Advanced

COLLABORATORS			
	TITLE : Advanced		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 23, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Advanced

1.1 MAIN

1.2 Conditional Branching with IF

```
      Conditional Branching with IF
      -----------------------------

ASM-One has the possibility to use Conditional Branching. This allowes
you to assemble your source based on certain conditions.

Obvious exmaples are that of excluding 68020++ code from a version
ment for the 68000/010.


                           UNDER CONSTRUCTION
```

1.3 Building and Using MACRO's

```
      Building and Using MACRO's
      --------------------------

MACRO's are pieces of coding presented by just one 'instruction' with
some 'operands'.

The most obvious thing people do with MACRO's is to replace repetive
coding with just one 'instruction'.

In this chapter I will try to explain how you can Build and Use your
own MACRO's.

     Before you start
     A simple MACRO
     A complex MACRO
```

1.4 Some usefull information before you start

```
    Some usefull information before you start
    -----------------------------------------

Where to place MACRO’s in your source ?
---------------------------------------

The best place to place MACRO definitions is in the beginning of your
source. Our in a seperate INCLUDE file, which is included in the
beginning of your source.

Beginning a MACRO
-----------------

You begin a MACRO by making up a label to define the MACRO. After
the label (in the next colom) you will put the word ’MACRO’:

mymacro:  MACRO

In the middle of the MACRO
--------------------------

He will be the coding that make up the MACRO.

Ending a MACRO
--------------

A MACRO should ALWAYS be ended with the word ’ENDM’:

    ENDM

MACRO Operands
--------------

MACRO operands start with an backslash (\) followed by a number ranging
from 0 to 9.

Special Operand
---------------

Since a MACRO could be used several times within you source, it would
be impossible to assign a unique label to labels within the MACRO.

Luckely there is a special symbol for that: \@.

Whenever ASM-One encounters this symbol within a MACRO, ASM-One will
replace the \a for a number.

MACRO Depth
-----------
```

MACRO’s can be called in MACRO’s up till 5 levels.

1.5 A simple MACRO

```
      A simple MACRO
      --------------

A very simple MACRO that I use myself is the following:

Gfx:    MACRO
    move.l  gfxbase,a6
    jsr _LVO\1(a6)
    ENDM

To use this MACRO in my source, I would do the following:

start:    Gfx Text


ASM-One will translate this to:

start:    move.l  gfxbase,a6
    jsr _LVOText(a6)

As you can see, ASM-One has replaced the ’\1’ with ’Text’.

Other MACRO’s that are used quite often are the PUSH and PULL MACRO’s.


PUSH:   MACRO
    movem.l \1,-(a7)
    ENDM

PULL:   MACRO
    move.l  (a7)+,\1
    ENDM

You could make the a little bit more complex by using one operand to
specify the address register to store the registerlist:

PUSH:   MACRO
    movem.l \1,-(\2)
    ENDM

PULL:   MACRO
    movem.l (\2)+,\1
    ENDM

Although this is almost the same as using the MOVEM instruction.

Advantage of these MACRO’s are that you will not forget how to store
a registerlist on stack. Many people have trouble with the - and +
before or after the address register. This is the option to make sure
you will always be right.
```

As you can see, these MACRO's will simply save you some serious typing
or thinming and thus preventing you from making mistakes.

1.6 A complex MACRO

A complex MACRO

Here's an example of a more complex MACRO:

```
do_it:    MACRO
    btst  #\1,d0
    bne.b \@1
    move.l  \2,d0
    lsl.w #\3,d0
    bra.b *+4

\@1:    moveq #0,d0
    ENDM
```

You could call this MACRO by typing the following in your source:

```
start:    do_it 10,d2,3
```

Also showed here is how to branch out of a MACRO. Since you don't
know where the MACRO will be used, you can't branch to a specific
label. If you would do that, you could get in to serious problems.

The only thing left is to specify the branch in bytes calculated from
the current offset (in assembler -- aangegeven met een -- *).

But be carefull !!! These hardcoded branches will NOT be updated
when something changes in your source !!!

1.7 Some MACRO's explained

Some MACRO's explained

If you browse to the AMiga Includes now and then, you will notice
a lot of MACRO's. These MACRO's are used so a C-like programming
style can be used in the Includes.

Many people have no clue at all what these MACRO's do. So I'll take this
oppertunite to explain some of these (very) complex MACRO's.

BITDEF MACRO

The BITDEF MACRO is as followes (example from the exec/exec.i):

```
  BITDEF  AF,68010,0  ; also set for 68020
```

As you can see, the MACRO uses three operands. The first is a pice of text used to uniquely identify this group of Bit Definitions (namely that the bits of the AttnFlags). The second is used to specify the uniquely identify the bit definition with the group (namely the type of processor). And the third is used to specify the actual value of the bit.

When compiled, we will get the following line in our source:

```
AFB_68010 equ 0
```

As you can see, a 'B' as been added to the label. This 'B' is not specified as one of the operands for the BITDEF MACRO. So this 'B' is inserted by the MACRO directly or by another MACRO called within the BITDEF MACRO.

Now let us look how the BITDEF MACRO looks like (exec/types.i):

```
**
**  Bit Definition Macro
**
**  Given:
**  BITDEF  MEM,CLEAR,16
**
**  Yields:
**  MEMB_CLEAR  EQU 16      ; Bit number
**  MEMF_CLEAR  EQU 1<<16     ; Bit mask
**

BITDEF      MACRO   ; prefix,&name,&bitnum
      BITDEF0 \1,\2,B_,\3
\@BITDEF    SET     1<<\3
      BITDEF0 \1,\2,F_,\@BITDEF
      ENDM

BITDEF0     MACRO   ; prefix,&name,&type,&value
\1\3\2      EQU     \4
      ENDM
```

Although this may look quite complex, it is actualy quite simple.

First let us look at the BITDEF MACRO itself, line by line:

```
BITDEF    MACRO
```

Here's where the MACRO definition starts.

```
    BITDEF0 \1,\2,B_,\3
```

Here another MACRO is called. As you rememebered the BITDEF MACRO needed THREE operands. But FOUR operands are specified here. Which means that the BITDEF0 operand needs FOUR operands !!

```
\@BITDEF  SET 1<<\3
```

Here, a value is assign to the label \@BITDEF. Since the BITDEF MACRO is used in virtualy every assembler include, it would be impossible to use a standard label. Therefore the '\@' is used. ASM-One will replace '\@' with a number.

Also remember, the THIRD operand (\3) was the ACTUAL value of the bit (zero in our example).

```
      BITDEF0 \1,\2,F_,\@BITDEF
```

Here the bitmask is generated. Since the value of the bit is different @BITDEF is specified as the value instead of \3.

```
      ENDM
```

End here the MACRO is ended.

If you look closely, you will see that there's not a lot happening in this MACRO. The real stuff is done in the BITDEF0 MACRO. Which needs FOUR operands and consists of only ONE line !!!

Let's see what that line does:

```
\1\3\2      EQU     \4
```

Ah, it's te actual line of coding that's been generated here.

As you can see, the MACRO needs FOUR operands. Operands \1, \2 and \3 will form the label, operand \4 the value.

Many people get confused here, becose the know the only specified THREE operands to the BITDEF MACRO, and don't understand where the FOURTH operand is comming from. But BITDEF0 and BITDEF are two DIFFERENT MACRO's.

Operand \1 for the BITDEF MACRO is NOT the operand \1 for the BITDEF0 MACRO !!

It is as follows:

BITDEF

Operand	Value
\1	AF
\2	68010
\3	0

BITDEF0

Operand	Value
\1	AF
\2	B_
\3	68010
\4	0

Now you will say that the \1 operand is the same for BOTH MACRO's since they have the same value. It may seem like that, but when I rearrange the operands for the BITDEF0 MACRO, I'll still get the same result:

```
BITDEF      MACRO   ; prefix,&name,&bitnum
      BITDEF0 \3,\2,B_,\1
      ENDM

BITDEF0     MACRO   ; prefix,&name,&type,&value
\4\3\2      EQU     \1
      ENDM
```

So as you can see, the operands are ONLY related to the operands specified for the MACRO. Operand \1 for the BITDEF MACRO has the 'AF' as value, while operand \1 for the BITDEF0 MACRO has '0' as value.

If you open exec/types.i you will see even more complex MACRO's.

Maybe you will now be able to read them and understand what they actualy do.

1.8 Using the NARG symbol

```
      Using the NARG symbol
      ---------------------
```

The NARG symbol is ONLY usefull with MACRO's. The NARG symbol will hold the number of operands passed to the MACRO. Together with Conditional Branching you can make decisions IN the MACRO.

Here's an example

```
adres:    MACRO
    IF  NARG=2
    move.l  d0,(\1,\2)
    ELSE
    move.l  d0,(\1,\2,\3)
    ENDIF
    ENDM
```

This MACRO can be called with 2 or 3 operands:

```
start:    adres 10,a1
    adres 10,a1,d1
    rts
```

Which would be translated to the following code:

```
start:    move.l  d0,(10,a1)
    move.l  d0,(10,a1,d1)
    rts
```

```
Note: Outside the MACRO’s NARG will not work or may have unknown values.
      Only use it WITHIN MACRO’s !!!
```

1.9 Errors & Debugging

```
        Errors
        ------

Besides some specific MACRO errors (like MACRO overflow, etc.) there
are some general errors you could encounter.

It’s known that ASM-One’s MACRO code has bugs, to swim around these
bugs it’s adviced to write MACRO’s using the following rules:

1) Place comment INFRONT or AFTER the MACRO, not IN the MACRO

2) Avoid the use of ANY unnecassary character IN the MACRO

3) NEVER use the backslash (\) as something different than it’s
   ment to do in a MACRO (identify Operands)


        Debugging
        ---------

MACRO’s are hard to debug, specialy when they are complex. And it’s getting
more difficult because ASM-One will give an error on the line the
MACRO is used, and not WITHIN the MACRO itself.

If you doubt the working of your MACRO, place it OUTSIDE a MACRO. Than
ASM-One is able to generate normal errors.

Another option is to TURN OFF the ’Source-Level-Debugger’, and debug
your code directly. In this case ASM-One will step through the assembled
source and you will see exactly what ASM-One has assembled.

Unfortunately, most errors are assembling errors, and the will force
you to work without the debugger.
```