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 - I

H4.SMR/159 -05

THIRD COLLEGE ON MICROPROCESSORS:
TECHNOLOGY AND APPLICATIONS IN PHYSICS

7 October - 1 November 1985

ASSEMBLER LANGUAGE PROGRAMMING FOR THE MOTOROLA MC 6809

VON EICKEN H.

DD Division
CERN
Geneva 23
Switzerland

MC 6809 Register Package:

8 bits	accum. A
8 bits	accum. B
16 bits	index register X
16 bits	index register Y
16 bits	user stack ptr. U
16 bits	hardware stack ptr. S
16 bits	program counter PC
8 bits	direct page reg. DP
8 bits	cond. code reg. CC

acc. A	acc. B	combined accum. D

MC 6809 Condition Code Register:

7	6	5	4	3	2	1	0	bit No.
E	F	H	I	N	Z	V	C	flags

	0	1	
C:	no	with	carry or borrow
V:	no	with	magnitude overflow
Z:	not	is	zero
N:	not	is	negative
I:	enable	disable	normal interrupts
H:	no	with	half carry
F:	enable	disable	fast interrupts
E:	part	full	register stacking

First programming problem:

Move contents of memory location $0410
into memory location $0411

* 8-bit data transfer
* use accumulator A

Memory contents before execution

loc:	040F	??
	0410	6A
	0411	12

Instructions:

```
LDA   $0410   load data
STA   $0411   transfer to new loc.
```

Memory contents after execution:

loc:	040F	??
	0410	6A
	0411	6A

First programming problem:

* Memory contains data
* Memory contains program to be executed

Let's translate our program:

```
LDA $0410    we need extended addressing
-->     B6    04    10     (three bytes)
STA $0411    we need extended addressing
-->     B7    04    11     (three bytes)
```

Now let's load it into memory:

loc:	03FF	??
	0400	B6
	0401	04
	0402	10
	0403	B7
	0404	04
	0405	11

Second programming problem:

Add the first 15 integers

Constraints:

* program starts at loc. $0400
* result stored into loc. $0420

Possible programming solutions:
(expressed in a Pascal like syntax)

```
VAR
 Count  : INTEGER;   to count repetitions
 EndVal : INTEGER;   to end repetitions
 Sum    : INTEGER;   to calculate the sum
```

--> WHILE <condition> DO <body>

```
Sum:= 0; Count:= 0;
WHILE Count < EndVal DO BEGIN
  Count:= Count + 1;
  Sum  := Sum + Count;
END;
```

--> FOR <iterative condition> DO <body>

```
Sum:= 0;
FOR Count:= 1 TO EndVal DO
  Sum:= Sum + Count;
```


Second programming problem:

--> REPEAT <body> UNTIL <condition>

```
Sum:= 0; Count:= EndVal;
REPEAT
  Sum  := Sum + Count;
  Count:= Count - 1;
UNTIL Count = 0;
```

--> let's program this for MC 6809

Data:

```
Sum     -->  loc. $0420
EndVal  -->  loc. $0421
Count   -->  loc. $0422
```

Program:

```
      CLR   $0420      Sum:= 0;
      LDA   $0421      Count:= EndVal;
      STA   $0422
Loop  LDA   $0422      Sum:= Sum+Count;
      ADDA  $0420
      STA   $0420
      LDA   $0422      Count:= Count-1;
      DECA
      STA   $0422
      BNE   Loop       UNTIL Count = 0;
```

Second programming problem:

--> Now let's translate it:

loc.	contents	opcode	addressing mode
0400	7F 04 20	CLR	extended
0403	B6 04 21	LDA	extended
0406	B7 04 22	STA	extended
0409	B6 04 22	LDA	extended
040C	BB 04 20	ADDA	extended
040F	B7 04 20	STA	extended
0412	B6 04 22	LDA	extended
0415	4A	DECA	inherent
0416	B7 04 22	STA	extended

* we have to compute the relative distance

  hence: $041B - $0409 --> $0012

* we branch backwards

  hence: - $0012 --> $FFEE

* signed short or long branch ?

15 … 8	7	6 … 0	bit No.
xxxx xxxx	x	<dist.>	

041A	26 EE	BNE	relative

Second programming problem:

Handcoding or assembly language?

Are there any problems with handcoding?

* easy to make mistakes, nobody checks!
* address calculation/allocation tedious
* just imagine you want to change.....

What is Assembly Language?

* easier to create and modify a program
* easier to read and understand than handcode
* symbolic machine instructions (opcodes)
* symbolic identifiers for registers
* symbolic address identifiers (labels)
* assembler directives for extra services

Flex MC 6809 Mnemonic Assembler:

The assembler is a computer program:

- # reading lines containing the source language of a MC 6809 assembler program
- # translating source language into machine language for the MC 6809 microprocessor
- # producing an annotated listing of the program
- # producing a symbol table output

```
                  +-----------+   -> machine   code
                  |  MC 6809  |
source code ->    |           |   -> listing
                  | assembler |   -> symbol table
                  +-----------+
```

From where does it read?

Where does it store the information?

Assembler source line format:

Each source line may have up to 4 fields:

<label> <operation> <operand> <comment>

Label field:

starts in column 1, if with:

'*' (star) : --> comment line

' ' (space): --> label field is empty

identifier : --> label symbol

--> ERROR

Operation field:

follows label field with 1 blank minimum

opcode (machine instruction mnemonic)

pseudo (assembler directive)

macro call (evaluated body inserted here)

[illegible]

Operand field:

after operation field with 1 blank minimum

contents depend on operation mnemonic

* nothing
* register specifications
  (address modes)
* numeric constants, symbols,
  expressions

Comment field:

after operand field with 1 blank minimum

* optional but essential!!!
* comment flow of program
* opcode explanation is useless...

Assembly language elements:

Identifier:

* composed with:

  letters A-Z and a-z
  numbers 0-9
  underscore

* first character must be a letter
* upper/lower case letters are distinct
* first six characters are significant
* reserved identifiers:

"A", "B", "D"	- accumulators
"X", "Y"	- index registers
"U", "S"	- stack pointers
"DP"	- direct page reg.
"CC"	- condition code reg.
"PC", "PCR", "*"	- program counter

Assembly language elements:

Constants:

* decimal constant, (0 - 9)
  -32768 <= N <= 32767
* hexadecimal constant, (0 - 9), (A - F)
  0 <= N <= $FFFF
* octal constant (0 - 7)
  0 <= N <= @177777
* binary constant (0 - 1)
  0 <= N <= %1111111111111111
* character constant, ASCII character
  prefixed with "' " (apostrophe)

Opcodes:

* mnemonic for machine instruction
  (CLR, DEC, etc.)
* the [illegible] between
  opcodes and machine instructions
* assembly [illegible] opcode and it's
  operands

Assembly language elements:

Assembler directives: (or: pseudo codes)

are instructions to the assembler!

* module identification
  (NAM, END)
* origin control
  (ORG)
* listing control
  (ERR, OPT, PAG, SPC, TTL, STTL)
* data generation and storage reservation
  (FCB, FDB, FCC, RMB)
* symbol definition
  (EQU, REG, SET)
* object code control
  (RPT, SETDP)
* conditional assembly
  (IF, IFN, IFC, IFNC, ELSE, ENDIF)
* macro definition
  (MACRO, DUP, ENDD, EXITM, ENDM)

SEE: chapter V of TSC 6809 Assembler
manual for more details ...

Assembly language elements:

Expressions:

* an expression is a combination of:

symbols	constants
operators	parentheses

* arithmetic operators:

+	unary or binary addition
-	unary or binary subtraction
*	multiplication
	division

* logical operators:

&	logical AND
	logical OR
!	logical NOT

* shift operators:

>>	shift right
<<	shift left

Assembly language elements:

Expressions:

* relational operators:

=	equal
<	less than
>	greater than
<>	not equal
<=	less than or equal
>=	greater than or equal

* operator precedence:

parenthesized expressions
unary + and -
shift operators
multiply and divide
binary add and subtract
relational operators
logical NOT
logical AND and OR

MC 6809 addressing modes:

operand format	MC 6809 addressing mode
no operand	accumulator and inherent
<expression>	direct, extended, relative
<<expression>	forced direct
> <expression>	forced extended
[<expression>]	extended indirect
<expression>,R	indexed
<<expression>,R	8-bit offset indexed
> <expression>,R	16-bit offset indexed
<accumulator>,Q	accum. offset indexed
[<expression>,R]	indexed indirect
<[<expression>,R]	same with. 8-bit offset
>[<expression>,R]	same with 16-bit offset
[<accumulator>,Q]	same with accum. offset
Q+	auto increment by 1
Q++	auto increment by 2
[Q++]	auto increment indirect
-Q	auto decrement by 1
--Q	auto decrement by 2
[--Q]	auto decrement indirect
# <expression>	immediate
<register list>	immediate

with: R = S U X Y PC PCR
and : Q = S U X Y

Second programming problem:

Program our problem in assembly language:

```
        NAM   Add first 15 integers

* Define begin of program area:

        ORG   $400
Start   CLR   Sum         Sum:= 0;
        LDA   EndVal      initialize count
        STA   Count
Loop    LDA   Sum         accumulated sum
        ADDA  Count       add loop count
        STA   Sum         save it
        LDA   Count       re-load count
        DECA              next iteration
        STA   Count       save it
        BNE   Loop        repeat until zero
        ???               and now what?

* Define begin of data area:

        ORG   $420
Sum     RMB   1           to contain sum
EndVal  FCB   15          initial loop value
Count   RMB   1           to contain count

        END
```

[illegible]

[illegible]

```
        NAM    Add first 15 integers
```

[illegible] request

[illegible]

[illegible]

```
        ORG    [illegible]
[illegible] CLR    Sum            [illegible]
        LDA    EndVal         initialize count
Loop    STA    Count          save current count
        ADDA   Sum            accumulated sum
        STA    Sum            save it
        LDA    Count          re-load count
        DECA                  next iteration
        BNE    Loop           repeat until zero
        SWI                   monitor call
        FCB    Rosy           return to Rosy

* Define begin of data area:

        ORG    $420
Sum     RMB    1              to contain sum
EndVal  FCB    15             initial loop value
Count   RMB    1              to contain count

        END
```


[illegible]

Initialize an array to zero

Constraints:

- program starts at loc $0200
- array "Bin" 5 bytes long, starts at loc $00[illegible]

Let's do it the hard way:

```
        NAM    Initialize an array

        ORG    $10            data area
Bin     RMB    5              reserve 5 bytes

        ORG    $200           program area
Init    CLRA                  zero array "Bin"
        STA    Bin            Bin(1)
        STA    Bin+1          Bin(2)
        STA    Bin+2          Bin(3)
        STA    Bin+3          Bin(4)
        STA    Bin+4          Bin(5)
        SWI                   return to Rosy
        FCB    0

        END    Init
```

Third programming problem:

Now let's use indexed addressing

Structure:

```
-->     FOR <iterative condition> DO <body>

        FOR Index:= 1 TO 5 DO
         Bin [Index] := 0;

-->
        NAM    Initialize an array

        ORG    $10           data area

Bin     RMB    5             reserve 5 bytes

        ORG    $200          program area

Init    CLRA                 zero array "Bin"
        LDX    #0            our index
Loop    STA    Bin,X         zero next element
        LEAX   1,X           step index
        CMPX   #5            all done?
        BNE    Loop          if not
        SWI                  else --> Rosy
        FCB    0

        END    Init
```


Third programming problem:

Structure:

```
-->     FOR <iterative condition> DO <body>

        FOR Index:= 5 DOWNTO 1 DO
         Bin [Index] := 0;

-->
        NAM    Initialize an array

        ORG    $10           data area
Bin     RMB    5             reserve 5 bytes
        ORG    $200          program area

Init    CLRA                 zero array "Bin"
        LDX    #5            largest index
Loop    STA    Bin-1,X       zero next element
        LEAX   -1,X          step index
        BNE    Loop          if not done
        SWI                  else --> Rosy
        FCB    0

        END    Init
```

MC 6809 effective addressing modes:

--> effective address

is the address ultimately used by the MC 6809 microprocessor for it's operation

we use <ea> as abbreviation

--> inherent addressing

operation code specifies address

```
DECA          --> 0400  4A
```

effective address for DECA is accumulator A

[illegible]

[illegible]

code

```
LDX  #0       --> 0400  8E
                  0401  00   <-- <ea>
                  0402  00
```

effective address is in program counter once instruction has been decoded

MC 6809 effective addressing modes:

--> extended addressing

address of data follows operation code

```
LDA  $0421  --> 0400  B6
                0401  04
                0402  21
```

effective address is contained in the two bytes following op. code

--> direct addressing

address of data is computed using:

contents of byte after opcode

contents of direct page register

```
STA  $0010  --> 0400  97
                0401  10
```

assume direct page register contains 05

<ea> --> 05 10

contents of accum. A is stored into memory address 0510

MC 6809 effective addressing modes:

--> indexed addressing

base register	-->	register to be used for <ea> calculation
offset	-->	constant value or contents of accum. to be used for <ea> calculation
post byte	-->	byte following opcode defines the indexed addressing mode

* the effective address is calculated using the contents of base register and the offset (if specified)

* for indirect indexed addressing mode the effective address is used to read the word at the memory location it is pointing to and use it's contents as final effective address

MC 6809 effective addressing modes:

post byte:								addressing mode:
0	R	R	n	n	n	n	n	5-bit offset
1	R	R	0	0	0	0	0	autoincrement by 1
1	R	R	I	0	0	0	1	autoincrement by 2
1	R	R	0	0	0	1	0	autodecrement by 1
1	R	R	I	0	0	1	1	autodecrement by 2
1	R	R	I	0	1	0	0	zero offset
1	R	R	I	0	1	0	1	accumulator B offset
1	R	R	I	0	1	1	0	accumulator A offset
1	R	R	I	1	0	0	0	8-bit offset
1	R	R	I	1	0	0	1	16-bit offset
1	R	R	I	1	0	1	1	accumulator D offset
1	x	x	I	1	1	0	0	PC with 8-bit offset
1	x	x	I	1	1	0	1	PC with 16-bit offset
1	x	x	1	1	1	1	1	extended indirect

address mode field

indirect field with
0 for direct mode
1 for indirect mode

register field with
00 for X register
01 for Y register
10 for U register
11 for S register
xx don't care

Indexed addressing:

--> constant offset from base register

```
STA   Bin,X       0400   A7
                  0401   88 (1 00 0 1000)
                  [illegible]
```

[illegible]

loc. 0402

or:

```
STA   Bin-1,X     0400   A7
                  0401   0F (0 00 0 1111)
```

<ea> -->   contents of register X
+ signed 5-bit offset in post byte

or:

```
LEAX 1,X          0400   30
                  0401   01 (0 00 0 0001)
```

<ea> -->   contents of register X
+ signed 5-bit offset in post byte

Indexed addressing:

--> accumulator offset from base register

* the effective address is calculated by adding the signed contents of accumulator A, B or D to the contents of the base register X, Y, S or U

* this addressing mode is very useful to handle lookup tables

Example:

Assume we are reading a character from a terminal in ASCII format and have to convert it for further processing.

We use the value of the ASCII character as an index into a table containing the conversion value:

```
LDX   #Table     code table base address
LDB   CHAR       ASCII character code
LDA   B,X        corresponding code
```

Note: ASCII begins with code "00"!

MC 6809 effective addressing modes:

--> program counter relative addressing

* this address mode is implied for conditional branch instructions:

  we are branching to an instruction that is "some locations" away from where we are

  the branch instruction contains a constant signed displacement that is added to the program counter when the branch is taken

* following the same principle there is another indexed addressing mode:

  " constant offset
  from
  program counter "

MC 6809 effective addressing modes:

Indexed addressing:

--> constant offset from program counter

* the effective address is calculated by adding the signed offset to the contents of the program counter

* the offset is either an 8-bit or a 16-bit offset, there is no special 5-bit or accumulator offset

* the assembler will calculate the offset if we designate the address to be "program counter relative" using "PCR":

```
LEAX  L-Value,PCR
```

* this addressing mode is very useful to write position-independent code.

MC 6809 effective addressing modes:

Indexed addressing:

--> autodecrement / autoincrement

```
STD   --S        0400   ED
                 0401   E3 (1 11 0 0011)
```

<ea> -->   contents of register S
           decremented by two

assume: accumulator D contains: 0102
stack pointer S contains: 0500
location 0500 contains: EF

before:

loc:		
0500	EF	<-- stack pointer S
04FF	??	
04FE	??	

after:

loc:		
0500	EF	
04FF	02	
04FE	01	<-- stack pointer S

MC 6809 effective addressing modes:

Indexed addressing:

--> autodecrement / autoincrement

```
LDD   S++        0400   EC
                 0401   E1 (1 11 0 0001)
```

<ea> -->   contents of register S
           incremented by two

assume: accumulator D contains: 0000
stack pointer S contains: 04FE

before:

loc:		
0500	EF	
04FF	02	
04FE	01	<-- stack pointer S

after:

loc:		
0500	EF	<-- stack pointer S
04FF	??	
04FE	??	and accum. D --> 0102

Fourth programming problem:

Write a routine adding all positive integers of a given interval (allow 16-bit integers)

Constraints:

* write it as a subroutine
* program starts at loc. $0400
* parameters are at loc. $0600

loc.	contents		meaning
0600	IntBeg	-->	start of interval
0602	IntEnd	-->	end of interval
0604	SumInt	-->	resulting sum

Convention:

accumulators A and B are free for use

Structure:

```
-->    FOR <iterative condition> DO <body>

       SumInt:= 0;
       FOR Count:= IntBeg TO IntEnd DO
        SumInt:= SumInt + Count;
```


Fourth programming problem:

Add "N" consecutive integers:

```
        TTL   Subroutine AddInt
        STTL  Add "N" consecutive integers

        ORG   $400        program area

AddInt  CLR   SumInt      SumInt:= 0;
        CLR   SumInt+1
        LDD   IntBeg      start of interval
AddLoop STD   --S         save current count
        ADDD  SumInt      accumulated sum
        STD   SumInt      save it
        LDD   S++         re-load count
        ADDD  #1          step to next value
        CMPD  IntEnd      over interval end?
        BLS   AddLoop     no, continue
        RTS               yes, simply return

        ORG   $600        parameter area

IntBeg  RMB   2           begin of interval
IntEnd  RMB   2           end of interval
SumInt  RMB   2           sum of integers
```

***** There is at least one error! *****

[illegible]

[illegible]

--> first error:

we do not verify the interval,
we simply assume:

IntBeg <= IntEnd

--> second error:

we do not check the carry bit
after adding the next number
to detect number overflow

--> but what to do in case of error?

- just give up

- tell user on display and
  return to Rosy

- return an error indication
  and allow calling program to
  decide.

[illegible]

Write a routine adding all positive integers
of a given interval (allow 16-bit integers)

Constraints:

* write it as a subroutine

* program starts at loc. $0400

* parameters are at loc. $0600

loc.	contents		meaning
0600	IntBeg	-->	start of interval
0602	IntEnd	-->	end of interval
0604	SumInt	-->	resulting sum

Convention:

accumulators A and B are free for use

Error handling:

on return accumulator B will contain
a completion code as follows:

0 --> no error
1 --> IntBeg > IntEnd
2 --> sum too large for 16-bits

Fourth programming problem:

Structure:

```
IF <interval bad> THEN
   <return with code 1>
ELSE BEGIN
  <initialise sum to zero>
  FOR <iterative condition> DO BEGIN
    <add sum and count>
    IF <result too large> THEN
       <return with code 2>
    ELSE
       <update sum>
  END
  <return with code 0>
END
```


Fourth programming problem:

Add "N" consecutive integers:

```
        TTL    Subroutine AddInt
        STTL   Add "N" consecutive integers

        ORG    $400        program area

AddInt  CLR    SumInt      SumInt:= 0;
        CLR    SumInt+1
        LDD    IntBeg      start of interval
        CMPD   IntEnd      <= interval end?
        BLS    AddLoop     yes, no error
        LDB    #1          no, set code
        JMP    AddExit     and exit
AddLoop STD    --S         save count
        ADDD   SumInt      accumulated sum
        BCC    AddOk       if sum fits
        LDB    #2          else set code
        LEAS   2,S         and clear stack
        BRA    AddExit     and exit
AddOk   STD    SumInt      save it
        LDD    S++         re-load count
        ADDD   #1          step to next value
        CMPD   IntEnd      over interval end?
        BLS    AddLoop     no, continue
        CLRB               yes, say no error
AddExit RTS                and simply return
```

Type of subroutine parameters:

--> VALUE parameter

"value" of the parameter is passed

Example:

numbers, characters, value of expressions

restricted to be an "input only" parameter for the subroutine

--> VARIABLE parameter

"address" of the parameter is passed

Example:

address of a variable, startaddress of an array

"input" and / or "output" parameter for the subroutine

Call a subroutine to initialise an array of a given size to zero:

```
CALL Init (ArrayAddress, SizeValue)
```


Type of subroutine parameters:

--> Subroutine or Function parameter

the address of a subroutine or function is passed to the subroutine

Example:

Assume we want to write a subroutine integrating a function over interval [A, B]

The subroutine therefore must obtain function values f(x) for values of x in the interval [A, B]

```
CALL Integrate ( FunctionAddress,
                 ValueA, ValueB,
                 ResultAddress    )
```

* only the address of the function is passed

* the parameter definition for the function is unknown!

foil 41

[illegible] to pass parameters to subroutines:

--> processor registers

* fast and simple

* limited by the number of registers

(rewrite fourth programming problem
using register[illegible] X and [illegible] for input
[illegible]
register X to return the result [illegible])

--> dedicated memory locations (mail box)

* associated with the subroutine

* calling program places input in and
retrieves output from the mailbox

* address has to be known by caller

* mailbox must be in RAM
(coordination problems)

This method was used in our fourth
programming problem!

foil 42

[illegible] to pass parameters to subroutines:

--> parameter areas

* associated with the calling program

* base address is passed to subroutine
using a register

* possible to use it for different
[illegible] hence it's size must be
[illegible] to hold the largest number
of parameters

* parameter area must reside in RAM
(coordination problems)

* "in-line" parameter area is a special
case (i.e. return address is address
of first parameter)

Example:

```
BSR   Init       initialise an array
FDB   Bin        address of array Bin
FCB   5          size of array Bin
---              next instruction
```

Note: Should only be used, if the
parameters are "constant"

Ways to pass parameters to subroutines:

--> stack oriented parameter passing

* parameters are "pushed" on the system stack prior to calling the subroutine

* dynamic allocation (stack overflow!)

* no worry about [illegible]

PROBLEMS:

* order of pushing must be defined

* space for output parameters

* addressing of parameters

* who "cleans" the stack?

Example:

```
LDA   #5        push size of Bin
STA   -S
LDX   #Bin      push address of Bin
STX   --S
BSR   Init      get it initialised
LEAS  3,S       remove parameters
```

Macro - definition and use:

* macro assigns a name to an instruction sequence

* use of the name causes assembler to insert the instruction sequence into your program

--> TwoComp, a macro to compute two's complement of accumulator D

* macro definition:

```
TwoComp MACRO

        COMB          complement low byte
        COMA          complement high byte
        ADDD  #1      now two's complement

        ENDM
```

* use of macro:

```
        LDD   Value1  to be complemented
        TwoComp
        STD   Value2  two's complement
```

Macro and parameters:

* macros allow parameter substitution
* parameters are indicated in the body by:

  &1, &2, ... , &9

  thus allowing up to 9 parameters
* these 'formal' parameters are replaced by the 'actual' parameters used in the macro call

  MACNAM <par.1>, <par.2>, ... , <par.9>

NOTE:

* macros must be defined before they are used
* local labels are not supported!
* number of arguments used in a macro call is not available

SEE:

chapter VII of TSC 6809 Assembler manual for more details ...

Define macro with parameters:

--> TwoComp compute 2's complement of first argument and store result into second argument

* macro definition:

```
TwoComp MACRO

        LDD    &1        number to complement
        COMB             complement low byte
        COMA             complement high byte
        ADDD   #1        now two's complement
        STD    &2        store 2's complement

        ENDM
```

* use of macro now:

```
        TwoComp Value1,Value2
```

NOTE:

* we expect to obtain two memory addresses! (Why?)
* there is an error in the macro!

Macro definition - Conditional assembly:

MON - macro for ROSY monitor requests

```
* define range of monitor requests

MonMin   EQU    0          lowest request
MonMax   EQU    46         highest request
MonStop  EQU    1          stop execution
         SPC    3

* define macro to handle monitor requests

MON      MACRO
MonPar   SET    &1         get request number
* assert parameter is valid
         IF     MonPar<MonMin
         ERR    too small for Rosy
MonPar   SET    MonStop
         ELSE
         IF     MonPar>MonMax
         ERR    too large for Rosy
MonPar   SET    MonStop
         ENDIF
         ENDIF
* generate the monitor call
         SWI
         FCB    MonPar
         ENDM
```


Macros - their advantage and disadvantage

Advantages of macros:

* shorter source programs

* once debugged -->
  code error free whenever used

* easier to implement changes

* user interface same, but macro body may
  change (useful in collaborations)

* extend or clarify instruction set

Disadvantages of macros:

* macro is expanded every time it is used
  may waste memory

* single macro may generate lot of
  instructions

* macro call might have hidden effects...
  (registers, flags)

[illegible]

--> Purpose:

Provide a desk calculator program for simple 16 bit integer arithmetic on the MC 6809 microprocessor

[illegible]

use of the Floxy monitor system.

--> Program development phases:

for a detailed discussion see:

John F. Wakerly

Microcomputer Architecture and Programming, chapter 12

John Wiley & Sons, New York, 1981

[illegible] desk calculator

--> Program development phases:

* Problem definition
  (Purpose of the program)

* Requirements analysis
  -> computer, staff, cost, schedule

  [illegible]

  -> inputs, outputs, commands

* Design
  -> programs equal algorithms plus data structures
  -> bottom-up / top-down development
  (Program, modules, interfaces)

* Documentation
  -> self documenting code

* Coding
  -> coding rules

* Testing and debugging
  -> bottom-up / top-down development
  use of debug monitor

* Maintenance
  -> bug correction, new features, functional improvements

--> External specification

* prompt user for first operand
* prompt user for operator
* prompt user for second operand
* display result on screen
* [illegible]

--> Design

* top-down design strategy:

  - main module with processing
    loop calling other modules

  - module to read an operand

  - module to read an operator

  - module to add two integers

  - module to subtract two integers

  - module to multiply two integers

  - module to divide two integers

  - module to handle error messages


```
                NAM     Calcul
                ORG     $400

*               Allocate space for variables

Arg-L           RMB     2       left-hand argument
Arg-R           RMB     2       right-hand argument
Res-Val         RMB     1       result of operation
Dec-Opr         RMB     1       decoded operator
*                               1 --> add
*                               2 --> subtract
*                               3 --> multiply
*                               4 --> divide

*               FLOXY monitor calls

InOn            SET     12      open primary input
OutOn           SET     13      open primary output
PCRLF           SET     27      write CR LF
Write           SET     35      general print rout.
DecWord         SET     0       format for decimal

*               define monitor call macro

MON             MACRO
                SWI
                FCB     &1
                ENDM
```

Project: Calcul - a simple desk calculator

```
*           Begin of main program

Calcul      MON    InOn        open input
            MON    OutOn       open output

*           read left-hand argument

W-Loop      LDX    #Arg-L      push address
            STX    --S
            JSR    GetNum      read number
            LEAS   2,S         remove parameter

*           read and decode operator

            LDX    #Dec-Oper   push address
            STX    --S
            JSR    GetOper     read operator
            LEAS   2,S         remove parameter

*           read right-hand argument

            LDX    #Arg-R      push address
            STX    --S
            JSR    GetNum      read number
            LEAS   2,S         remove parameter
```


Project: Calcul - a simple desk calculator

Structure:

```
-->      IF <operator> EQ '+' THEN <call add>
    ELSE IF <operator> EQ '-' THEN <call sub>
    ELSE IF <operator> EQ '*' THEN <call mul>
    ELSE IF <operator> EQ '/' THEN <call div>
    ELSE <call error>

--> CASE <operator> OF
      '+'  : <call add>
      '-'  : <call sub>
      '*'  : <call mul>
      '/'  : <call div>
      ELSE: <call error>
    END of case
```

Project: Calcul - a simple desk calculator

Structure:

```
-->       IF <dec.oper> EQ 1 THEN <call add>
     ELSE IF <dec.oper> EQ 2 THEN <call sub>
     ELSE IF <dec.oper> EQ 3 THEN <call mul>
     ELSE IF <dec.oper> EQ 4 THEN <call div>
     ELSE <call error>

--> CASE <dec.oper> OF
       1 : <call add>
       2 : <call sub>
       3 : <call mul>
       4 : <call div>
       ELSE: <call error>
     END of case
```


Project: Calcul - a simple desk calculator

```
*          decide which routine to call

           LDA   Dec-Oper    decoded operator
           DECA
           ASLA              word access
           LDX   #C-Table    case table addr.
           JMP   [A,X]       case element jum

*          case table with element addresse

C-Table    FDB   C-Add       integer add
           FDB   C-Sub       integer subtract
           FDB   C-Mul       integer multiply
           FDB   C-Div       integer divide

*          CALL  I-Add(Arg-L,Arg-R,Res-Val)

C-Add      LDX   #Res-Val    push result addr
           STX   --S
           LDX   #Arg-R      push Arg-R addr.
           STX   --S
           LDX   #Arg-L      push Arg-L addr.
           STX   --S
           JSR   I-Add       call processor
           LEAS  6,S         remove parameter
           BRA   C-End       skip other elem.
```

```
*          perform error test

C-End      TSTB                call successful?
           BEQ    Display      yes, show result
           JSR    InError      no, report error
           LBRA   W-Loop       and restart

*          [illegible]

Display    [illegible]
           LDB    #DecWord     format spec.
           MON    Write        display result
           MON    PCRLF        follow with CR LF
           LBRA   W-Loop       and restart

*          insert library routines

           LIB    GetNum
           LIB    GetOper
           LIB    InError
           LIB    I-Add
           LIB    I-Sub
           LIB    I-Mul
           LIB    I-Div

           END    Calcul
```



```
****       GetNum - read an unsigned integer
*
* On stack:
*                address to store number
*                return address
*

*          [illegible] monitor calls:

ErrMess    SET    2             error message
PData      SET    25            print line
Read       SET    36            gen. read rout.

*
GetNum     LDX    #NumPrompt
           MON    PData         prompt for number
           LDB    #DecWord      read 16-bit int.
           MON    Read
           TSTB                 successful?
           BEQ    NumExit       yes
           MON    ErrMess       no, give message
           BRA    GetNum        try again
*
NumExit    STX    [2,S]         store result
           RTS                  and return
*
NumPrompt  FCC    "type argument: "
           FCB    4             EOT character
```

Project: Calcul - a simple desk calculator

```
****        I-Add - add two 16 bit numbers
*
* On stack:
*             address to store number
*             address of right-hand operand
*             address of left-hand operand
*             return address
*

*           FLOXY monitor calls:

Print    SET     26            print line, CR, LF

*           define parameters:

L-Arg    SET     2             offset from S-reg
R-Arg    SET     4             offset from S-reg
E-Valu   SET     6             offset from S-reg

*
I-Add       LDX    #AddPrompt
            MON    Print          send message
            LDD    #0             dummy result
*
AddExit     STD    [E-Valu,S] store result
            RTS                   and return
*
AddPrompt FCC      "I-Add: not yet implemented"
            FCB    4              EOT character
```


Multiply two 16 bit numbers:

```
*
****        multiply two 16 bit numbers
*
* On stack: address to store result
*           address of right-hand operand
*           address of left-hand operand
*           return address
*
* On exit:  32 bit result stored
*
L-Value  RMB     2             multiplicand
R-Value  RMB     2             multiplier
Result   RMB     4             32-bit result
Upp-16   EQU     Result        upper word
Low-16   EQU     Result+2      lower word
*
Upper    SET     0             upper byte
Lower    SET     1             lower byte
*
* Picture:
*
*  <          32 - bit result          >
*  <     Upp-16     > <     Low-16     >
*   upper    lower    upper    lower
*
*     n       n+1      n+2      n+3
*
```

Multiply two 16-bit numbers (cont.):

```
L-Arg   SET    2            offset from S-reg
R-Arg   SET    4            offset from S-reg
E-Valu  SET    6            offset from S-reg
*
Mul-32  LDD    [L-Arg,S]    value multiplicand
        STD    L-Value      keep it local
        LDD    [R-Arg,S]    value multiplier
        STD    R-Value      keep it local
        CLR    Upp-16+Upper
        CLR    Upp-16+Lower
*
* multiply L-Value(low) * R-Value(low)
*
        LDA    L-Value+Lower
        LDB    R-Value+Lower
        MUL
        STD    Low-16
*
* multiply L-Value(low) * R-Value(upper)
*
        LDA    L-Value+Lower
        LDB    R-Value+Upper
        MUL
        ADDD   Upp-16+Lower  adds upper
        STD    Upp-16+Lower
        BCC    Mul-1         carry?
        INC    Upp-16+Upper  yes, incr. upper
```


Multiply two 16-bit numbers (cont.):

```
*
* multiply L-Value(upper) * R-Value(low)
*
Mul-1   LDA    L-Value+Upper
        LDB    R-Value+Lower
        MUL
        ADDD   Upp-16+Lower  adds 'middle'
        STD    Upp-16+Lower
        BCC    Mul-2         carry?
        INC    Upp-16+Upper  yes, incr. upper
*
* multiply L-Value(upper) * R-Value(upper)
*
Mul-2   LDA    L-Value+Upper
        LDB    R-Value+Upper
        MUL
        ADDD   Upp-16+Upper  add result upper
        STD    Upp-16+Upper
*
* store final result
*
        LEAX   [E-Valu,S]    result address
        STD    X++           store upper
        LDD    Low-16+Upper
        STD    0,X           store lower
        RTS
```