"/"X")"<-"]
    ref = prim $(la prim)
    prim = "M"/"Q"/"Z"/"RØ"/"R1"/"R2"/"R3"/"R4"/"R5"/"R6"/
        "OS"/"E1"/"E2"/"SKØ"/"SK1"/.../"SK63"/"SKZ"/rname
    exp = bool["LCY" const/("LCH"/"LCL")(const/"Z")/
        ("+"/"-"/"!") bool][("+"/"-") const]["MRG"const]
    bool = ["NOT"] (prim/const)[("OR"/"AND"/"EQV"/"EOR")
        ["NOT"] prim]
```

Many of the combinations allowed by the above syntax are illegal from the viewpoint of the functional characteristics of the microprocessors. These illegalities will be described below. Of course, the meaning of the legal combinations will be described also.

### 3.1.1 References and primaries

In order to be able to use the microprocessor registers effectively one should be aware of which busses they can be loaded from or read onto. In addition, some registers can only be read.

The M, Q, and Z registers may each and separately be loaded from either or both of the two main busses (X and Y). Also, the M register may be loaded from the main memory under control of the central memory interface. The two boolean boxes are used to generate any of the 16 possible logical functions of M and Q or Z and Q. The outputs of the boolean boxes may be put through the adder, or the left boolean box output may go through the cycler. In either case, the final output goes onto the X buss.

The holding registers RØ to R6 may be loaded from the X and/or Y buss. They may be read only onto the Y buss. The RØ register is loaded, but not read, independently of the other holding registers. Therefore, it is possible to load, at the same time, one of R1 to R6 <u>and</u> RØ. It is not possible to read two holding registers.

The OS, E1, and E2 registers may only be read onto the Y Buss. They cannot be loaded. The E1 and E2 registers are actually busses, not registers.

The scratch pad registers SKØ to SK63 may be loaded from the X buss and they may be read onto the Y buss. SKZ is not a a register, but signifies that the scratch pad address be taken from the Z register. Reading or loading the scratch pad register takes 200 nsec. So VCY is automatically set by the compiler when the scratch pad is referenced.

In the definition of prim, rname is of course a symbolic name of a register as discussed in section 2.2 on register declarations.

A reference, ref, consists of a sequence of primaries separated by assignment operators, la. The assignment operator "←X←" indicates that the expression exp is to be forced to go onto the X buss. "←Y←" is handled analogously. Each of the primaries listed in the reference is to be loaded from either the X or Y buss. If the buss is not specified, then if there is a choice of busses the X buss will be used. The only case where there is a choice is when a constant is being referenced. It is an error to try to use both the X and Y buss in a single reference.

### 3.1.2 Boolean Expressions

A boolean expression, bool, may consist of either a constant or one of the 16 possible logical functions of either M and Q or Q and Z. The possibilities are listed in the table which follows. The value associated with each possibility is the value to which one must set BL or BR to generate that particular function.

| VALUE | LEFT BOOL BOX (BL) | RIGHT BOOL BOX (BR) |
|---|---|---|
| Ø | M AND Q | Z AND Q |
| 1 | M EQV Q | Z EQV Q |
| 2 | Q | Q |
| 3 | NOT M OR Q | NOT Z OR Q |
| 4 | M | Z |
| 5 | M OR NOT Q | Z OR NOT Q |
| 6 | M OR Q | Z OR Q |
| 7 | -1 | -1 |
| 1ØB | Ø | Ø |
| 11B | NOT M AND NOT Q | NOT Z AND NOT Q |
| 12B | NOT M AND Q | NOT Z AND Q |
| 13B | NOT M | NOT Z |
| 14B | M AND NOT Q | Z AND NOT Q |
| 15B | NOT Q | NOT Q |
| 16B | M EOR Q | Z EOR Z |
| 17B | NOT M OR NOT Q | NOT Z OR NOT Q |

In the boolean function table above it does not matter in which order the operands appear. "M OR Q", for example, is the same thing as "Q OR M".

### 3.1.3. Arithmetic expressions

There are two types of boolean expressions (bool). The first type is a logical expression involving registers M, Q, or Z. The second type is not an expression, but is simply a primary (excluding M, Q, and Z) or a constant.

Examples of type 1: M AND Q, Q, Q EOR Z.

Examples of type 2: RØ, R5, OS, E2, SK6, SKZ, const.

Only the first type of boolean expression may be operated on by the adder or the cycler. All arithmetic operations on boolean expressions require that the two booleans not emanate from the same boolean box.

"+" performs addition of two booleans. One may be added to the resulting expression and a constant may be merged with the final resulting expression.

Examples: M + Z, Q + Z + 1 MRG 4B7, M + Z + Ø MRG 77B,
NOT M AND Q + Q EOR Z + 1 MRG 77B6.

Illegal: M OR Q + Z + 3, M + Q - 1.

"!" does the same thing as "+" except that VCY is not set as it normally is, It may be used only when it is known that no carry will be generated. In other words, "!" acts as a merge under the right conditions.

"-" performs two's complement subtraction. One may be subtracted from the resulting expression and a constant may be merged with the final resulting expression.

Examples: Z - M, Q - Z -1 MRG 3301B, M - Q MRG 10,
NOT Z OR Q - Q EQV M -1 MRG 4B7.

Illegal: M AND Q - Z - 3, M - Q +1.

Cycle operations require that the boolean expression emanate from the left boolean box. In other words, only logical expressions involving M and Q may be cycled. The cycler and adder cannot be operated simultaneously.

Following is a description of the cycle operations.

bool "LCY" const: The cycle count, const, must be Ø,1,2,3,4,8,12,16, or 2Ø

bool "LCL" (const/"Z"): The cycle count is taken from the two low order bits of either the constant or the Z register.

bool "LCH" (const/"Z"): The cycle count is taken from bits 19,2Ø, and 21 of either the constant or the Z register.

After the cycle operation a constant may be merged with the resulting expression, but nothing may be added.

Examples: M AND Q LCY 8, M LCY Ø, Q EOR M LCL Z,
M OR Q LCL 15, NOT Q LCH Z, NOT M LCH 15,
NOT M AND NOT Q LCY 16 MRG 77B5.

Illegal: Q AND Z LCY 8, M LCY 15, Q LCH Z + 6.

Expressions involving type 2 booleans may only be of the form: const/prim ["+" const] ["MRG" const]. If the expression is a constant then the constant is gated onto the appropriate buss, or the X buss if no buss is specified (either explicitly like "←Y←", or implicitly like "SK1 ← const"). The holding registers are the only non-upper registers which can have a constant added to them, and this constant must be 1. Any non-upper register may have a constant merged with it.

Examples: 6,7ØB,4B7/3,SK3 MRG 77B,SKZ,R1,R2+1,RØ+1 MRG 6,
R6 MRG 3, OS MRG 3, E1,E2 MRG 77B6,*.

Illegal: SK3+1,SK3+77B, 6 MRG 7, OS+6, E1+1 MRG 7B7.

### 3.1.4 Assignments

An assignment assn may consist solely of an expression exp. In this case the expression will be gated onto the appropriate buss, but no register will be loaded from that buss.

If a reference, ref, and an assignment operator, la, are present then each of the primaries of the reference will be loaded from the buss onto which the expression was gated.

Following are numerous examples of assignments:

```
    M ← Q ← Z ← 1
    M ← R1 ← RØ ← NOT M AND Q + 1 MRG 77B
    SKZ ← M OR Q - Q EOR Z - 1
    Q ← SK63 ← R6 + 1 MRG 4B7
    M
    Q AND NOT M + NOT Z + 1 MRG 7Ø1B
    RØ + 1
    E2
    Z ← RØ ← E1 MRG 1
    63 + 1ØØB2 * 7
    R5 ←SK1Ø
    - 1 MRG 1     (the - 1 comes from the boolean box)
    M ← Z RØ ← M AND Q LCY 12
    R1 ← SK8 ← Q LCH Z MRG 77B3
    Q ← Q LCL 23
```

Here are some illegal assignments:

```
    M ← Q ← RØ + 3
    SK6 ← RØ + 1
```

Illegal assignments (continued)

R1 ← $\underline{\text{R2}}$ ← M + Z

$\underline{\text{E1}}$ ← $\underline{\text{OS}}$ ← M ← 3

E2 $\underline{+}$ SK6

M + Q $\underline{+\ \text{Z}}$

Q ← Q LCL 23 $\underline{+\ 2}$

M $\underline{+\ \text{Z}}$ LCY 8 MRG 3

$\underline{\text{Z}}$ LCH Z

$\underline{\text{SK3}}$ LCY 4

RØ ← R5 ← $\underline{\text{SKZ}}$ ← R5 + 1

### 3.2 Memory operations

The syntax for a memory operation is as follows:

```
    memory:op = ("FETCH"/"PREFETCH"/"HFETCH"/"STORE"/
                "PRESTORE"/"HSTORE"/"OFETCH"/"OHFETCH")
                [assn] / "RESET"
```

For memory operations, M is the data register and RØ the address register. The optional assignment after a memory operation is intended to be the source of the address. The compiler actually prefixes the assignment with "RØ←" so that "FETCH SK3 ← M + Q" becomes "FETCH RØ ← SK3 ← M + Q" and is equivalent to "RØ ← SK3 ← M + Q, FETCH". "RESET" does not take an address, so it can't have an optional assignment.

Memory operations are special conditions. This means that no other special condition may be used while accessing the memory. Especially troublesome conflicts occur when one tries to do a cycle operation or access the scratch pad with address in the Z register simultaneously with a memory operation.

## 3.3 Branch instructions

A branch instruction has the syntax:

```
    branch = ("GOTO"/"DGOTO")(assn/const)[cond]/("CALL/"DCALL")
                    const [cond]/("RETURN"/"DRETURN")[cond]
    cond   = "IF" bname / "ON" assn relop "Ø"
    relop  = "="/"#"/(">"/"<")["="]
```

A GOTO or DGOTO (deferred GOTO) can have either an assignment or a constant as an argument. If the argument is an assignment the branch address is taken from the X buss, and hence the assignment should use the X buss instead of the Y buss. If the branch address is constant it is placed in field B of the microprocessor instruction word.

A CALL or DCALL can have only a constant branch address while a RETURN or DRETURN may have none at all.

An unconditional branch is one without the optional branch condition cond. If cond is present, however, the branch will occur if the branch condition is true.

The construct "IF" bname is used whenever the branch condition bname has been declared using the declaration described in section 2.4.

There are     branch conditions predefined in MICRO for which the construct "ON" assn relop "Ø" is used. These conditions are:

X = Ø, X # Ø, X > Ø, X < Ø, X >= Ø, X <= Ø, Y < Ø, and Y>= Ø, where X is the X buss, Y is the Y buss, and M is the M register.

The compiler decides, after compiling the assn, what is being tested and which relation the test consists of. The appropriate branch condition is then automatically selected. All branch conditions besides these ten must be declared.

Following are some typical branch instructions:

```
               DEFINE BCONDITION RØ>=Ø, RØGEZ ← 12B, (QCALL RØGEZF);
               DEFINE BCONDITION ATT1SET ← 36B, (QCALL ATT1SETF);
       FOO:    GOTO 1ØØB;
       SAM:    DGOTO SAM IF ATT1SET;
               CALL ZAP ON M ← Q ← M OR Q LCY 4 MRG 3 >= Ø;
               GOTO FOO IF RØ>=Ø;

       ZAP:    DRETURN ON M + 1 MRG 6 < Ø;
               GOTO ZAP ON M ← M + 1 # Ø;
```

## 3.4 Special condition instructions

A special condition instruction is defined simply as special: cond = sname where sname is the name of a special condition as defined by the declaration discussed in section 2.3.

There are a number of predefined special conditions which are used in MICRO and which have already been described. They are the memory operations, cycle operations, and addressing the scratch pad from the Z register.

Following are some examples of the use of special conditions:

```
           DEFINE SCONDITION ALERT ← 14B, (QCALL ALERTF);
           DEFINE SCONDITION POT ← 15B, (QCALL POTF);
    GOG:   ALERT, DGOTO GOG;
           GOTO GOG ON M < Ø, POT;
```

The following will not work:

```
           M ← M LCY 16, ALERT;
           FETCH Z, POT;
           SKZ ← M AND Q, ALERT;
```

3.5 Field assignment

Sometimes the user finds himself in a situation where it is not possible to code an instruction using the standard MICRO language. In this case the user must resort to specifying the actual bits and fields of the 90-bit instruction word. The syntax for doing this is:

```
    field = fname ["←"const]
    fname = ".MC"/".MCONT"/".DGO"/".B"/".IHR"/".TCX"/
            ".TCY"/".TSPY"/".THY"/".TXW"/".TYW"/
            ".TAX"/".LOC"/".SSP"/".TOSY"/".LRØ"/
            ".LSPX"/".VCY"/".MS"/".RRN"/".LRN"/".LMX"/
            ".LMY"/".LQX"/".LQY"/".LZX"/".LZY"/
            ".BL"/".BR"/".TE1Y"/".TE2Y"/".C"
```

As an example, the following two statements are equivalent:

```
    Q ← Z ← M EOR Q LCH Z MRG 77B3;
    .BR ← 1ØB, .BL ← 16B, .MS ← 12B, .C ← 77B3, .TCX, .LQX,
        .LZX;
```

A description of each field may be found in the appendix.

## 4.∅ Miscellaneous features

### 4.1 Program

A program is defined by the following syntax equations:

```
    program = $statement "END" ";"
    statement = decl / [$ label] inst ";" /
                "*" $(-crlf character) crlf /
                "@" $(-crlf character) crlf
```

A line whose first non-blank character is an asterisk "*" is considered to be a comment and is completely ignored except that it is output to the expanded file which will be mentioned in the section on the operation of MICRO.

If "@" is the first (not first non-blank) character, then the whole line up to the carriage-return-line-feed is output to the object file except for the "@". The purpose of this is to enable the user to write NARP code in MICRO. For example, the user may wish to keep in his MICRO code the NARP subroutines which are called when the simulator encounters user defined special conditions and branch conditions.

## 5.∅ Operation of MICRO

The compiler for MICRO exists as a subsystem called MICRO. If the subsystem is not on the drum it may be retrieved from KDF file ()MICRO. The symbolics are on KDF files ()1MIC and ()2MIC and may be assembled using NARP. Starting location is "GO".

When called, MICRO asks for the source file, object file, and optionally the expanded file. The object file is the file onto which MICRO puts the NARP code representing the microcode. This file is then mangled by Paul Heckel's macro infested NARP to produce a binary file which may then be loaded with Paul's simulator. If the object file name is terminated by comma instead of period, MICRO asks for the expanded file which is a file onto which MICRO dumps the source code with all macros expanded. Also, for each instruction, the compiled value of each field of the microprocessor word is output along with a listing of which bits are set.

Unlike QSPL, MICRO does not have any confusing rubout logic. Micro may be dumped just like NARP in order to preserve declarations and macros.

At the end of compilation, MICRO outputs to the teletype the number of microwords compiled, execution time used, and various statistics concerning tables in the compiler:
S = number of characters of string storage remaining, M = number of words remaining in the macro table, H = number of entries remaining in the symbol table, and K = largest scratch

pad address used.

## 6.0 Interface between MICRO and the simulator

The code which MICRO produces must be assembled with a special version of NARP which is cluttered up with numerous macros. This program is called FNARP and is written by Paul Heckel.

To use FNARP do the following:

```
     @():FNARP.
        SOURCE FILE:  <object file produced by MICRO>.
        OBJECT FILE:  <binary file to be loaded with DDT>.
```

No errors should occur when using FNARP. If errors do occur, then there is either a serious problem with MICRO or FNARP, or else the errors are due to NARP code introduced by the user via the "@" feature of MICRO.

The way in which the object file produced by FNARP is loaded and run in the simulator is discussed in detail in a separate document written by Paul Heckel.

If FNARP is not on the drum, it may be read from KDF file (PIRTLE)FNARP.

## 7.0 Appendix

1. Syntax of MICRO
2. List of branch conditions
3. List of special conditions
4. Bit assignment of microprocessor word
5. Summary of fields
6. Macros to implement field logic

## A1 Syntax of MICRO

```
program = $statement "END" ";"

statement = decl / [$label] inst ";" /
            "*" $(-crlf character) crlf /
            "@" $(-crlf character) crlf

decl = macro / rdef / sdef / bdef / pdef /
            "ORG" const / "RORG"

macro = "MACRO" mname "<-" $(string[arg]) ";"

string = $(-";" (character / "↑;"))

arg = *1* / *2* / *3* / *4* / *5* / *6* / *7* / *8* / *9*

mname = word

rdef = "DEFINE" "REGISTER" prim "AS" rname ";"

rname = word

sdef = "DEFINE" "SCONDITION" sname "<-" const ","
            "(" opcode ")" ["," "NOVCY"] ";"

sname = word

bdef = "DEFINE""BCONDITION" bname ["," bnamel] "<-"
            const "," "(" opcode ")" ["," "NOVCY"] ";"

bname = 1$(-("," / ";" / " ") character)

bnamel = word

opcode = $(-("(" / ")") character)

pdef = "DEFINE" "PARAMETER" pname "<-" const ";"

pname = word

label = $(lname ":")

lname = word

inst = partial:exp $("," partial:exp)

partial:exp = branch / memory:op / special:cond /
            assn / field
```

```
branch = ("GOTO" / "DGOTO") (assn / const) [cond] /
         ("CALL" / "DCALL") const [cond] /
         ("RETURN" / "DRETURN") [cond]
cond = "IF" bname / "ON" assn relop "Ø"
relop = "=" / "#" / (">" / "<") ["="]
memory:op = ("FETCH" / "PREFETCH" / "HFETCH" / "STORE" /
         "PRESTORE" / "HSTORE" / "OFETCH" / "OHFETCH")
         [assn] / "RESET"
special:cond = sname
assn = [ref la] exp
la = "←" [("Y" / "X") "←"]
ref = prim $(la prim)
prim = "M"/"Q"/"Z"/"RØ"/"R1"/"R2"/"R3"/"R4"/"R5"/"R6"/
         "OS"/"E1"/"E2"/"SKØ"/"SK1"/.../"SK63"/"SKZ"/rname
exp = bool ["LCY" const / ("LCH" / "LCL")(const / "Z") /
         ("+" / "-" / "!") bool][("+" / "-") const]
         ["MRG" const]
bool = ["NOT"] (prim / const) [("OR" / "AND" / "EQV" /
         "EOR") ["NOT"] prim]
field = fname ["←" const]
fname = ".MC"/".MCONT"/".DGO"/".B"/".IHR"/".TCX"/
         ".TCY"/".TSPY"/".THY"/".TXW"/".TYW"/
         ".TAX"/".LOC"/".SSP"/".TOSY"/".LRØ"/
         ".LSPX"/".VCY"/".MS"/".RRN"/".LRN"/".LMX"/
         ".LMY"/".LQX"/".LQY"/".LZX"/".LZY"/
         ".BL"/".BR"/".TE1Y"/".TE2Y"/".C"
const = xconst $(("+" / "-" / "*" / "/") xconst)
```

```
xconst = ["-"] (digit$digit[("B" / "D") digit] / pname /
         lname / "@" skname / "*")
skname = rname / "SKØ" / "SK1" / ... / "SK63"
word = letter $(letter / digit)
macro:call = mname ["(" cstring ["$"] cstring $("," cstring
         ["$"] cstring) ")"]
cstring = $(-(";" / "," / "(" / ")")(character / "↑;" /
         "↑," / "↑(" / "↑)"))
```

A2 List of branch conditions

The starred conditions are predefined in MICRO and need not be defined by the user.

| Value | Condition |
|---|---|
| Ø* | Never branch |
| 1* | Always branch |
| 2* | X = Ø |
| 3* | X # Ø |
| 4* | X < Ø |
| 5* | X >= Ø |
| 6* | X > Ø |
| 7* | Y >= Ø |
| 1Ø* | Y < Ø |
| 11 | RØ < Ø |
| 12 | RØ >= Ø |
| 13* | X <= Ø |
| 14 | Not X AND 777777B = Ø |
| 15 | Not X AND 777777B # Ø |
| 16 | Z >= Ø |
| 17 | Z < Ø |
| 2Ø* | Always branch |
| 21 | Y AND 7 # Ø |
| 22* | BL = Ø |
| 23* | BL # Ø |
| 24 | Y even |
| 25 | Y odd |

| Value | Condition |
|---|---|
| 26 | Attention latch 1 not set, reset |
| 27 | Request strobe latch 1 = Ø and request strobe latch 2 = Ø |
| 3Ø | Protect # X |
| 31 | Request strobe latch 2 = Ø |
| 32 | Special flag A not set |
| 33 | Special flag A set |
| 34 | Attention latch 2 not set, reset |
| 35 | Attention latch 3 not set, reset |
| 36 | Attention latch 1 set, reset |
| 37 | Undefined |
| 4Ø | Undefined |
| 41 | Undefined |
| 42 | Local memory parity error = Ø, reset |
| 43 | Undefined |
| 44 | Central memory parity error = Ø, reset |
| 45 | Breakpoint = 1 |
| 46-77 | Undefined |

## A3 List of special conditions

The starred conditions are predefined in MICRO and need not be defined by the user.

| Value | Condition (function) |
|---|---|
| Ø* | No action |
| 1* | LCY 1 |
| 2* | LCY 2 |
| 3* | LCY 3 |
| 4* | LCY 4 |
| 5* | LCY 8 |
| 6* | LCY 12 |
| 7* | LCY 16 |
| 1Ø* | LCY 2Ø |
| 11* | LCL Z |
| 12* | LCH Z |
| 13* | SKZ (scratch pad address in Z) |
| 14 | ALERT |
| 15 | POT |
| 16 | PIN |
| 17 | Request strobe 1 |
| 2Ø | Unprotect |
| 21 | Unusable |
| 22 | Load memory request priority field |
| 23 | Reset request strobe latch 1 |
| 24 | Reset central memory request |
| 25 | Protect |

| Value | Condition (function) |
|---|---|
| 26 | Reset device attached to I/O connector |
| 27 | Undefined |
| 3Ø | Set special flag A |
| 31 | Reset special flag A |
| 32 | Reset request strobe latch 2 |
| 33 | Request strobe 2 |
| 34-37 | Undefined |
| 4Ø | Release |
| 41 | Prestore |
| 42 | Store |
| 43 | Store and hold |
| 44 | Fetch |
| 45 | Fetch and hold |
| 46 | Undefined |
| 47 | Prefetch |
| 5Ø-57 | Undefined |
| 6Ø | Set bank B |
| 61 | Set bank A |
| 62 | Clear map |
| 63 | Undefined |
| 64 | Oddword fetch |
| 65 | Oddword fetch and hold |
| 66-77 | Undefined |

A4 Bit assignment of microprocessor word

| Bit | Name | Bit | Name |
|---|---|---|---|
| Ø | MC(Ø) | 15 | B(7) |
| 1 | MC(1) | 16 | B(8) |
| 2 | MC(2) | 17 | B(9) |
| 3 | MC(3) | 18 | C(Ø) |
| 4 | MC(4) | 19 | C(1) |
| 5 | MC(5) | 2Ø | C(2) |
| 6 | MCONT(Ø) | 21 | C(3) |
| 7 | MCONT(1) | 22 | C(5) |
| 8 | B(Ø) | 23 | C(6) |
| 9 | B(1) | 24 | C(7) |
| 1Ø | B(2) | 25 | C(8) |
| 11 | B(3) | 26 | C(9) |
| 12 | B(4) | 27 | C(1Ø) |
| 13 | B(5) | 28 | C(11) |
| 14 | B(6) | 29 | C(12) |

| Bit | Name | Bit | Name |
|---|---|---|---|
| 3Ø | C(12) | 45 | TSPY |
| 31 | C(13) | 46 | THY |
| 32 | C(14) | 47 | TXW |
| 33 | C(15) | 48 | TYW |
| 34 | C(16) | 49 | TAX |
| 35 | C(17) | 5Ø | LOC |
| 36 | C(18) | 51 | SSP(Ø) |
| 37 | C(19) | 52 | SSP(1) |
| 38 | C(2Ø) | 53 | SSP(2) |
| 39 | C(21) | 54 | SSP(3) |
| 4Ø | C(22) | 55 | SSP(4) |
| 41 | C(23) | 56 | SSP(5) |
| 42 | IHR | 57 | TOSY |
| 43 | TCX | 58 | LRØ |
| 44 | TCY | 59 | LSPX |

| Bit | Name | Bit | Name |
|---|---|---|---|
| 6Ø | MS(Ø) | 75 | LQY |
| 61 | MS(1) | 76 | LZX |
| 62 | MS(2) | 77 | LZY |
| 63 | MS(3) | 78 | BL(Ø) |
| 64 | MS(4) | 79 | BL(1) |
| 65 | MS(5) | 8Ø | BL(2) |
| 66 | RRN(Ø) | 81 | BL(3) |
| 67 | RRN(1) | 82 | BR(Ø) |
| 68 | RRN(2) | 83 | BR(1) |
| 69 | LRN(Ø) | 84 | BR(2) |
| 7Ø | LRN(1) | 85 | BR(3) |
| 71 | LRN(2) | 86 | VCY |
| 72 | LMX | 87 | DGO |
| 73 | LMY | 88 | TE1Y |
| 74 | LQX | 89 | TE2Y |

A5 Summary of fields

| Field | Use |
|---|---|
| MC | Branch condition field. |
| MCONT | Instruction sequence control. |
| B | Branch address. |
| C | 24 bit constant field. |
| IHR | Increment holding register output. |
| TCX | Gate constant field onto X buss. |
| TCY | Gate constant field onto Y buss. |
| TSPY | Gate scratch pad register onto Y buss. |
| THY | Gate holding register selected by RRN onto Y buss. |
| TXW | Transfer X buss to holding register input. |
| TYW | Transfer Y buss to holding register input. |
| TAX | Gate adder output onto X buss. |
| LOC | Adder low order carry. |
| SSP | Select one of 64 scratch pad addresses to be loaded or read. |
| TOSY | Gate OS register onto Y buss. |
| LRØ | Load holding register RØ from X buss or Y buss. |
| LSPX | Loads scratch pad word addressed by SSP or Z register from the X buss. |
| MS | Special condition field. |
| RRN | Specifies one of 7 holding registers to be read into the incrementer. |

| Field | Use |
|---|---|
| LRN | Specifies one of the holding registers R1-R6 to be loaded from the X or Y buss. RØ cannot be specified in this way. |
| LMX | Load M from X buss. |
| LMY | Load M from Y buss. |
| LQX | Load Q from X buss. |
| LQY | Load Q from Y buss. |
| LZX | Load Z from X buss. |
| LZY | Load Z from Y buss. |
| BL | Left boolean box control field. |
| BR | Right boolean box control field. |
| VCY | Force 2ØØ nsec. cycle. |
| DGO | Deferred conditional branch. |
| TE1Y | Transfer E1 buss to Y buss. |
| TE2Y | Transfer E2 buss to Y buss. |

## A6 Macros to implement field logic

Bob Van Tuyl has written some macros designed to implement pseudo-QSPL type field operations. The way in which fields are defined and the operations one may do with them is described below. To use the package, the user should put the contents of KDF file (LEWENDAL)FIELD into his microprogram ahead of any code which uses field logic.

One may define a field by saying:

DF(name, displacement, first bit, last bit)

Following are the available field operations:

| Operation | Result |
|---|---|
| DISP(name) | Displacement of field. |
| MASK(name) | Mask of field. |
| NMASK(name) | Complement of mask of field. |
| SHFT(name) | Shift required to right-adjust field. |
| ONE(name) | Value of one in field. |
| LDCY(name) | Value of cycle to do on a load in order to right-adjust field. |
| STCY(name) | Value of cycle to do on a store to restore field from right-adjusted position to proper position in word. |
| STUFF(name) | M AND Q LCY LDCY(name)<br>Idea is to right adjust field. |
| NSTUFF(name) | M AND NOT Q LCY LDCY(name). |

Note that for STUFF and NSTUFF the value produced by LDCY must be Ø,1,2,3,4,8,12,16, or 2Ø.