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:

  Assembler Source Code (with macro definitions and macro calls) → Expanded Source Code → Object Module → Load Module → Execute Module → Macro Facility → Assembler → Linkage Editor → Loader → Macro Library → Other Modules
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: \textbf{PUTC}

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

\begin{verbatim}
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
\end{verbatim}

If our assembler comes with a macro facility for handling this format, then \texttt{PUTC} can be used in a program in the same manner as an instruction. In particular, if the definition of \texttt{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 \texttt{PUTC}. 

Macro Expansion of of **PUTC** (first version)

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

```
&INIT

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

&CHAR

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

&DEV

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

\[
\text{ONE} \& \text{INIT}
\]

then if the value of \&INIT is DUMP\(X\), the macro facility will expand the construction as

\[
\text{ONE}DUMPX
\]

Note that this can be interpreted successfully because the \& and trailing blank allow \&INIT to be identified. If the order was reversed (\&INIT\(\text{ONE}\)) this would not be possible. For this case, the symbol \(\rightarrow\) (or \(\sim\)) is used to provide a right delimiter; i.e.,

\[
\&\text{INIT} \rightarrow \text{ONE}
\]

Hence a construction such as

\[
X\&\text{PARM} \rightarrow Y
\]

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,

\[
\text{JEQ\ \\ NEXT} \\
\text{DUMPX\ \ PUTC\ \ (LINE,X), =X'04'} \\
\text{NEXT\ \ STA\ \ LAST}
\]

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

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

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 \texttt{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, \textit{special system qualifiers} are provided. In this case the convention is that under macro expansion, the "$" symbol is replaced by

- \$AA within the 1\textsuperscript{st} macro expansion
- \$AB within the 2\textsuperscript{nd} macro expansion
- \ldots
- \$AZ within the 26\textsuperscript{th} macro expansion
- \$A0 within the 27\textsuperscript{th} macro expansion
- \ldots
- \$A9 etc.
- \$BA
- \ldots
- \$Z9

The \$ system qualifier is advanced for each macro expansion, \textit{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
```

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 2\textsuperscript{nd} 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.,

<table>
<thead>
<tr>
<th>EQ</th>
<th>for =</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT</td>
<td>for &lt;</td>
</tr>
<tr>
<td>GT</td>
<td>for &gt;</td>
</tr>
<tr>
<td>NE</td>
<td>for ≠</td>
</tr>
<tr>
<td>LTE</td>
<td>for &lt;=</td>
</tr>
<tr>
<td>GTE</td>
<td>for &gt;=</td>
</tr>
</tbody>
</table>

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.

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

then the call

SWAPR S, T, SWAPAREA

has the expansion

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SWAPR S, T, SWAPAREA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STA SWAPAREA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMO S, T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDT SWAPAREA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and the call (first macro call, so $ → $AA)

SWAPR S, T

has the expansion

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SWAPR S, T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STA $AATEMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMO S, T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDT $AATEMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>J $AANEXT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$AATEMP RESW 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$AANEXT RESW 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $&<\text{name}>=<\text{value}>$ or $&<\text{name}>=$. In the first form, $<\text{value}>$ 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 $\rightarrow A$) is

<table>
<thead>
<tr>
<th></th>
<th>PUTC</th>
<th>CHAR=MSG</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA</td>
<td>$\text{SAASAVE}$</td>
<td></td>
</tr>
<tr>
<td>LDCH</td>
<td>$\text{MSG}$</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>$\text{AALOOP}$</td>
<td></td>
</tr>
<tr>
<td>$\text{AAASAVE}$</td>
<td>RESW</td>
<td>1</td>
</tr>
<tr>
<td>$\text{AAASAVE}$</td>
<td>TD</td>
<td>$=X'04'$</td>
</tr>
<tr>
<td>$\text{AAASAVE}$</td>
<td>JEQ</td>
<td>$\text{AALOOP}$</td>
</tr>
<tr>
<td>WD</td>
<td>$=X'04'$</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>$\text{SAASAVE}$</td>
<td></td>
</tr>
</tbody>
</table>
If the next call is

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

then the expansion is

```plaintext
.LOOP PUTC CHAR=MSG, DEV==X'05'
ST  $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

```plaintext
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

```plaintext
&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:

```plaintext
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

\texttt{MYTIX TLOC}

to expand as

\begin{verbatim}
  . MYTIX TLOC
  TIX TLOC
\end{verbatim}

and a call such as

\texttt{MYTIX REG=A}

to expand as

\begin{verbatim}
  . MYTIX REG=A
  TIXR A
\end{verbatim}

The form the macro definition might take is

\begin{verbatim}
MACRO &INIT MYTIX &LOC, &REG=
  IF (%LENGTH(&LOC) NE 0)
    &INIT TIX &LOC
  ELSE
    &INIT TIXR &REG
  ENDIF
MEND
\end{verbatim}
It is also perfectly OK if a macro doesn't even generate any code; for example, consider the macro to lay in code for incrementing register \( X \) via the \( \text{TIXR} \) operation.

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

Then the call

```
increx 4
```

has the expansion

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

and the call

```
increx 0
```

has the expansion

```
.    INCRX 0
```