DOUBLE-ASS

Source: 128er SH 22, Markt & Technik Verlag, Germany

Author: Rene van Belzen
e-mail: hurray@xs4all.nl
www : http://www.xs4all.nl/~hurray/cbm/

Brief description

With this 2-pass assembler you can code as well for the 8502 microprocessor as for the Z-80 microprocessor. Both processors are present in that wonderful machine, the CBM-128.

This document describes how to use the Double-Ass assembler, but doesn't teach how to create correct machine language programs. It assumes you have experience with the built-in Monitor of the 128-mode. However, this document contains some clues on how to code in assembly language.

Introduction

The 8502 normally runs only in CBM-64 and in CBM-128 mode, and the Z-80 only in CP/M mode. There is no reason why the Z-80 shouldn't run in the 128 mode too, so it does when you tell it to. Using a trick, that is explained later, you can switch between processors. You activate the one, while the other is silenced. When 'switched back' the 8502 runs happily on like it even never had stopped at all.

So far so good, but a pitfall is the lack of a solid user interface in the 128-mode for the Z-80. In ROM only the most necessary routines to start CP/M and restart C=128 mode are availabel. Nothing like the 128 ML-Monitor here.

So, when programming the Z-80 it makes only sense to produce short routines you can PEEK() into memory with a BASIC program. With Double-Ass all that has changed. Now you can design your own Z-80 routines, along with 8502 routines. You can use the block transfer power of the Z-80 along with the Kernel routines of the 8502.

The program Double-Ass is 9 Kbytes in length and is located in RAM bank 1, and can not be overwritten by BASIC source code. It is also possible to place the software on EPROM, so that it's permanently available. This option will not be covered here, however.

BTW, the machine codes for the 6502, 6510 (C=64), and 8502 (C=128) are exactly the same. The only difference seems to lay in the internal lay-out and processing of the data (and of course in the optional 2 Mhz CPU clock speed of the 8502).

Loading and running the program

------- --- ------- --- -------

To load the program type the following lines (each line activated by pressing the <RETURN>-key):
DLOAD "DOUBLE-ASS"
RUN

After a reset, the assembler can be re-installed by the following line (again, activate the line by pressing on the <RETURN>-key):
BANK 1 : SYS 5376

Operating the double assembler

--------- --- ------ ---------

Double-Ass is a full featured assembler that leans heavily on Basic 7.0. So you can still use commands you know as a BASIC programmer/operator. Furthermore you will find the following commands:

1. FIND(string)

This command lists all program lines in the source code in which the text string occurs. There are no spaces allowed in the FIND string (even not between 'FIND' and the string).

Examples:

(note: in this document the commands between '<' and '>' are keyboard entries, e.g. <RETURN> means you have to press the Return button on the keyboard)

Find the lines with the word 'LABELS' in it.

```
-------------------------------------
FINDLABELS<RETURN>
1010 ;LABELS

READY.
-------------------------------------
```

Find the lines with the word 'BUFFER' in it.

```
-------------------------------------
FINDBUFFER<RETURN>
1020 BUFFER = $0C00
2030 LDA BUFFER, X
3050 STA BUFFER, X
4000 ;THE BUFFER IS EMPTY

READY.
-------------------------------------
```

2. MERGE "FILE NAME", UNIT

This command loads the file that is specified by "FILE NAME" under (after) the source code that was already present in memory. If you want the line numbers appear in the correct order, use RENUMBER to re-link

the lines

E.g. suppose the line numbers in the old file run from 10 till 250. And suppose those in the merge file run from 100 till 300. After merging the line numbering will be 10, 20, ... 250, 100, 110, ... 300. After "RENUMBER 10,10" the line numbering will be 10,20,...,460.

The syntax of merge is similar to that of LOAD (C=64 BASIC 2.0 command).

Examples:

Merge the file "SECOND.SRC" from unit 8, and renumber the lines from 1000 and upwards, with an increment of 10.

```
---------------------------------------
MERGE "SECOND.SRC", 8<RETURN>

RENUMBER 1000, 10<RETURN>
---------------------------------------
```

Merge the file "GAME.SRC" from unit 1 (Datasette), and renumber the lines (10, 20, ...)

```
---------------------------------------
MERGE "GAME.SRC", 1<RETURN>

RENUMBER<RETURN>
---------------------------------------
```

The principles of 2-pass assembly

Double-Ass is an 2-pass assembler. This means that two steps take place:

STEP 1. Calculation of the so called "symbol table". In this table all labels are calculated into 16 bit values. Those values are derived from the label declarations in the source code. In general a label represents a memory location.

STEP 2. Combining of the source code and the symbol table into an object code (machine language).

How to make source code

The assembly source code is typed in lines, just like BASIC programs. The lines are stored temporary in memory, and can be saved just as any other Basic program listing.

Double-Ass knows if it is displayed in 40 or 80 columns mode, and adjusts the lay-out accordingly.

In the lines you can find two types of symbolic coding:
1. processor opcodes (8502 or Z-80)
2. pseudo opcodes (directions for the assembly)

BTW: Opcode stands for 'operation code'.

The pseudo opcode technique provides the programmer with ways to design his programs in a flexible way. Nothing is solid and concrete at the coding stage. He can play around with different strategies, without being bothered with the level of detail the microprocessor insists on. When the most efficient strategy is found he can assemble the source code to machine executable object code, and test the program.

The most common use of pseudo opcoding is labeling (the other pseudos are discussed below), which will be discussed in detail now.

Labels

A label is a bit like variables in the BASIC interpreter. BASIC can distinguish between the 2 first characters, Double-Ass between the first 40 characters. The first letter has to be a letter of the alphabet (a, b, ... z), the others may be letters or figures. Note that you can only use undercast letters, no capital letters.

Many assemblers use a maximum of 6 characters for a label. To ensure portability, you might consider limitting your labels to that maximum of 6 characters.

Labels can be used in different ways:

1. Definitions
The label gets an explicit value.

Example:

Assign the value $D600 to the label VDCREG.

```
---------------------------------------
VDCREG = $D600
---------------------------------------
```

2. Redefinitions
The label gets a new explicit value.

Example:
Increment the original value by 1, without introducing a new label (note the double '==', instead of the single '=' of the definition).

```
---------------------------------------
VDCREG == VDCREG + 1
---------------------------------------
```

3. Value of the program counter
The microprocessor keeps track of which opcode to perform next in a special register, the program counter. The notation of this program counter is an asterix '*'.

Example:
Assign the value of the program counter to the label PRGCNT.

```
---------------------------------------
```

```
PRGCNT = *
---------------------------------------

Note: alternative notations for the abobe expression are:
---------------------------------------
PRGCNT
PRGCNT:
---------------------------------------
```

4. Implicit definitions
The label is assigned a value in relation to the position in the source list.

Example:
Define the start addres of the object code (executable machine language).

```
---------------------------------------
*= $0B00
---------------------------------------
```

Mind you, don't place a space between '*' and '='!

Note: some assemblers have a specific way to define the start of the object code, namely:

```
---------------------------------------
.ORG = $0B00
---------------------------------------
```

Double-Ass doesn't recognize this pseudocode. If you are using source code listings made with other assemblers, please replace the expression '.ORG' with '*='.

Processor opcodes

These commands will be transfered to the machine language program during assembly. They are the executable commands. If you want to learn using them, I suggest you read a good book on either processor, or scan the Internet for useful documents.

8502 opcodes

```
---------------------------------------
ADC AND ASL BIT BPL BMI BVC BVS
BCC BCS BNE BEQ CMP CPX CPY DEC
EOR INC JMP JSR LDA LDX LDY LSR
NOP ORA PHA PHP PLA PLP RTI RTS
ROL ROR SBC SEC SED SEI TAX TAY
TSX TXA TXS TYA STA STX STY
---------------------------------------
```

Z-80 Opcodes

```
---------------------------------------
ADC  ADD  AND  BIT  CALL CCF  CP
CPD  CPDR CPI  CPIR CPL  DAA  DEC
DI   DJNZ EI   EX   EXX  HALT IM
IN   INC  IND  INDR INI  INIR JP
JR   LD   LDD  LDDR LDI  LDIR NEG
```

```
NOP  OR   OTDR OTIR OUT  OUTD OUTI
POP  PUSH RES  RET  RETI RETN RL
RLA  RLC  RLCA RLD  RR   RRA  RRC
RRCA RRD  RST  SBC  SCF  SET  SLA
SRA  SRL  SUB  XOR
---------------------------------------
```

Parameters

Parameters can be considered as the data where upon the (pseudo) opcode operates. They may consist of constants and labels, and a combination of these two.

Constants are noted in the following ways:

```
$31DE              hexadecimal
6590               decimal
%11 0001 1011 1110 binary
'AZ'               ASCII
```

Constants and labels can be mixed in the following ways:

```
+ addition
- subtraction
* multiplication
/ division
! logic OR
& logic AND
^ logic XOR
< low order value byte (lo-byte) of an 16-bit value
> high order value byte (hi-byte) of an 16-bit value
```

Examples of '^', '<' and '>'

```
---------------------------------------
%101010 ^ %010101
---------------------------------------
(result %111111)

---------------------------------------
< $1234
---------------------------------------
(result $34)

---------------------------------------
> $1234
---------------------------------------
(result $12)
```

There is no hierarchy, operations are evaluated as they are written down from left to right, with the exception of brackets '(' and ')' which are evaluated first.

Furthermore the different notations of constants may be mixed in one expression:

```
---------------------------------------
$13 + %1010 + 'C'
```

```
---------------------------------------
```

The addresses are calculated as efficient as possible on assembly. That means that a expression like:

```
---------------------------------------
LDA $91
---------------------------------------
```

uses the zeropage addressing technique. A absolute (16-bits) addressing technique can be forced with:

```
---------------------------------------
LDA !$91
---------------------------------------
```

which is assembled to:

```
---------------------------------------
AD 91 00 LDA $0091
---------------------------------------
```

Of course the same applies when a label is used:

```
---------------------------------------
ZEROPAGE = $91
ADC !ZEROPAGE
---------------------------------------
```

Pseudo opcodes
------ -------

```
---------------------------------------
;    =    ==   *=   .ASC .BYT .WOR
.MOD .OBJ .LIS .SYM .LST .SST .END
.FIL .LF  .IF= .IF< .SLO .FAS .ERR
---------------------------------------
```

;

format: ; some comment
With this pseudo opcode you can add comment that the assembler ignores. It is good practice to comment your source code. If you want to change it years later, it is not so difficult to understand what the routines are supposed to do.

=
==
*=

See above.

.ASC

format: .ASC "TEXT"
With this pseudo opcode you can pin down text to a certain location in memory. Where it occurs, the text is inserted. The number of characters is limited by the length of a BASIC line (160 characters). Longer texts should be listed in more than one line.

When you place a label in the line before the .ASC pseudo opcode you can refer to the first character in the text with that label.

.BYT

format: .BYT byte, byte, ...

Just as with .ASC you can define a string of characters (or values) with the pseudo opcode .BYT. The same restrictions and possibilities apply.

.WOR

format: .WOR word, word, ...
Similar to .BYT, only with 16-bit values in the lo/hi notation. E.g. $1234 is stored as:

```
---------------------------------------
>00B13 34 12
---------------------------------------
```

.MOD

format: .MOD 1 or .MOD 2
With this pseudo opcode you can switch to the Z-80 mode (.MOD 1) and the 8502 mode (.MOD 2). You can use the same pseudo opcodes, but only the appropriate processor opcodes are valid. If no .MOD is defined the 8502 mode applies.

If you want to switch from the 8502 processor to the Z-80 processor, you can use the routine below in the section Switching between processors.

.OBJ

This pseudo opcode implies where to put the object code (the executable machine language program). Here are the possibilities:

save no object code - choose:

```
---------------------------------------
.OBJ NOTHING
.OBJ N
---------------------------------------
```

save object code to memory (take care no to overwrite the assembler!) - choose:

```
---------------------------------------
.OBJ MEMORY
.OBJ M
---------------------------------------
```

save to memory with a certain memory bank configuration - choose:

```
---------------------------------------
```

```
.OBJ MEMORY, BANK
.OBJ M, BANK
---------------------------------------
```

save to a external storage device (dev. = 1, 8, ... 15), with an optional secundary address (sa) and file name (nm$) (those last two are obligatory in case of a floppy disc drive):

```
---------------------------------------
.OBJ dv (, sa)(, nm$)
---------------------------------------
```

.LIS

This pseudo opcode generates a formatted list of the source code. The code is a text file, not an BASIC program file. .LIS has similar varieties as .OBJ (s. above):

no listing:

```
---------------------------------------
.LIS NOTHING
.LIS N
---------------------------------------
```

list to screen:

```
---------------------------------------
.LIS SCREEN
.LIS S
---------------------------------------
```

list to external storage device:

```
---------------------------------------
.LIS dv (, sa)(, nm$)
---------------------------------------
```

.SYM

During pass 1 (calculation of labels) the symbol table is generated. If you place the .SYM pseudo opcode in the last line (with the highest line number), you get a list of labels after pass 1 is completed. Again, here are some varieties:

no listing:

```
---------------------------------------
.SYM NOTHING
.SYM N
---------------------------------------
```

list to screen:

```
---------------------------------------
.SYM SCREEN
.SYM S
---------------------------------------
```

list to external device (1, 4, 5, 8, ... 15):

```
---------------------------------------
```

.SYM dv (, sa)(, nm$)

.SST and .LST

You can store the symbol table (not the listing) itself on floppy disc, and retrieve it again when needed.

store to disc:

.SST dv, sa, nm$
---

load from disc:

.LST dv, sa, nm$

Note: (if you didn't know already) the secundary address for storing is 1 (one) and for loading 0 (zero). So in .SST use 1 for the secundary address, and in .LST use 0 for sa.

.FIL and .LF

To actually link the files you have to instruct the assembler to do so, with a .FIL statement. .FIL loads the next source file.

.FIL has the syntax:

.FIL dv, nm$

Place the .FIL statement as the last statement in the source file when assembling more than one source file. In the last source file in the assembly row use .LF instead. .LF should point to the first file in the assembly row.

.LF has the following syntax:

.LF dv, nm$

.END

With this pseudo opcode, that is optional, you invoke pass 2 immediately. If any opcodes follow, they are not processed by the assembler.

You could place some line of BASIC after the .END statement (e.g. comment lines from one programmer to another that don't have to appear in the formatted source listing, and are temporary in nature).

.FAS

Normally Double-Ass runs in 1 Mhz modus. With this pseudo opcode you can force the assembler to double it's speed. After assembly the processor speed returns to 1 Mhz.

.SLO

Switches back to the 1 Mhz modus. This can be omitted, because Double-Ass automatically returns to the 1 Mhz modus after assembly.

.ERR

If the 2 Mhz mode is active and a error occurs, Double-Ass switches back to 1 Mhz, and wait for a press on a button by the user, so he can see the error that occurred. After the user has pressed a key, the assembler continues at full speed.

.IF=

syntax:
.IF= value 1, value 2, line number

Value 1 and 2 are valid Double-Ass expressions. If values 1 and 2 are equal, the assembly continues on the specified line number, otherwise the assembly continues right after the .IF statement.

.IF< value 1, value 2, line number

Similar to .IF=. If value 1 is smaller than value 2 then assembly continues at the specified line number, otherwise the assembly continues after the .IF< statement.

Switching between processors
--------- ------- ----------

Switching between the 8502 microprocessor and the Z-80 microprocessor is a delicate business, which should be programmed with care. Use the following subroutine to assure crash-free operation.

```
---------------------------------------
1000 *= $8000      ; start on $8000 (change if necessary)
1010 .LIS N        ; no listing
1020 .OBJ M        ; object code in memory
1030 :
1040 :             ; This part is essential, no errors allowed
1050 :
1060 LDA $FF00     ; MMU bank configuration register
1070 PHA           ; save it on stack
1080 SEI           ; disable the system interrupt
1090 LDA #$C3      ; store Z-80 opcode - JP
1100 STA $FFEE     ; on boot-link address $FFEE .. $FFF0
1110 LDA #<Z80     ; lobyte Z-80 routine, defined below
```

```
1120 STA $FFEF
1130 LDA #>Z80     ; hibyte Z-80 routine
1140 STA $FFF0
1150 LDA #$3E      ; set the configuration register
1160 STA $FF00     ; with the appropriate value
1170 LDA $D505     ; save the mode configuration register
1180 PHA           ; for later
1190 LDA #$B0      ; and set the mode configuration
1200 STA $D505     ; to 'Z-80 active'
1210 :             ; when this last instruction is executed
1220 :             ; the Z-80 is active and the 8502 is frozen
1230 NOP           ; IMPORTANT: give the 8502 time to start
1240 PLA           ; now back in good ol' 8502 mode
1250 STA $D505     ; restore old mode configuration
1260 PLA           ; and
1270 STA $FF00     ; old memory configuration
1280 CLI           ; enable the system interrupt
1290 RTS           ; end of essential subroutine
1300 :
1500 :             ; Routine for the Z-80
1510 :
1520 Z80           ; Label for the Z-80 routine (s.a. above)
1530 .MOD 1        ; Z-80 opcodes now valid in assembler
1540 LD A, $3F     ; change value of configuration register
1550 LD ($FF00), A ; to the appropriate value
1560 :
1570 :             ; *** sample program
1580 :             ;     - clear VIC-II graphic screen
1590 :
1600 LD A, $00     ; Fill byte $00 on the first
1610 LD ($2000), A ; location of graphic screen ($2000)
1620 LD HL, $2000  ; load HL with address value $2000
1630 LD DE, $2001  ; load DE with address value $2001
1640 LD BC, 7999   ; load BC register with 7999
1650 :             ; (number of bytes to fill)
1660 LDIR          ; execute (HL) -> (DE), 7999 times
1670 :
1680 :             ; *** end of sample program
1690 :
1700 :             ; Crucial for the switch back
1710 :
1720 JP $FFE0      ; jump to bootlink routine in the Z-80 ROM
1730 :             ; i.e. switch 8502 on, and Z-80 off
---------------------------------------
```

Don'ts and do's
------ --- ----

Don't write

```
---------------------------------------
* = $1300
---------------------------------------
```

but write

```
---------------------------------------
*= $1300
---------------------------------------
```

Always place spaces between opcodes and their operands:

```
---------------------------------------
LDA#$00
---------------------------------------
```

this is wrong, it should be typed like this:

```
---------------------------------------
LDA #$00
---------------------------------------
```

Try to put as much of the source code in one file as possible. This safes time during the assembly, because fileloading and saving can be timeconsuming on a CBM machine. Merge the different parts in one major source code file. You can then easily fine-tune the whole program.

Try to only use standarized extensions for the files:
1. no extension or .BIN for the object file
2. .LST for a BASIC listing with source code
3. .SYM for a file with the symbol table
4. .SRC for a formatted source listing

Use as less comment in your own work source listing as possible. Rather than commenting inside your file, use external commenting. Extensive commenting 'chews up' memory and takes a lot of assembly time.

When documenting the final release (obligatory when you write for someone else) you can best make two files: one for assemble with as little comment as possible, and a formatted print-out of that same file to disk (the .LIS source code dump), which you extend with comments in a wordprocessor.

It is important in multi file assembly to only give the start address at the beginning of the first link file. Also the .OBJ statement has to be in the second line of the first link file. Other constructions might confuse the assembler.

```
---------------------------------------
END OF DOCUMENT
```