INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
34100 TRIESTE (ITALY) - P.O.B. 586 - MIRAMARE - STRADA COSTIERA 11 - TELEPHONES: 224281/2/3/4/5/6
CABLE: CENTRATOM - TELEX 460392-1

SMR/90 - 2

COLLEGE ON MICROPROCESSORS:

TECHNOLOGY AND APPLICATIONS IN PHYSICS

7 September - 2 October 1981

MPL COOKBOOK - A guide to MPL

U. RAICH

CERN, Geneva
Switzerland

1. ABSTRACT

This guide was written for the Microprocessor College in Trieste 1981. It will give anybody interested an aid in getting acquainted with the Motorola MPL Cross-Compiler which is installed on the Trieste CDC machine. It is not intended to be a replacement of the Motorola MPL reference manual but rather to be an aid to make that manual more understandable. This is achieved mainly by means of sample programs. To discover all the possibilities of the compiler it is recommended to study the Motorola MPL reference manual. But if you never used this language before then you will hopefully find all the information needed to run successfully your first MPL program. The sample programs are very simple but they will show you how the compiler works.

2. INTRODUCTION

2.1 WHAT IS A MPL PROGRAM?

A MPL program consists of the following parts:

1. The statement: "PROCEDURE OPTIONS (MAIN)"
2. Declaration part starting with "DECLARE"
3. Program statements
4. "END" statement

Before explaining these parts in detail, let us first try to write a simple program. Notice that the above order of the four parts is maintained.

```
!
!   OUR FIRST PROGRAM:      PROGRAM ADD
!
PROCEDURE OPTIONS(MAIN)     !ALLOCATES STACK AND TEMPORARY STORAGE
DECLARE NUM1,NUM2,SUM       !RESERVES SPACE FOR VARIABLES
        SUM=NUM1+NUM2
$       SWI                 !RETURNS CONTROL TO MONITOR
$       FCB     0
        END
```

2.2 WHAT DOES THE MPL COMPILER MAKE OUT OF OUR PROGRAMS?

The compiler translates our programs into a sequence of assembler statements which can be passed to the Motorola assembler. Only the assembler produces an executable program. The work of the compiler can be seen if we refer to the listing produced from our small program (see listing 1). After the standard Motorola header we find our MPL statements preceeded by a "*". The translation is the assembler code directly following the MPL statement. Notice that the declaration part reserves space for the variables needed and that "PROCEDURE OPTION (MAIN)" automatically allocates an area for the stack and temporary storage (variable T).

2.3 THE CHARACTERS "!" AND "$"

These two characters have the following special meaning:

1. The "!" character is used to indicate a comment. All characters in a line following the "!" are interpreted as a comment and the MPL compiler ignores them. If the first character in a line is a "!" the whole line is considered to be a comment (see first lines in listing 1).

2. A "$" in the first position of a line indicates that the line is an assembler statement which must not be translated but will be passed to the assembler unaltered. This allows MPL and assembler statements to be mixed. Very time critical parts of a program may thus be written in assembler code while the rest of the program is written in MPL.

LISTING 1

Addition program

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*      MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***     COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !   OUR FIRST PROGRAM:     PROGRAM ADD
*00003 !
*00004 PROCEDURE OPTIONS(MAIN)   !ALLOCATES STACK AND TEMPORARY STORAGE
T       RMB   40
Z000    LDS   #T+39
*00005 DECLARE NUM1,NUM2,SUM     !RESERVES SPACE FOR VARIABLES
        JMP   Z001
NUM1    RMB   1
NUM2    RMB   1
SUM     RMB   1
*00006          SUM=NUM1+NUM2
Z001    LDAA NUM1
        ADDA NUM2
        STAA SUM
       SWI                      !RETURNS CONTROL TO MONITOR
       FCB      0
*00009          END
ZFFF    EQU   *
        END
```

3. DECLARATIONS

As you can see from listing 1 the compiler reserved 1 byte for each of the variables NUM1,NUM2,SUM. In this chapter we will see how to declare addresses, character strings, arrays etc. which cause more than 1 byte to be reserved.

The general form of a declaration is the following:

```
[level] name [(occurence)] [variable type] [(m)   ]
                                           [(m,n)]
[DEFINED name] [BASED] [INITIAL]
```

Here everything given in brackets is optional

3.1 SIMPLE VARIABLES (NOT ARRAYS)

Here we can forget about "level" and "occurence" since these values are only used in conjunction with declarations of data structures. DEFINED, BASED and INITIAL will be explained later, so we stay with a form:

```
DECLARE                          example: DECLARE
name1 [type] [(m)   ],                    NUM1 BINARY(1),
             [(m,n)]
name2 [type] [(m)   ],                    NUM2 BINARY(1),
             [(m,n)]
name3 [type.....                          PROD BINARY(2),

namen [type] [(m)   ]                     PPRINT DECIMAL(5,2)
             [(m,n)]
```

which is the most commonly used type of a declaration. Notice the comma after each declaration except the last one and don't forget them in your program!!!!

Table 1 gives you all the possible variable types and the number of bytes needed to store them.

variable type	parameters	space needed [bytes]
1. no specification	BINARY (1) assumed	1
2. BINARY (m)	m: number of bytes; m=1,2	1 if m=1 2 if m=2
3. BIT (m)	m: number of bits m=1-8	1
4. DECIMAL (m,n)	m: number of digits in total n: number of digits after decimal point m=1-12; n=1-12	1-12
5. CHARACTER (m)	m: number of ASCII characters m=1-256	1-256
6. LABEL	used in conjunction with computed GOTO statement only	

Table 1: Variable types

3.2 SYMBOLIC NAMES FOR VARIABLES AND LABELS

Symbolic names in MPL consist of up to 6 alphanumeric characters, the first one being alphabetic. You should try to use meaningful names because this makes your programs much more readable.

The following names must not be used because they are reserved keywords for either the MPL compiler or the assembler:

A	GE	OR
AND	GO	ORIGIN
B	GOTO	PROC
BASED	GT	PROCEDURE
BEGIN	IAND	RETURN
BIT	IEOR	SHIFT
BY	IF	SIGNED
CALL	INIT	T
DCL	INITIAL	THEN
DECLARE	IOR	TO
DEF	LABEL	WHILE
DEFINED	LE	X
DO	LT	Z000
ELSE	MAIN	Z001
END	NE	.
EOR	NOT	.
EQ	OPTIONS	ZFFF

All 4 character names starting with Z are not allowed.

Table 2: Reserved keywords for the compiler or assembler

3.3 THE USE OF "DEFINED", "BASED", "INITIAL"

1. "DEFINED" is used to redefine a variable. This we can only apply on the last variable declared on the same level (see also chapter on declarations of data structures). This statement should however be avoided since it makes the programs hard to debug.

2. "BASED" allows you to allocate storage space dynamically in a predefined large memory space. It is used only in conjunction with pointers. This rather advanced feature of MPL is not treated in this manual.

3. "INITIAL (value)" gives the variable the initial value specified in the bracket. In the case of arrays several initial values may be assigned. INITIAL is very frequently used.

Notice that the decimal variables are <u>not floating point numbers</u> that you can use for calculations but only ASCII character strings used for printing messages! So a multiplication of two decimal numbers will result in pure nonsense.

As a summary of this section we will improve our first program :

```
!
!PROGRAM TO SHOW DECLARATIONS OF SIMPLE VARIABLES
!
PROCEDURE OPTIONS(MAIN)
DECLARE
NUM1 BINARY(1) INITIAL(4),NUM2,SUM,
D CHARACTER(5)
$       PRINT EQU $1000
        D='SUM ='
        SUM=NUM1+NUM2
        CALL PRINT(D,SUM)
$       SWI
$       FCB     0
        END
```

This program adds 4 to the contents of NUM2 and stores the result into memory location SUM. The variable D is loaded with the character string: (delimited by quotes) 'SUM =' and the subroutine PRINT prints the result in the form: SUM =14 (if NUM2 was hex. 10). Notice how "INITIAL" has been translated.

3.4 CHARACTER STRINGS, ADDRESSES, HEX VALUES

Character strings and addresses are indicated by single quote signs. So D='SUM =' assigns the character string "SUM =" to the variable D (see listing 2). PTR='NUMBER' will load PTR with the address of NUMBER (PTR has to be declared as BINARY (2) of course). Double quotes are used to indicate hex numbers e.g. PTR="FFFF".

LISTING 2

Declaration of simple variables

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*      MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***     COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !PROGRAM TO SHOW DECLARATIONS OF SIMPLE VARIABLES
*00003 !
*00004 PROCEDURE OPTIONS(MAIN)
T       RMB  40
Z000    LDS  #T+39
*00005 DECLARE
        JMP  Z001
*00006 NUM1 BINARY(1) INITIAL(4),NUM2,SUM,
NUM1    FCB  4
NUM2    RMB  1
SUM     RMB  1
*00007 D CHARACTER(5)
D       RMB  5
Z001    EQU  *
        PRINT EQU #1000
*00009          D='SUM ='
        LDX  #Z42E
        STX  T
        LDX  #D
        LDAB #5
        JSR  ZF0A
*00010          SUM=NUM1+NUM2
        LDAA NUM1
        ADDA NUM2
        STAA SUM
*00011          CALL PRINT(D,SUM)
        JSR  PRINT
        BRA  Z002
        FDB  D
        FDB  SUM
Z002    EQU  *
        SWI
        FCB     0
*00014          END
Z42E    FCC  5,SUM =
        PAGE
ZF0A    STX  T+2  *** EQUAL SIZE MOVE ***
ZF0A1   LDX  T
        LDAA 0,X
        INX
```

```
        STX  T
        LDX  T+2
        STAA 0,X
        INX
        STX  T+2
        DECB
        BNE ZF0A1
        RTS
ZFFF    EQU  *
        END
```

3.5 DECLARATION OF DATA STRUCTURES

3.5.1 Vectors and Matrices

Let us first consider simple mathematical arrays. Here the declaration has exactly the same form as in the case of simple variables except that "occurence" has to be specified. "Occurence" gives you the number of elements you have in your data structure, e.g. VECTOR (3) has 3 elements, MATRIX (2,2) has 4 elements. The type of each element is specified in the same manner as for simple variables. As an example we rewrite our program to allow the addition of 2 (2,2) matrices in double precision (BINARY(2)). Note that the matrices are stored in the following order: ARR(1,1), ARR(1,2), ARR(2,1), ARR(2,2).

```
!
!PROGRAM TO ADD TWO 2X2 MATRICES
!
PROCEDURE OPTIONS(MAIN)
DECLARE
ARRA(2,2) BINARY(2) INITIAL("1FF","10","3541","0"),
ARRB(2,2) BINARY(2) INITIAL(7,8,9,10), !VALUES ARE DECIMAL
ARRC(2,2) BINARY(2)
        ARRC(1,1)=ARRA(1,1)+ARRB(1,1)
        ARRC(1,2)=ARRA(1,2)+ARRB(1,2)
        ARRC(2,1)=ARRA(2,1)+ARRB(2,1)
        ARRC(2,2)=ARRA(2,2)+ARRB(2,2)
$       SWI
$       FCB      0
        END
```


LISTING 3

Array declarations

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*     MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***     COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !PROGRAM TO ADD TWO 2X2 MATRICES
*00003 !
*00004 PROCEDURE OPTIONS(MAIN)
T       RMB  40
Z000    LDS  #T+39
*00005 DECLARE
        JMP  Z001
*00006 ARRA(2,2) BINARY(2) INITIAL("1FF","10","3541","0"),
ARRA    FDB  $1FF
        FDB  $10
        FDB  $3541
        FDB  $0
*00007 ARRB(2,2) BINARY(2) INITIAL(7,8,9,10), !VALUES ARE DECIMAL
ARRB    FDB  7
        FDB  8
        FDB  9
        FDB  10
*00008 ARRC(2,2) BINARY(2)
ARRC    RMB  8
*00009          ARRC(1,1)=ARRA(1,1)+ARRB(1,1)
Z001    LDAA ARRA+1
        LDAB ARRA
        ADDA ARRB+1
        ADCB ARRB
        STAB ARRC
        STAA ARRC+1
*00010          ARRC(1,2)=ARRA(1,2)+ARRB(1,2)
        LDAA ARRA+3
        LDAB ARRA+2
        ADDA ARRB+3
        ADCB ARRB+2
        STAB ARRC+2
        STAA ARRC+3
*00011          ARRC(2,1)=ARRA(2,1)+ARRB(2,1)
        LDAA ARRA+5
        LDAB ARRA+4
        ADDA ARRB+5
        ADCB ARRB+4
        STAB ARRC+4
        STAA ARRC+5
```

```
*00012          ARRC(2,2)=ARRA(2,2)+ARRB(2,2)
        LDAA ARRA+7
        LDAB ARRA+6
        ADDA ARRB+7
        ADCB ARRB+6
        STAB ARRC+6
        STAA ARRC+7
        SWI
        FCB      0
*00015          END
ZFFF    EQU  *
        END
```

3.5.2 General data structures

As you saw in the matrix addition program (listing 3) each element in the matrix was of the same type namely BINARY(2). Suppose you want to write a command string interpreter like the one that is implemented in the development station. Such a program may be efficiently written with the aid of tables of the following form:

Command string	Jump adress	Parameters needed	
HELP	ADDHLP	00000000B	row 1
DECO	ADDDEC	11000000B	row2
EPRO	ADDEPR	11000000B	.
CONV	ADDCON	00010000B	.
etc.			

Here the first column are ASCII strings, the second are addresses and the third of type BIT(n), where n is the number of parameters allowed. If a bit is set the corresponding parameter has to be given. To implement such a structure we must study the concepts of "levels". You can decompose your data stucture into several substructures, in about the same way as you decompose a big program into a main program, a subroutine, a sub-subroutine etc., and you declare the types of these substructures. Level 1 is the main routine in a big program by analogy, here it is the entire structure. Level 2 is the first level subroutine, here the rows. Level 3 is a subroutine called by a subroutine, here the individual elements. This means that you can assign a type to an individual element in a row using "level". It is not possible however to change the type within a column. (The second element in a row is always of type BINARY(2)). As you see our example is still a rather primitive one since in fact the elements of the rows could itself be a data structure. The declaration for the command string table (CSTRTB) of the example is done in listing 4 for 20 rows. Don't be puzzled that the compiler allocates 134 bytes to PAR . This is just the space needed for the whole structure when the number of bytes already allocated (4 for COMSTR and 2 for JPADDR) are substracted from 20*(4+2+1)=140.

LISTING 4

Declaration of general data structures

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*     MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***    COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !DECLARATION OF A COMMAND STRING TABLE
*00003 !
*00004 PROCEDURE OPTIONS(MAIN)
T       RMB  40
Z000    LDS  #T+39
*00005 DECLARE
        JMP  Z001
*00006 01 CSTRTB,
CSTRTB  EQU  *
*00007     02 ROW(20),
ROW     EQU  *
*00008       03 COMSTR CHAR(4),     !COMMAND STRING
COMSTR  RMB  4
*00009       03 JPADDR BINARY(2),   !JUMP ADDRESS
JPADDR  RMB  2
*00010       03 PAR    BIT(8)       !PARAMETERS TO BE GIVEN
*00011 !
*00012 !COMMAND STRING INTERPRETER
*00013 !
PAR     RMB  134
Z001    EQU  *
        SWI
        FCB     0
*00016          END
ZFFF    EQU  *
        END
```

4. ARITHMETIC AND LOGIC STATEMENTS

Since the handling of arithmetic and logic statements in MPL works like in any other high level language (like Fortran etc.) and the description of these statements in the MPL reference manual is rather understandable I will only give a list of all the possibilities and show their use by an other example.

-	unary -
SHIFT	arithmetic shift or rotate
IAND	logical AND
IOR	logical OR
EOR	logical exclusive OR
*	multiply
/	divide
+	add
-	subtract

Table 3: Arithmetic and logic statements

In the above table the order of priority is maintained, so that in an arithmetic expression the unary - is executed first and the subtraction last. The order of computation can be changed using brackets.
Only the SHIFT operation needs further explanation:
It has the form:

WORD SHIFT K

where WORD is a variable of type BINARY or BIT and -8<K<8 gives the number of bits that WORD will be shifted. If K is positiv it will be shifted left otherwise it will be shifted right. If WORD is of type BINARY the SHIFT will be translated into an arithmetic shift instruction if WORD is of type BIT the SHIFT will become a rotate instruction.

The logical expressions are:

EQ	equal
NE	not equal
GT	greater than
GE	greater or equal
LT	lower than
LE	lower or equal

Table 4: Logical expressions

These logical expressions are used in conjunction with control statements like "IF", "DO WHILE" etc. which are explained in the chapter on control statements. To see how one uses arithmetic and logic expressions, we will write a program that converts a binary number into ASCII characters which you can print on the terminal. Such a routine is of course also used in the ROSY monitor.

```
!
!CONVERSION BINARY-ASCII
!
PROCEDURE OPTIONS(MAIN)
DECLARE WORD,STR1,STR2
       STR1=WORD IAND "F0"       !MASK OFF LOW NIBBLE
       STR1=STR1 SHIFT -4        !SHIFT TO BIT1 0-3
       STR2=WORD IAND "0F"       !MASK OFF HIGH NIBBLE
!
!IF STR1 IS LOWER OR EQUAL 9 WE HAVE TO ADD "30" TO GET
!THE CORRECT ASCII VALUE
!IF IT IS A...F WE HAVE TO ADD "41".
!
IF STR1 LT 10 THEN STR1=STR1+"30" ELSE STR1=STR1+"41"
!
!THE SAME THING HAS TO BE DONE FOR STR2 OF COURSE
!
IF STR2 LT 10 THEN STR2=STR2+"30" ELSE STR2=STR2+"41"
$       SWI
$       FCB    0
        END
```


LISTING 5

Conversion binary-ASCII

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*     MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***    COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !CONVERSION BINARY-ASCII
*00003 !
*00004 PROCEDURE OPTIONS(MAIN)
T       RMB  40
Z000    LDS  #T+39
*00005 DECLARE WORD,STR1,STR2
       JMP  Z001
WORD    RMB  1
STR1    RMB  1
STR2    RMB  1
*00006        STR1=WORD IAND "F0"       !MASK OFF LOW NIBBLE
Z001    LDAA WORD
       ANDA #240
       STAA STR1
*00007        STR1=STR1 SHIFT -4        !SHIFT TO BIT1 0-3
       ASRA
       ASRA
       ASRA
       ASRA
       STAA STR1
*00008        STR2=WORD IAND "0F"       !MASK OFF HIGH NIBBLE
*00009 !
*00010 !IF STR1 IS LOWER OR EQUAL 9 WE HAVE TO ADD "30" TO GET
*00011 !THE CORRECT ASCII VALUE
*00012 !IF IT IS A...F WE HAVE TO ADD "41".
*00013 !
       LDAA WORD
       ANDA #15
       STAA STR2
*00014 IF STR1 LT 10 THEN STR1=STR1+"30" ELSE STR1=STR1+"41"
       LDAA STR1
       CMPA #10
       BGE  Z003
       ADDA #48
       STAA STR1
       BRA  Z004
*00015 !
*00016 !THE SAME THING HAS TO BE DONE FOR STR2 OF COURSE
*00017 !
Z003    LDAA STR1
```

```
        ADDA #65
        STAA STR1
*00018 IF STR2 LT 10 THEN STR2=STR2+"30" ELSE STR2=STR2+"41"
Z004    LDAA STR2
        CMPA #10
        BGE  Z006
        ADDA #48
        STAA STR2
        BRA  Z007
Z006    LDAA STR2
        ADDA #65
        STAA STR2
Z007    EQU  *
        SWI
        FCB   0
*00021          END
ZFFF    EQU  *
        END
```

5. THE "ORIGIN" STATEMENT

The ORIGIN statement enables you to load your program to any location in memory you like. You just write:ORIGIN "xxxx" ,where xxxx is a hexadecimal address and the whole program following the ORIGIN statement will be loaded to the address specified. The ORIGIN statement can occur anywhere in your program. Note however that your program will then not be relocatable any more. It is therefore recommended (especially for subroutines written in MPL) to specify the load address when running the pusher.

6. CONTROL STATEMENTS (GOTO IF DO)

Every high level language incorporates instructions which allow to change the order of execution of the statements. These correspond to JMP and BRA statements in the Motorola 6800 assembler.

6.1 LABELS

A feature which is peculiar to MPL is the fact that there exist different sorts of labels. One can put a label on an executable statement where the label name has to be followed by a colon

e.g.: LOOP: A=A+1

but one can also declare a variable of type "LABEL" in the declaration part and assign a value to it either by an INITIAL statement or by an assignment statement of the form:

VAR='LOOP'

where "LOOP" is a label on an executable statement. VAR can also be an array of type LABEL. (refer to listing 6 for example)

The control statements are thoroughly explained in the MPL reference manual so I will only give a compilation of the possibilities and an other sample program.

6.2 THE GOTO STATEMENT

There exist 3 different types of GOTO statements:

1. Unconditional GOTO:
   `GOTO LAB` — where LAB is a label on an executable statement
2. Assigned GOTO:
   `GOTO VAR` — where VAR is a variable of type LABEL
3. Computed GOTO:
   `GOTO (x1,x2,x3...xn),I`
   `GOTO VAR(I)`
   Here the program continues at label xi or VAR(I), depending on the value of I.

GOTO's to a label should only be used for constructing "missing" structured constructs.

6.3 THE IF AND DO STATEMENTS

A conditional branch instruction can be obtained with the statement:

```
IF L THEN S1 [ELSE S2]
```

where S1 and S2 are executable statements and L is a logical expression. S1 and S2 can also be blocks of statements grouped together using a DO statement of the form:

```
DO
 :
 :
END
```

The other forms of DO statements:

```
DO I= M1 TO M2 [BY M3]
DO WHILE L
```

or a mixture of these:

```
DO I=M1 TO M2 [BY M3] WHILE L
```

are used to program loops. For an example of the use of these control statements see listing 6 .

7. PROCEDURES

7.1 THE MAIN PROCEDURE

In every program written until now we used the statement: PROCEDURE OPTIONS(MAIN) which reserves space for temporary storage of variables and stack. The program itself immediately follows this scratchpad and the space needed for the variables. It is however often desirable to place the stack and the scratchpad into RAM whereas the program resides in EPROM. This can be done writing:

```
PROCEDURE OPTIONS(MAIN,STACKNAME)
```

If a stackname is given the programmer has to reserve workspace using the variable T and he has to allocate the stack. In this case DECLARE has to precede the PROCEDURE statement because STACK has to be declared before it is used. See in listing 6 how this works.

7.2 SUBROUTINE PROCEDURE

7.2.1 Setting up and calling subroutines

It is a very good practice to write programs in blocks or modules where each module performs a certain task. This is usually done by means of subroutines. In listing 2 such a scheme was already used: We called the routine PRINT to output the result of the addition program. The character string "SUM =" and the result SUM was transferred to the subroutine. The general form of a subroutine call is:

```
CALL SUBNAM(X1,X2,X3....Xn)
CALL SUBNAM<X1,X2,X3>
```

or

where X1....Xn are parameters to be transferred. To set up a subroutine we write:

```
SUBNAM:   PROCEDURE(X1,X2,X3.....Xn)
SUBNAM:   PROCEDURE<X1,X2,X3>
```

in our example:

```
PRINT:    PROCEDURE(CHAR,RESULT)
```

(refer to the section on parameter transfers below for more details).
Every MPL subroutine must contain the instruction "RETURN" to return control to the calling program.

You can either compile the subroutines together with the main program where you just write one block after the other and then you compile everything in one go, or you can compile subroutines independently in which case the temporary storage variable T has to be allocated using: $T RMB 40 or DECLARE T(40) and you have to tell the main program where it can find the subroutine via an EQU statement or via an external reference when using relocatable code (see listing 6). In the case where all programs are compiled together the storage space for the variables is common to all programs which means that a variable name declared in one program may not be used for an other variable in an other program. On the other hand parameter transfers are not necessary.

7.2.2 Parameter transfers

There are 3 different methods to transfer parameters to subroutines:

1. Transfer using MPU registers: CALL SUBNAM<X1,X2,X3>

   In this case only 3 parameters can be transferred (X1 and X2 in the accumulators A and B, and X3 in the index register X). X1 and X2 must be 1 byte variables and X3 must be of type BINARY(2). Null parameters are allowed. So you may write: CALL READPR<,,PRESET> if you only want to transfer the 2 byte variable PRESET.

2. Transfer with parameter list: CALL SUBNAM(X1,X2,....Xn)

   The compiler translates this into a JSR followed by the parameters.Of course a jump to the first executable assembler statement has to precede the parameter list (refer to listing 6 to see how the compiler manages this).

3. Transfer via external references

   This is not provided in the compiler itself but it can be done using the XDEF and XREF assembler directives. Note however that you can only transfer parameters from a MPL program to an assembler program in this way. (XDEF must be provided in the MPL program). Since every variable used in a MPL program must be defined by a DECLARE and this automatically assigns memory space to it, you cannot use variables defined by the XREF directive in MPL subroutines.

7.2.3 Linkage of MPL programs

The MPL Cross compiler does not support the linkage facility! It is however possible to link MPL program using the XDEF and XREF assembler directive.

Program 6 shows you how to use control statements and subroutines.

Suppose an experiment uses 2 TDCs and 2 ADCs and a data word consists of 6 bits of data and 2 address bits which identify the instrument like this:

bit nr.	7	6	5	4	3	2	1	0
	i1	i0	d5	d4	d3	d2	d1	d0

Then the source listing of a program to fill a spectrum for each of the four instruments might look as follows:

```
!
!READOUT PROGRAM FOR A LITTLE EXPERIMENT
!SHOWS THE USE OF CONTROL STATEMENTS AND SUBROUTINES
!INCLUDING PARAMETER TRANSFER
!
DECLARE
TYPE(4) LABEL INITIAL(TDCA,TDCB,ADCA,ADCB),
TDC1(64),TDC2(64),ADC1(64),ADC2(64),
PRESET BINARY(2),I,INSTR,DATA,T(40),STACK
!
PROCEDURE OPTIONS(MAIN,STACK)
!
$      XREF  GETDAT
$      XDEF  DATA
$READPR EQU $3000
$INITIA EQU $3100
$      XREF   CLEAR
$      XREF   DSPLAY
!
!CLEAR SPECTRA
!
DO I=1 TO 64
   TDC1(I)=0
   TDC2(I)=0
   ADC1(I)=0
   ADC2(I)=0
END
!
!INITIALIZE EXPERIMENT
!
```

```
CALL INITIA
CALL READPR<,,PRESET>       !READ PRESET FROM TERMINAL
!
!GETDAT PUTS DATA INTO THE MEMORY LOCATION RESERVED FOR DATA BY THE
!MPL PROGRAM. THE LOCATION IS KNOWN BECAUSE OF THE XDEF STATEMENT.
!
CALL GETDAT
DO WHILE TDC1(DATA) LT PRESET
   INSTR=DATA IAND "C0"          !MASK OFF DATA BITS AND CALCULATE
   INSTR=INSTR SHIFT-6           !INSTRUMENT IDENTIFICATION
   INSTR=INSTR+1
   DATA=DATA IAND "3F"
!
!  NEXT WE CASE ON THE INSTRUMENT IDENTIFICATION BITS
!  TO JUMP TO THE PART OF THE PROGRAM WHERE THE CORRESPONDING
!  SPECTRUM IS FILLED
!
   GOTO TYPE(INSTR)
      TDCA: TDC1(DATA)=TDC1(DATA)+1
            GOTO ENDCAS
      TDCB: TDC2(DATA)=TDC2(DATA)+1
            GOTO ENDCAS
      ADCA: ADC1(DATA)=ADC1(DATA)+1
            GOTO ENDCAS
      ADCB: ADC2(DATA)=ADC2(DATA)+1
   ENDCAS: CALL CLEAR              !CLEARS ALL INSTRUMENTS
END
CALL DSPLY(TDC1,TDC2,ADC1,ADC2)
END
```


LISTING 6

Control statements and subroutines

```
   NAM  MPLR10
*MOTOROLA SPD, INC. OWNS AND IS RESPONSIBLE FOR M68MPL
*       COPYRIGHT 1975 AND 1976 BY MOTOROLA INC.
*
*     MOTOROLA'S M6800 MPL COMPILER, RELEASE 1.2
*
*
***    COMPILED WITH MPL VERSION 1.2
*00001 !
*00002 !READOUT PROGRAM FOR A LITTLE EXPERIMENT
*00003 !SHOWS THE USE OF CONTROL STATEMENTS AND SUBROUTINES
*00004 !INCLUDING PARAMETER TRANSFER
*00005 !
*00006 DECLARE
*00007 TYPE(4) LABEL INITIAL(TDCA,TDCB,ADCA,ADCB),
TYPE    FDB  TDCA
        FDB  TDCB
        FDB  ADCA
        FDB  ADCB
TDC1    RMB  64
TDC2    RMB  64
ADC1    RMB  64
*00008 TDC1(64),TDC2(64),ADC1(64),ADC2(64),
ADC2    RMB  64
PRESET RMB  2
I       RMB  1
INSTR   RMB  1
DATA    RMB  1
T       RMB  40
*00009 PRESET BINARY(2),I,INSTR,DATA,T(40),STACK
*00010 !
STACK   RMB  1
*00011 PROCEDURE OPTIONS(MAIN,STACK)
Z000    LDS  #STACK
*00012 !
     XREF  GETDAT
     XDEF  DATA
READPR EQU $3000
INITIA EQU $3100
     XREF   CLEAR
     XREF   DSPLAY
*00019 !
*00020 !CLEAR SPECTRA
*00021 !
*00022 DO$I=1 TO 64
       LDAA #1
Z001   STAA I
*00023    TDC1(I)=0
       LDX  #TDC1
```

```
        LDAB I
        JSR  ZF1F
        CLR  0,X
*00024     TDC2(I)=0
        LDX  #TDC2
        LDAB I
        JSR  ZF1F
        CLR  0,X
*00025     ADC1(I)=0
        LDX  #ADC1
        LDAB I
        JSR  ZF1F
        CLR  0,X
*00026     ADC2(I)=0
        LDX  #ADC2
        LDAB I
        JSR  ZF1F
        CLR  0,X
        LDAA I
        CMPA #64
        BCC  Z002
        INCA
        JMP  Z001
*00027 END
*00028 !
*00029 !INITIALIZE EXPERIMENT
*00030 !
*00031 CALL INITIA
Z002    JSR  INITIA
*00032 CALL READPR<,,PRESET>       !READ PRESET FROM TERMINAL
        LDX  PRESET
*00033 !
*00034 !GETDAT PUTS DATA INTO THE MEMORY LOCATION RESERVED FOR DATA BY THE
*00035 !MPL PROGRAM. THE LOCATION IS KNOWN BECAUSE OF THE XDEF STATEMENT.
*00036 !
        JSR  READPR
*00037 CALL GETDAT
        JSR  GETDAT
*00038 DO WHILE TDC1(DATA) LT PRESET
Z003    LDX  #TDC1
        LDAB DATA
        JSR  ZF1F
        LDAA 0,X
        CMPA PRESET+1
        BGE  Z005
*00039     INSTR=DATA IAND "C0"          !MASK OFF DATA BITS AND CALCULATE
        LDAA DATA
        ANDA #192
        STAA INSTR
*00040     INSTR=INSTR SHIFT-6           !INSTRUMENT IDENTIFICATION
        ASRA
        ASRA
        ASRA
        ASRA
```



```
        ASRA
        ASRA
        STAA INSTR
*00041     INSTR=INSTR+1
        INC  INSTR
*00042     DATA=DATA IAND "3F"
*00043 !
*00044 !  NEXT WE CASE ON THE INSTRUMENT IDENTIFICATION BITS
*00045 !  TO JUMP TO THE PART OF THE PROGRAM WHERE THE CORRESPONDING
*00046 !  SPECTRUM IS FILLED
*00047 !
        LDAA DATA
        ANDA #63
        STAA DATA
*00048     GOTO TYPE(INSTR)
        LDX  #TYPE
        JSR  ZF04
        BRA  *+5
        FCB  2
        FDB  INSTR
        LDX  0,X
        JMP  0,X
*00049       TDCA: TDC1(DATA)=TDC1(DATA)+1
TDCA    LDX  #TDC1
        LDAB DATA
        JSR  ZF1F
        INC  0,X
*00050                GOTO ENDCAS
        JMP  ENDCAS
*00051       TDCB: TDC2(DATA)=TDC2(DATA)+1
TDCB    LDX  #TDC2
        LDAB DATA
        JSR  ZF1F
        INC  0,X
*00052                GOTO ENDCAS
        JMP  ENDCAS
*00053       ADCA: ADC1(DATA)=ADC1(DATA)+1
ADCA    LDX  #ADC1
        LDAB DATA
        JSR  ZF1F
        INC  0,X
*00054                GOTO ENDCAS
        JMP  ENDCAS
*00055       ADCB: ADC2(DATA)=ADC2(DATA)+1
ADCB    LDX  #ADC2
        LDAB DATA
        JSR  ZF1F
        INC  0,X
*00056    ENDCAS: CALL CLEAR             !CLEARS ALL INSTRUMENTS
ENDCAS  JSR  CLEAR
        JMP  Z003
*00057 END
*00058 CALL DSPLY(TDC1,TDC2,ADC1,ADC2)
Z005    JSR  DSPLY
```

```
        BRA   Z006
        FDB   TDC1
        FDB   TDC2
        FDB   ADC1
        FDB   ADC2
*00059  END
Z006    EQU   *
        PAGE
ZF04    STX   T+6   *** COMPUTE SUBSCRIPTS
        PSHA
        PSHB
        TSX
        LDX   2,X
        LDAA  1,X
ZF041   PSHA
        STX   T+2
        LDAB  2,X
        LDX   3,X
        LDAA  0,X
        BNE   *+7
        SBA
        LDAB  #$FF
        BRA   *+6
        DECA
        JSR   ZF44
        ADDA  T+7
        ADCB  T+6
        STAB  T+6
        STAA  T+7
ZF042   LDX   T+2
        INX
        INX
        INX
        PULA
        DECA
        DECA
        DECA
        BNE   ZF041
        LDX   T+6
        PULB
        PULA
        RTS
ZF44    PSHB        UNSIGNED SUBSCRIPT MULT
        LDAB  #8
        PSHB
        TSX
        CLRB
        RORA
ZF441   BCC   *+4
        ADDB  1,X
        RORB
        RORA
        DEC   0,X
        BNE   ZF441
```



```
        INS
        INS
        RTS
        SPC   2
ZF1F    BNE   ZF1F2 *** ADD TO INDEX ***
        DEX
        RTS
ZF1F2   DECB
ZF5F    STX   T+6
        ADDB  T+7
        STAB  T+7
        BCC   ZF1F1
        INC   T+6
ZF1F1   LDX   T+6
        RTS
ZFFF    EQU   *
        END
```

8. POINTERS

Pointers are used when handling arrays where the array elements themselves are data structures. Let us take the declaration of the command string interpreter in the chapter on array declarations: The elements of the command string table are the rows and each row consists of 3 elements. Pointers allow you to address the elements of the rows. Pointers are also useful when handling linked lists. Since these are however rather advanced programming techniques which we will not use in our exercises we will skip this chapter completely.

9. HOW TO USE THE MPL COMPILER ON THE TRIESTE CDC

Before running a MPL program, several translations have to be performed. First you have to compile the MPL source program using the MPL compiler. This produces a file suitable as input file to the assembler. You then assemble this file to get a Cufom object file. This Cufom file you may link to other Cufom files or libraries to get a new Cufom file, which you finally translate to Motorola S-format suitable for down-line loading. This final translation is done by the pusher. Of course you have to get the MPL compiler, the MPL library, the assembler, the linker and the pusher into your local file base before using these programs.

If you want to compile only, you have to type:

```
GET,M68MPL,MPLLIB/UN=MPTOOLS        (gets the MPL compiler
                                     (M68MPL) and the MPL
                                     library (MPLLIB) into
                                     your local file base)

M68MPL,MPLSRC,ASMSRC,MPLLIB         (starts compilation)
```

where MPLSRC: Name of MPL source program
ASMSRC: Name of output file from the MPL compiler

(these two names have to be given)

This sequence of commands will translate your MPL source file into an assembler source file and print error messages if you used incorrect MPL statements. If no errors are found you may proceed with assembling ,linking etc.

Since it is quite a complicated story to do all these translations, a system program was written to perform all the steps needed.
To use this program you simply type:

```
GET,PROCFIL/UN=MPTOOLS                (gets the program
                                      "PROCFIL")
BEGIN,MICMPL,,S=MPLSRC,[L=LNAM,B=BNAM,ALIB=MACLIB,CLIB=CUFLIB]
```

where

MPLSRC: Name of MPL source program

LNAM: Name of file where the assembler listing and the listings of the linker and of the pusher will be stored. If ommited the file is called: LISTING.

BNAM: Name of output file from the pusher. This file will be down-line loaded and executed. If ommited the file is called: LOADMOD.

MACLIB: Name of assembly language source file to be inserted into the compiled MPL program. MACLIB has to be specified only if a statement:
$ INSERT MACLIB
is used in the MPL code (rather advanced feature normally used to insert macro libraries)

CUFLIB: Name of Cufom library to be linked to the MPL program.

Everything given in brackets is optional.

This sequence of commands will get the necessary system programs into your local file base and it will then compile, assemble, link and push your program. So you get a file (BINNAM) ready for down-line loading after execution of these commands.

CONTENTS

LIST OF LISTINGS

LIST OF TABLES