Assembly Language Macros

- An assembly language **macro** is a template whose format represents a pattern of 0 or more assembly language statements that might be common to multiple programs.

- For this purpose, a *macro language* is used to provide a syntax for defining macros.

- Where a sequence of assembly language statements can be represented by a macro, a *macro call* is inserted into the assembly program source code where the assembly code would otherwise go.

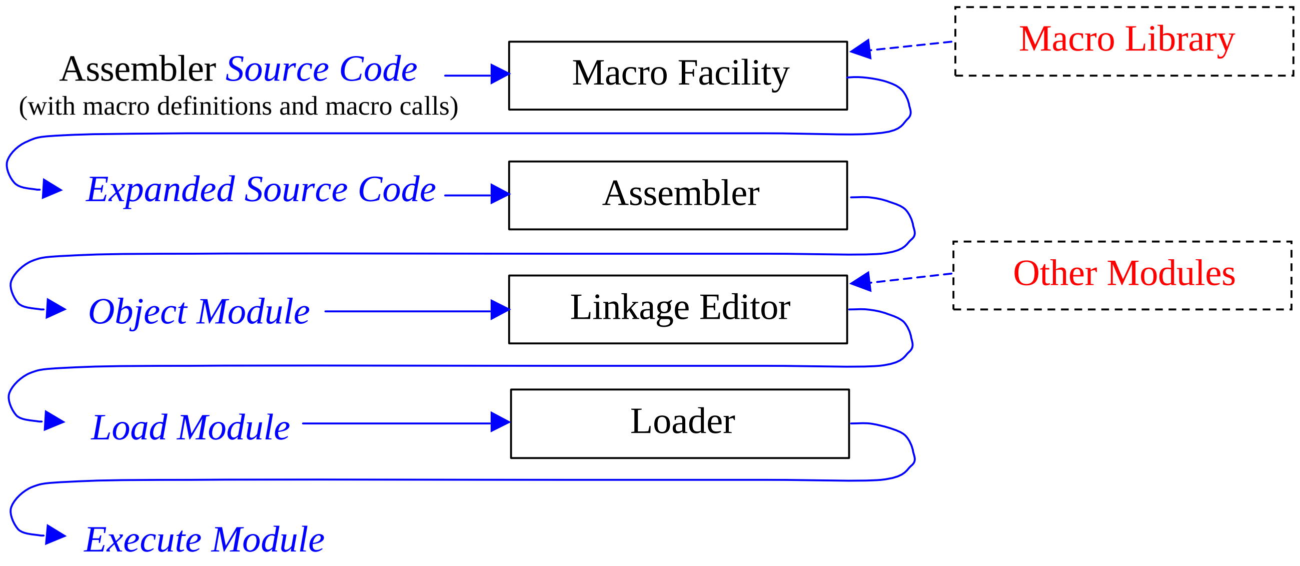
- A *macro facility* is used to interpret macro definitions and expand each macro call as it occurs with the requisite pattern of assembly language statements, providing *expanded source code* ready for the assembler.

  Hence, the macro facility is a *preprocessor*, which interprets all macro calls into assembly code prior to passing the *expanded code* on to the assembler. A macro facility is an add-on piece of system software, a convenience for the programmer, to facilitate production of multiple lines of commonly occurring code via single macro calls embedded in the programmer's assembly program. The macro preprocessor included with C compilers (with calls such as `#include`) uses the same idea, albeit assembly language macro facilities predate the similar compiler preprocessors.

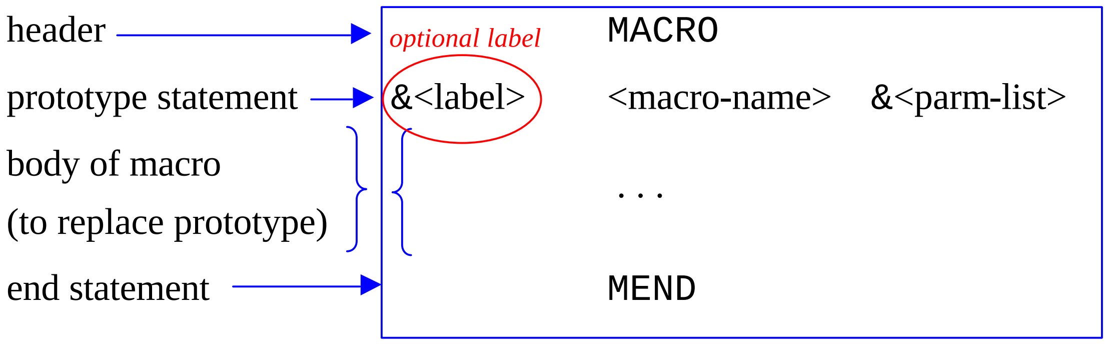
Macro Definition

- Assembly language macro definitions can be predefined and placed in a macro library, or can be included "in-line" with the assembly language program.

- The handling sequence for the program becomes:



Macro Definition Format



Note that the macro prototype statement differs from that of the course text book to permit inclusion of a label.

A First Example: **`PUTC`**

Consider the following simple definition of a macro for the output of a character:

```
          MACRO
&INIT     PUTC        &CHAR, &DEV
&INIT     STA         PUTCSAVE
          LDCH        &CHAR
          J           PUTCLOOP
PUTCSAVE  RESW        1
PUTCLOOP  TD          &DEV
          JEQ         PUTCLOOP
          WD          &DEV
          LDA         PUTCSAVE
          MEND
```

If our assembler comes with a macro facility for handling this format, then `PUTC` can be used in a program in the same manner as an instruction. In particular, if the definition of `PUTC` is in the macro library or included “in-line” in the assembly code, then the preprocessor will be able to expand any statement whose op code is `PUTC`.

Macro Expansion of of `PUTC` (first version)

Suppose that you have a code fragment utilizing `PUTC` as follows:

```
&INIT                                 &CHAR     &DEV
                   . . .
                   JEQ        NEXT
    DUMPX          PUTC       (LINE,X), =X'04'
    NEXT           STA        LAST

                   . . .
```

Then the macro facility will apply the `PUTC` definition to produce the expanded code:

```
                   . . .

                   JEQ        NEXT
.DUMPX             PUTC       (LINE,X), =X'04'
DUMPX              STA        PUTCSAVE
                   LDCH       LINE,X
                   J          PUTCLOOP
PUTCSAVE           RESW       1
PUTCLOOP           TD         =X'04'
                   JEQ        PUTCLOOP
                   WD         =X'04'
                   LDA        PUTCSAVE
NEXT               STA        LAST

                   . . .
```

Note that the parentheses around `LINE,X` are stripped (parentheses are used to group comma separated items).

Problems with the First Version of `PUTC`

The `PUTC` macro definition given as a first example has a severe flaw in that it can only be used once in any given assembly language program. If it was used a second time, the labels `PUTCSAVE` and `PUTCLOOP` would get generated again in the expansion of `PUTC` in its second location, which in turn would cause a "duplicate label" error when the expanded code is passed on to the assembler.

System Variables

To correct for this problem, two additional types of macro "*system variables*" are provided to go with the `&` prefixed variable names of the prototype statement:

1. **Set Variables** - `&` prefixed variables that can be "set" by the `SET` directive (with limited arithmetic capability); e.g. `&GCNT`.

2. **System Qualifiers** - `$` prefixed symbols with a value that is automatically adjusted to a new value each time a macro expansion (not necessarily the same macro) occurs; e.g., `$LOOP`. `$` is expanded to `$AA` for the 1st macro expansion, `$AB` for the 2nd and so forth.

Concatenation of Symbols

Macro text elements *can be concatenated together*; for example, if the prototype statement has the label `&INIT` and appears in the macro definition in the construction

```
ONE&INIT
```

then if the value of `&INIT` is `DUMPX`, the macro facility will expand the construction as

```
ONEDUMPX
```

Note that this can be interpreted successfully because the `&` and trailing blank allow `&INIT` to be identified. If the order was reversed (`&INITONE`) this would not be possible. For this case, the symbol → (or ~) is used to provide a right delimiter; i.e.,

```
&INIT→ONE
```

Hence a construction such as

```
X&PARM→Y
```

generates

```
X12Y
```

if `&PARM` has the value `12`.

Revised PUTC using SET Variables

If we revisit the PUTC macro taking advantage of set variables, an "improved" construction might be:

```
&GCNT       SET      0                    (global SET)
            MACRO
&INIT       PUTC     &CHAR, &DEV
&GCNT       SET      &GCNT + 1
&INIT       STA      SAVE&GCNT
            LDCH     &CHAR
            J        LOOP&GCNT
SAVE&GCNT   RESW     1
LOOP&GCNT   TD       &DEV
            JEQ      LOOP&GCNT
            WD       &DEV
            LDA      SAVE&GCNT
            MEND
```

The global SET takes place as the macro definitions are read, initializing the set variable to 0. It is subsequently incremented by the internal SET statement each time the PUTC macro is expanded.

Macro Expansion of Second Version of `PUTC`

Using the same code fragment as for the first version of `PUTC`,

```
                 . . .

                 JEQ       NEXT
   DUMPX         PUTC      (LINE,X), =X'04'
   NEXT          STA       LAST

                 . . .
```

for the sake of illustration assume that 4 prior macro calls for `PUTC` have occurred before the macro facility encounters this code fragment, so the value of the global set variable `&GCNT` has incremented to 4. Then the expansion in this case will be

```
                 . . .

                 JEQ       NEXT
   .DUMPX        PUTC      (LINE,X), =X'04'
   DUMPX         STA       SAVE5
                 LDCH      LINE,X
                 J         LOOP5
   SAVE5         RESW      1
   LOOP5         TD        =X'04'
                 JEQ       LOOP5
                 WD        =X'04'
                 LDA       SAVE5
   NEXT          STA       LAST

                 . . .
```

Note that the in expanding the call, the macro facility incremented the value of `&GCNT` from `4` to `5`.

Problems with the Second Version of `PUTC`

There is still a potential problem with the approach of using set variables, because a name such as `SAVE3` is one a programmer might use (and so inadvertently set up code for which the macro facility produces a duplicated symbol).

System Qualifiers

For this reason, *special system qualifiers* are provided. In this case the convention is that under macro expansion, the "`$`" symbol is replaced by

`$AA`	within the 1st macro expansion
`$AB`	within the 2nd macro expansion
`. . .`	
`$AZ`	within the 26th macro expansion
`$A0`	within the 27th macro expansion
`. . .`	
`$A9`	etc.
`$BA`	
`. . .`	
`$Z9`	

The `$` system qualifier is advanced for each macro expansion, *whether or not the expansion makes use of it*.

A Final Version of `PUTC` using System Qualifiers

Revisiting the `PUTC` macro using system qualifiers, an "even better" construction might be:

```
                    MACRO
    &INIT           PUTC       &CHAR,  &DEV
    &INIT           STA        $SAVE
                    LDCH       &CHAR
                    J          $LOOP
    $SAVE           RESW       1
    $LOOP           TD         &DEV
                    JEQ        $LOOP
                    WD         &DEV
                    LDA        $SAVE
                    MEND
```

This time, if 4 prior macro expansions have occurred, *not necessarily to* `PUTC` *and not necessarily using the* `$` *system qualifier*, the `$` system qualifier has been advanced through `$AA`, `$AB`, `$AC`, and `$AD`, so for this expansion the code will be

```
                    . . .

                    JEQ        NEXT
    .DUMPX          PUTC       (LINE,X),  =X'04'
    DUMPX           STA        $AESAVE
                    LDCH       LINE,X
                    J          $AELOOP
    $AESAVE         RESW       1
    $AELOOP         TD         =X'04'
                    JEQ        $AELOOP
                    WD         =X'04'
                    LDA        $AESAVE
    NEXT            STA        LAST
```

Þ **Each macro expansion gets its own `$` system qualifier, whether or not it uses the `$` system qualifier.**

Macro Calls Within a Macro

There is no reason that the macro facility cannot successfully process a macro call within a macro. For example,

```
                    MACRO
   &TOP             PUTMSG    &MSG, &DEV, &LEN
   &TOP             STA       $SAVA
                    STX       $SAVX
                    CLEAR     X
   $LOOP            PUTC      ((&MSG,X)), &DEV
                    TIX       &LEN
                    JLT       $LOOP
                    LDA       $SAVA
                    LDX       $SAVX
                    J         $NEXT
   $SAVA            RESW      1
   $SAVX            RESW      1
   $NEXT            RESW      0
                    MEND
```

Observations

- The PUTMSG macro (and the PUTC macro for that matter) generate code that first *saves the system state* (by saving the registers it works with), then restores the system state upon exit. Strictly speaking, this is not necessary (the programmer could do it), but it is advisable since the whole idea of using macros is to save the programmer work.

- The value of the $ system qualifier for a PUTMSG macro expansion is resumed after the PUTC expansion contained within it is completed (although PUTC gets its own $ system qualifier value). This means that if PUTMSG is used twice in succession by a programmer, then if for the first expansion the $ system qualifier has value $AC, on the 2nd call the $ system qualifier for PUTMSG will have the value $AE since there was an intervening expansion of PUTC (which gets the $ system qualifier value of $AD).

Example Macro Expansion of **PUTMSG**

For the code fragment,

```
                  . . .

                  JEQ        NEXT
     DUMPM        PUTMSG     =C’COP 3601’, =X'04', MSGLEN
     NEXT         STA        LAST

                  . . .
```

assuming that 4 prior macro expansions have occurred before the macro facility encounters this code fragment (so the `$` system qualifier is at `$AE`) , then the expansion will be

```
                  . . .

                  JEQ        NEXT
    .DUMPM        PUTMSG     =C’COP 3601’, =X'04', MSGLEN
    DUMPM         STA        $AESAVA
                  STX        $AESAVX
                  CLEAR      X
    .$AELOOP      PUTC       (=C’COP 3601’,X), =X'04'
    $AELOOP       STA        $AFSAVE
                  LDCH       =C’COP 3601’,X
                  J          $AFLOOP
    $AFSAVE       RESW       1
    $AFLOOP       TD         =X'04
                  JEQ        $AFLOOP
                  WD         =X'04
                  LDA        $AFSAVE
                  TIX        MSGLEN
                  JLT        $AELOOP
                  LDA        $AESAVA
                  LDX        $AESAVX
                  J          $AENEXT
    $AESAVA       RESW       1
    $AESAVX       RESW       1
    $AENEXT       RESW       0
    NEXT          STA        LAST

                  . . .
```

Note: the macro facility has to be “smart enough” to not expand `$` a 2nd time

`PUTC` *expansion:* (`$` *system qualifier has the value* `$AF`)

`PUTMSG` *expansion:* (`$` *system qualifier has the value* `$AE`)

Conditional Assembly

Conditional Assembly in a macro facility refers to mechanisms for providing program control over the code generation process. These require the addition of macro facility commands such as

- `IF-ELSE-ENDIF`
- `WHILE-ENDW`
- `GOTO`

along with branch point labels (e.g., `!EXIT`) and representational forms for comparison; e.g.,

`EQ`	for =	`NE`	for ≠
`LT`	for <	`LTE`	for <=
`GT`	for >	`GTE`	for >=

and logical operators `AND, OR, NOT` with the usual parenthesis grouping.

Additionally, since the macro facility is essentially a text processor, it is necessary to provide at least rudimentary string processing capabilities, including *system functions* for

- working with comma separated lists (`%NITEMS`)
- length of input parameters (`%LENGTH`)
- substrings (`%SUBSTR`)
- indexing into a string (e.g., `&MSG[0]`, or `&INIT[2]`)

`%NITEMS()` is the number of parameters given by the programmer; if a parameter `&L` is a comma separated list, then `%NITEMS(&L)` gives the number of items in the list.

`%LENGTH(&L)` gives the length of `&L` as a text string.

`%SUBSTR(&L, 3, 4)` gives the (up to) length 4 substring of `&L` starting from index 3.

`&L[2]` gives the character at index 2 of `&L`. `&L[2,4]` gives the string consisting of the characters at indices 2 and 4 of `&L`.

For example, given the macro definition

```
                 MACRO
&LABL            SWAPR     &ONE, &TWO, &TEMP
                 IF        (%NITEMS() GT 2)
&LABL            ST→&ONE   &TEMP
                 RMO       &TWO, &ONE
                 LD→&TWO   &TEMP
                 ELSE
&LABL            ST→&ONE   $TEMP
                 RMO       &TWO, &ONE
                 LD→&TWO   $TEMP
                 J         $NEXT
$TEMP            RESW      1
SNEXT            RESW      0
                 ENDIF
                 MEND
```

%NITEMS() = 3

then the call

```
                 SWAPR     S, T, SWAPAREA
```

has the expansion

```
.                SWAPR     S, T, SWAPAREA
                 STA       SWAPAREA
                 RMO       S,T
                 LDT       SWAPAREA
```

and the call (first macro call, so $ → $AA) %NITEMS() = 2

```
                 SWAPR     S, T
```

has the expansion

```
.                SWAPR     S, T
                 STA       $AATEMP
                 RMO       S,T
                 LDT       $AATEMP
                 J         $AANEXT
$AATEMP          RESW      1
$AANEXT          RESW      0
```

Keyword and Positional Parameters

The prototype statement's parameter list as given to this point uses what are known as *positional parameters*. The position of the parameter in the comma separated list determines which entry it represents (note two successive commas in the list represents an omitted parameter).

A *keyword parameter* is one specified in the comma separated list by the form &<name>=<value> or &<name>= . In the first form, <value> is the default used in the macro's expansion if the programmer does not supply the parameter. In the second form, the programmer must supply the value (by name). For example, if we revise the prototype statement for the final version of `PUTC` to be

```
  &INIT          PUTC      &CHAR=, &DEV==X'04'
```

then for the call

```
                 PUTC      CHAR=MSG
```

then the expansion (first macro call, so `$` $\rightarrow$ `$AA`) is

```
  .               PUTC      CHAR=MSG
                  STA       $AASAVE
                  LDCH      MSG
                  J         $AALOOP
  $AASAVE         RESW      1
  $AALOOP         TD        =X'04'
                  JEQ       $AALOOP
                  WD        =X'04'
    LDA           $AASAVE
```

If the next call is

```
LOOP         PUTC     CHAR=MSG, DEV==X'05'
```

then the expansion is

```
.LOOP        PUTC     CHAR=MSG, DEV==X'05'
             STA      $ABSAVE
             LDCH     MSG
             J        $ABLOOP
$ABSAVE      RESW     1
$ABLOOP      TD       =X'05'
             JEQ      $ABLOOP
             WD       =X'05'
  LDA        $ABSAVE
```

The same expansion would have resulted if the call had been

```
LOOP         PUTC     DEV==X'05', CHAR=MSG
```

In other words, because the parameters are "named" by the keywords, the order in which they are given does not matter. Key word parameters are sometimes used in combination with positional parameters to provide a default. For example, if the `PUTC` prototype statement was

```
&INIT        PUTC     &CHAR, &DEV==X'04'
```

then the `&CHAR` parameter is specified by position and the `&DEV` parameter is a keyword parameter with a default. In this case the following calls are all equivalent:

```
             PUTC     DEV==X'04', =C'X'
             PUTC     =C'X'
             PUTC     =C'X', DEV==X'04'
```

It may be argued that using keyword parameter lists burdens the programmer with having to remember names, but it can be very convenient for providing default values, and is particularly useful if there are multiple parameters needed in the macro.

For a simple example taking advantage of both conditional assembly and keyword parameters, suppose that you want a macro that decides whether to use `TIX` or `TIXR`; i.e., you want a call such as

```
                 MYTIX    TLOC
```

to expand as

```
  .              MYTIX    TLOC
                 TIX      TLOC
```

and a call such as

```
                 MYTIX    REG=A
```

to expand as

```
      .          MYTIX    REG=A
                 TIXR     A
```

The form the macro definition might take is

```
                 MACRO
  &INIT          MYTIX    &LOC, &REG=
                 IF       (%LENGTH(&LOC) NE 0)
  &INIT          TIX      &LOC
                 ELSE
  &INIT          TIXR     &REG
                 ENDIF
                 MEND
```

It is also perfectly OK is a macro doesn't even generate any code; for example, consider the macro to lay in code for incrementing register `X` via the `TIXR` operation.

```
                 MACRO
&INIT            INCRX     &AMT
&CNT             SET       &AMT
                 WHILE     (&CNT GT 0)
                 TIXR      X
&CNT             SET       &CNT - 1
                 ENDW
                 MEND
```

Then the call

```
                 INCRX     4
```

has the expansion

```
.                INCRX     4
                 TIXR      X
                 TIXR      X
                 TIXR      X
                 TIXR      X
```

and the call

```
                 INCRX     0
```

has the expansion

```
.                INCRX     0
```