ZASM - The RML Z80 Assembler

Distributed by Lifeboat Associates
164 West 83rd Street
New York, NY, 10024

R M L A B S O L U T E A S S E M B L E R

SUMMARY

The RML Assembler, written for the 380Z/280Z, is a program which translates instructions written in Zilog's machine code mnemonics into the equivalent binary codes which actually run on the machine. This resultant "object" code is absolute, that is, it is assembled to run in a predefined area of memory.

Like most assemblers, the translation is a two-pass process; the "source" program which forms the input to the assembler is presented twice. During the first pass, only error messages are output; the assembler constructs a table of user-defined symbols and labels. During the second pass, values from this table are combined with the incoming source text to form the object output. At the same time the source program can be listed and a table of symbols can be produced. The assembler outputs a message for each error that is detected, indicating the type of error and the line number and context in which it occurred. A cumulative error count is also available.

The assembler includes a number of special features which are intended to streamline the task of preparing machine language programs on cassette-based systems. Such programs can be readily produced on a system consisting only of a CPU with 16K memory, Video Terminal, Keyboard and two Cassette Recorders, yet optional devices such as a printer are used if available and the assembly of large programs is straight forward.

One such feature is that the assembler reads the source text into

memory during the first pass. If enough storage is available, subsequent passes can be made using this stored copy, thus bypassing the relatively slow process of re-reading the source from cassette. If the program is too large for it all to be stored in memory it can be assembled on a line by line basis in the conventional manner. With a 16K system, approximately 8K bytes are available for the storage of source text.

The "second pass" can be made any number of times, allowing the various outputs to be stored as required. Thus one might first perform Pass 2 with output directed to the screen to check for errors, then again with object directed to cassette (and the listing suppressed) to obtain a copy of the binary output, and finally with listing directed to the printer (and the object output suppressed) to obtain a "hard copy" of the source text and symbols table.

The assembler contains its own text editor, allowing source programs held in memory to be corrected or modified. This enables the user to put right typing errors and omissions during the course of assembly without the necessity of reloading the full Text Editor. The corrected source can be saved. The assembler's editor employs a subset of the commands available in RML's full Text Editor (TXED), allowing the user to switch between the two without confusion. Where large programs are being assembled, the source can be corrected during Pass 1 if necessary. The resident editor greatly speeds up the process of program development on a small system.

The distributed version of the assembler interfaces to the COS Monitor and Cassette File System. Provision has been made for the user to incorporate his own non-standard device driver routines; these can range from a simple teletypewriter handler to a full two-way link allowing the assembly of source held

on files in a larger host machine.

Source program syntax is virtually identical to that defined in the Z80 Assembly Language Programming Manual. Significant extensions include the ability to enter data in 'free format', which considerably simplifies the process of defining data tables, and allowing 8 bit arithmetic operators to be followed by one or two operands (e.g. ADD A,B and ADD B are equivalent). The expression analyser allows a full range of arithmetic and logical operators. Note, however, that neither macros nor conditional assembly are supported at present.

PAGE LEFT INTENTIONALLY BLANK

INTRODUCTION

The RML Z80 Assembler (ZAS) makes use of the normal Zilog mnemonic conventions. Any program written for the Z80 instruction set should assemble with little or no modification. The user is referred to the Zilog publication:

'Z80 Assembly Language Programming Manual'.

This describes the syntax of the assembler in detail and gives a comprehensive description of each instruction. The following notes are intended to supplement this manual; any differences between ZAS and Zilog's Assembler (on which the manual is based) are described in the following sections.

An assembler is a program which translates instructions written in machine code mnemonics into the equivalent binary codes which actually run on the machine. The text which forms the input to the assembler is usually referred to as the 'source' program and the machine codes that are output are called the 'object' code. ZAS assembles absolute object code, that is, the areas of memory in which this object code will run are defined by the user by special instructions embedded within the source text. Since the areas of memory in which the object code is intended to run usually overlap the memory locations used by the assembler, it is necessary to save the object on an intermediate medium, such as cassette tape; when assembly is complete the object code can be loaded into memory ready to run. With some assemblers the decision as to where in memory

the resultant object code will reside can be postponed until the loading phase; in such a case the assembler is said to produce relocatable code. ZAS does not produce relocatable code at present.

Like most assemblers, ZAS carries out the translation from source to object by means of 2 'passes'. During a pass an assembler reads the whole of the source text; thus a 2 pass assembler such as ZAS reads the source text twice. Operations carried out during the two passes are as follows:

Pass 1

The source text is read and a table is constructed containing the values of all user defined symbols. Error messages may be generated but there is no other output.

Pass 2

The source text is read again to produce three types of output:

1. Listing file
2. Object code
3. Error messages

The listing file consists of the original source text alongside the generated code. The format is

```
nnnn  bbbb...  llll.pp   source text
```

where

nnnn = Hexadecimal address at which generated code will be loaded.

bbbb... = Instruction coding generated for current line of source text.

llll	=	Line number
pp	=	Page number
source text	=	One line of source text copied exactly from the input.

The object code can be selected either to be in industry standard format (the 'Intel' format) or to be in RML binary format ready to be loaded with the COS L command.

The error messages are described in detail later in this manual.

The assembler may be used in the conventional manner where source text is read directly from an input device. Alternatively text may be loaded into memory and then assembled. If enough storage is available to hold the whole of the source text, the second pass can be made using this stored copy in memory, thus bypassing the relatively slow process of rereading the source from a cassette file. Should errors occur, there is a built-in editing facility, allowing corrections to be made to the stored copy of the source program. The program may then be reassembled. The corrected source text can be saved.

Programs too large to fit into memory in their entirety may be read in on a page by page basis, where a page is a number of lines of source text delimited by a form feed character. The input device or file can be reselected at any stage during this process allowing text modules to be combined. Errors encountered during pass one may be corrected and the source reassembled page by page without the need to restart assembly. Again these features are intended to speed up the process of program development for users whose systems have limited input/output capability, such as audio cassettes, where continual loading of assembler and editor and the creation of updated files can prove a very time consuming process.

Using a machine with 16K bytes of memory, about half is available for storing source text.

INPUT/OUTPUT

ZAS accepts source text from one input stream. Output is sent to three output streams, identified by the numbers 1 to 3. Output streams are associated with the following information:

STREAM 1 Listing file (pass 2 only)
Symbol table listing.
Output of source text from memory (see SAVE option).

STREAM 2 Error messages produced on pass 1 or 2.

STREAM 3 Hexadecimal object code (pass 2 only).

Input and output streams can be connected to any of the I/O devices associated with the system. ZAS can cope with up to eight input and eight output devices which are associated with the numbers 0 to 7. About half the I/O devices have been allocated within ZAS; users may add their own device handlers if they wish.

Input Device Codes

Code	Device
0	Memory
1	Keyboard
2*	Cassette File System
3*	Teletype (using RML SIO2 or SIO3 interface)
4 - 7	Unallocated

* Input from these devices is automatically copied into memory during pass 1 until the available storage is exhausted.

It is possible to assemble a program typed directly in from the keyboard; however this is not recommended (and such input is not copied into memory). Users are advised to use the resident editor to create such programs and then to assemble them using input from memory.

Output Device Codes

Code	Device
0	Null device
1	VT screen
2	Cassette File System
3	Teletype (using RML SIO2 or SIO3 interface)
4 - 7	Unallocated

The 'null' device is provided as a convenient way of suppressing unwanted output.

Whenever the assembler is restarted at its cold start address, it is set up to read from memory and all output is directed to the VT screen.

ASSEMBLER SYNTAX

Various versions of Z80 assemblers have had somewhat differing conventions regarding the format of instructions. The following sections are intended to clarify any differences

between the RML assembler and the Zilog (or Mostek) manual:

Instruction Format

The only delimiter allowed between the two parametets of an instruction is a comma. Thus

```
LD A,B
```

is correct, whilst

```
LD A B
```

will result in an error message.

8 bit Arithmetic and Logical Group

ZAS accepts either a one parameter form of these instructions where the destination is assumed by context to be the A register, or a two parameter form where 'A' is explicitly included. All the following examples are acceptable:

ADD A,C	or	ADD C
ADD A,24		ADD 24
CP A,20		CP 20
XOR A,C		XOR C

Relative Jumps

Examples of these are:

```
JR LABEL
JR Z,LABEL
DJNZ LABEL
```

The assembler constructs the relative offset jump of

```
LABEL -($ + 2)
```

Note that Decrement and Jump if non zero is written as above and not as DJ NZ,LABEL.

Interrupt Mode Instructions

These are written IM0, IM1 and IM2, without spaces preceding the digit.

Additional Instruction Names

Two additional names have been introduced to exploit features inherent in the COS Monitor:

Relative Procedure Call:

```
CALR LABEL
```

This effectively performs a procedure call to LABEL, where LABEL is within +129 to -126 locations of the call (the same construction as with a JR instruction). This form of procedure call may be exploited in writing position independent code. It is equivalent to

```
RST 20H
LABEL-$-1
```

Emulator Trap:

```
EMT VALUE
```

This is equivalent to

```
RST 38H
VALUE
```

Label

A label is composed of a string of one or more characters of which the first six must be unique. The permitted characters are

Letters, Digits and Period (.).

The first character must not be a digit. The assembler will distinguish between upper and lower case letters. The reserved system defined names use only upper case letters.

Labels must be followed by a colon (:).
Multiple labels at a given statement are allowed.

Assignments

A user defined symbol can be given a value by assignment of the form:

```
NAME EQU expression    or
NAME  =  expression
```

where NAME satisfies the above conditions for a label but does not have a terminating colon. The expression on the right hand side is evaluated during Pass 1 and thus cannot include any forward references. It is worth noting that short names use less space in the symbol table than long names; this may become significant where space is at a premium.

Number Bases

The assembler will accept numbers in several bases viz. binary, octal, decimal and hexadecimal. Numbers must always start with a digit (leading zeroes are sufficient) and may be followed by a single character to signify the base of the number, as follows:

B	Binary
O or Q	Octal
D or .	Decimal
H	Hexadecimal

If no base is specified, the current default will be used. This is initially set as Decimal at the start of each pass. The default radix may be changed to Octal by the pseudo-op instruction:

```
RADO    ; change default radix to Octal.
```

The default may be changed back to decimal by a pseudo-op call of the form:

```
RADD    ; change default radix to Decimal.
```

For example

```
DEFB 44    ; = 44 decimal
RADO
DEFB 44    ; = 36 decimal = 44 octal
RADD
DEFB 44    ; = 44 decimal
```

Comments

The occurrence of a semi-colon (;) indicates that the remainder of the line is to be treated as a comment, to be ignored by the assembler. The only exception is when a semi-colon occurs in the context of a text string.

Expressions

An expression may consist of either a single term (unary) or a combination of terms (binary). It consists of a series of valid arguments connected by operator symbols. The expression is evaluated from left to right in an order determined by the priorities of the operators concerned. Parentheses can be used to ensure correct grouping of terms. Note however that as a parameter to an instruction, an expression wholly enclosed in parentheses will be interpreted as a memory address. The arguments of an expression may be

1. A constant (eg. 0AH)
2. A user defined value, appearing elsewhere on the left hand side of an assignment.
3. The current location counter, denoted by the symbol $
4. An ASCII character (eg. 'A').

The operators are

Symbol	Operation	Priority	
+	Unary plus	4	Highest priority
-	Unary minus	4	
*	Multiplication	3	
/	Integer Division	3	

Symbol	Operation	Priority	
%	Modulo	3	
+	Addition	2	
-	Subtraction	2	
&	Logical AND	1	Lowest priority
!	Logical OR	1	
?	Logical XOR	1	

Expressions are evaluated using 16 bit arithmetic with no check for overflow except for the case of division by zero, which generates an error message. Any undefined user name encountered is given the value zero. An expression may be used in any context in which a numeric result is expected. The final value of an expression will be checked in context to determine its validity.

Origin Settings

Two pseudo-op instructions are available for this purpose:

1. ORG expr

sets the current location counter to the value 'expr'.

2. DEFS expr

reserves 'expr' bytes, that is, advances the current location counter by 'expr'.

Remember that expressions are evaluated during Pass 1 and thus may not make forward references.

Text Strings

A string of text may be assembled using the pseudo-op instruction:

```
DEFM 'string'
```

The characters enclosed between string quotes (') are assembled as a series of bytes of the corresponding ASCII values. The character uparrow (↑) has a special significance within a text string and should always be considered together with the following character as a single entity.

An uparrow followed by a single quote will assemble the single quote character. For example

```
DEFM 'don↑'t'    ; = don't
```

Two uparrows in sequence assemble a single uparrow. For example

```
DEFM 'x↑↑2+y↑↑2'    ; = x↑2 + y↑2
```

in all other cases the effect of the uparrow is to force the most significant bit of the byte to be set. For example

```
DEFM 'A'     ; = 41H
DEFM '↑A'    ; = 41H + 80H = 0C1H
```

Assembling 8 bit Constants

8 bit constants can be defined using the pseudo-op

```
DEFB 33,A1-8,...
```

This differs from the Zilog definition in that multiple values are permitted. An item terminated by a comma prompts the assembler to look for a further item. Spaces or tab characters may be included after the comma and before the next item.

Assembling 16 bit Constants

16 bit constants can be assembled using the pseudo-op:

```
DEFW A1+30,$+6,...
```

Each item will occupy 16 bits (2 bytes) of memory. As in the case of 8 bit constants, a comma is used to separate items in the list.

Free Format

To aid in the construction of data tables, a 'Free Format' mode is available. This effectively combines any of the terms associated with the pseudo-ops DEFB, DEFW and DEFM. The assembler recognises that a given line is to be interpreted in free format by the first significant character following any label specifications. These characters are:

Digits + - $ (and '

The free format lines may consist of any number of terms each separated by a comma. Terms are assigned a data type according to the following rules:

- Expressions are regarded as byte (8 bit) values.
- Expressions preceded by a hash (#) are assembled as word (16 bit) values.

- Items starting with a single quote (') are assembled as text strings.

Thus

10,20	is equivalent to	DEFB 10,20
#10,#20		DEFW 10,20
'text'		DEFM 'text'.

An example of using free format to construct a table follows:

```
;               NAME            AGE         ADDRESS
TABLE:
                'JOHN',         25,         #1234
                'JILL',         18,         #6666 ....
```

Note that the hash character will echo as £ on the VT screen.

Byte and word entries may contain any valid expression. In the case of byte entries the result must yield a valid 8 bit result. Care should be taken in free format with expressions involving character values. For example

```
'A'+3       (which should yield the value of 44H)
```

In this context the opening text quote will cause the assembler to attempt to interpret the remaining characters as a text string and it will generate an error message on encountering the illegal plus sign following the string. The problem can be avoided by ensuring that the text quote is never the first character. The following examples are all valid:

```
+'A'+3    ; = 44H
3+'A'     ; = 44H
#'A'+3    ; = 0044H
```

END Statement

The final statement of the source text should be of the form

```
END  nn
```

Any text following this statement is ignored by the assembler. On Pass 2 the assembler will generate coding on the object output stream which allows the loader program to perform an automatic start at the memory location defined by the expression 'nn'. If 'nn' is not specified, a value of zero is assumed. Note that this may be undesirable as a jump to location zero has a similar effect to pressing the reset button (for example transfer vectors in RAM are reset to their default values). In the situation where an automatic start is not desired but it is wished to avoid a reset, 'nn' may be specified to yield the value 3; this will start COS at its restart address and is equivalent to typing control-C.

The message 'END OF PASS' is generated at the end of each Pass.

Restrictions

The following are not available at present:

1. Macro facilites
2. Conditional assembly directives
3. Object code is not relocatable
4. Symbols may not be redefined.