ZAP

ZAP: Z-language Assembly Program

Joel M. Berez

Contents

1 Introduction

ZAP is a two (or three) pass absolute assembler for Z-code. The intention is for ZAP to provide relatively low-level support for all features available in ZIP (Z-language Interpreter Program).

A ZAP program consists of a single file containing line-oriented statements. A statement may produce code and/or data or may simply direct the assembly process.

During the first pass, the assembler checks statement syntax, calculates code/data locations, and attempts to resolve symbolic references. During the second pass, code is generated, any error messages are displayed, and an optional listing is produced. Due to the nature of this two-pass process, certain symbol definition restrictions exist that will be explained in the appropriate sections.

An optional prepass may be invoked to build the frequently used word table by omitting the table (`FWORDS`) from the input file. This is a table of substrings that may be inserted into larger strings in an abbreviated form. The assembler will identify any such substrings (defined by `.FSTR`) in strings that it assembles and use the proper format. This optional prepass will search all regular strings, find 32 good choices for substrings, and define them with `.FSTR`s in the `FWORDS` table. See the ZIP manual for further details concerning the string format.

Code that is generated by the assembler is directly executable by ZIP as long as the program follows all necessary conventions for the interpretor. ZAP will to a large extent insure that required data structures are defined, ZIP pointers are initialized, etc. However it cannot be responsible for such implementation dependent restrictions as maximum table sizes.

2 Language Syntax

2.1 Character Set

Except within strings or comments, only the following characters are allowed in a Z program:

```
A-Z             Symbol constituents
0-9,-           Symbol or number constituents
?,#,.           Symbol constituents

<space>,<tab> Ignored except as initial operand prefix
<cr>,<ff>     Ignored
<lf>          End-of-line character

,             General operand prefix
>             Return value operand prefix
/             Branch (on success) operand prefix
\             Branch (on failure) operand prefix
=             Value operand prefix (for assignments)
+             Addend operand prefix (addition of constants)

"             String delimiter

;             Comment prefix
```

2.2 Symbols

Symbols are used to represent values of various types. Because the assembler associates a type with each symbol along with its value when it is defined, no special method is required to specify the type when the symbol is used. However, certain naming conventions are suggested for the convenience of the programmer.

A symbol contains any positive number of characters from the set `{A-Z,0-9,?,#,.,-}`. By convention pseudo-ops, and only pseudo-ops, begin with a period. Other conventions are left to the discretion of the programmer.

Symbols may be global or local depending upon the type of value that is assigned to them. The range of a global symbol is the entire program, while the range of a local symbol is restricted to the function in which it is defined. While local symbols may be reused from one function to another, only constants (always global) may be truly redefined.

The global symbols include two predefined sets. Operators are the "hardware" Z-machine instructions and use the same mnemonics shown in the ZIP documentation. Pseudo-ops are used somewhat like operators, but are simply assembler directives that may or may not generate code/-data. They are explained later in this manual.

Global labels can refer to either global data or to functions. (In fact data, strings and tables, must be defined globally.) These are defined either through the colon-colon construct, by a pseudo-op, or by assignment to another global label. A few special globals are predefined by the assembler (e.g. `VOCAB` refers to the vocabulary table).

Constants refer to arbitrary user-specified values. These are defined by a direct assignment or by a pseudo-op. Unlike other symbols, constants may be redefined at any time.

Global variables are defined by the `.GVAR` pseudo-op within the `GLOBAL` table. They may be used interchangeably with local variables, including the special `STACK` variable, and represent unique data locations within the program.

Local symbols come in two varieties. Local variables are defined by the `.FUNCT` pseudo-op, which defines a function and allocates storage for its arguments and other local variables. These variables are allocated on the stack when the function is called and are for each such call assigned the initial values specified.

Local labels are the targets of branching instructions and are defined by the colon construct or by assignment to another local label.

2.3 Statement Syntax

A statement consists of the following four fields, all of which are optional:

```
<label> <operator> <operands> <comment>
```

A `<label>` is a single symbol terminated by either one or two colons, depending upon whether the symbol is defined to be local or global.

An `<operator>` is a pre-defined symbol from either the set of operators or the set of pseudo-ops. It is terminated by the beginning of another field or by the end-of-line.

The `<operands>` field begins with either a space or a tab, unless it is the first field on the line. It may contain one or more operands. Each operand after the first begins with an operand prefix character. The operands field may be continued to the next line by putting the prefix character on one line and the corresponding operand on the next line. Each line may contain a comment.

The `<comment>` begins with a semi-colon and ends at the end-of-line. It may contain any ASCII characters with the obvious exception of line-feed.

3 Pseudo-ops

3.1 Meta-syntax

Below each of the assembler pseudo-ops is described. The syntax to be used is shown first followed by a description. A pseudo-op statement consists of the pseudo-op name in the operator field followed by zero or more operands. A label may or may not be appropriate and a comment is always allowed.

The meta-syntax used for describing the statement format shows the pseudo-op followed by operand types, enclosed in angle brackets. Square brackets are used to enclose optional operands. Operand types that may be repeated zero or more times are followed by an ellipsis and enclosed in braces.

The following operand types are used:

<**number**> An integer between -32768 and 65536. Numbers larger than 32767 or smaller than 0 may be interpreted as either positive or negative, depending upon the machine instruction.

<**constant**> A fixed user-defined value. May be a `<number>`.

<**short constant**> A `<constant>` with a non-negative value less than 256.

<**long constant**> A `<constant>` that is not a `<short constant>`.

<**symbol**> A symbol as described in section 2.2.

<**pointer**> A symbol that refers to a disk location, probably defined with one of the colon constructs.

<**any**> Any `<symbol>` or `<number>`.

<**string**> Any number of characters of any ASCII value enclosed in double-quotes. A double-quote may be included in the string by using two consecutive double-quotes.

<**short string**> A `<string>` not requiring more than 255 words to represent.

3.2 Simple Data-generation Pseudo-ops

`.WORD <any>{,<any>...}`

Generates the two-byte value of each `<any>`.
For convenience, if `<any>` is written by itself, it will be interpreted as `.WORD <any>`.

`.BYTE <short constant>{,<short constant>...}`

Generates the one-byte values.

`.TRUE`

Equivalent to `.WORD 1`.

`.FALSE`

Equivalent to `.WORD 0`.

3.3 String Handling Pseudo-ops

`.ZWORD <string>`

Generates the four-byte value of `<string>`, left-justified and padded with spaces if necessary. `<string>` may not require more than two words.

`.STR <string>`

Generates `<string>` in two-byte words. The last word has the end-of-string bit set and, if necessary, is padded with shift5 characters. For convenience, if `<string>` is written without an operator, it will be interpreted as `.STR <string>`.

`.FSTR <string>`

Generates a string for the frequently used word table (`FWORDS`). First does a `.STR <string>`, except that `<string>` is not searched for fword substrings. Then adds the string to the table of fword substrings. All `.FSTR`s should be in the 32-word table following `FWORDS::`.

`.LEN <short string>`

Generates a byte containing the number of words required to represent the `<short string>`.

`.STRL <short string>`

Equivalent to:

```
.LEN <short string>
.STR <short string>
```

3.4 Assignment Pseudo-ops

`.EQUAL <symbol>,<any>`

Assigns to `<symbol>` the same value and type as `<any>`. If any is a `<number>`, the type becomes `<constant>`. For convenience, the short form of `<symbol>=<any>` will also be accepted by the assembler.

`.SEQ <symbol>{,<symbol>...}`

The symbols are assigned as constants with sequential values beginning with zero.

3.5 Special Purpose Pseudo-ops

`.TABLE [<number>]`

Declares that a table is being generated. Optionally specifies the maximum length in bytes.

`.PROP <length>,<property>`

Generates a one-byte property header. `<length>` is a `<constant>` between 1 and 8. `<property>` is a `<constant>` between 1 and 31.

`.ENDT`

Ends generation of the current table. If `<number>` was specified in the `.TABLE` statement, ensures that not more than `<number>` bytes have been used.

```
.OBJECT <symbol>,<number1>,<number2>,<object1>,
        <object2>,<object3>,<pointer>
```

Generates an object with the specified elements. `<symbol>` will be the object name and is assigned to the next available object number. `<number1>` and `<number2>` are the flag words. `<object1>`, `<object2>`, and `<object3>` are object symbols refering to the `<loc>`, `<first>`, and `<next>` pointers, respectively. `<pointer>` points to the property table.

Note: all objects must be defined together in the `OBJECT` table.

```
.GVAR <symbol>[=<any>]
```

Defines a new global variable named `<symbol>` and assigns `<any>` as the default value. `<any>` defaults to zero.

Note: all global variables must be defined together in the `GLOBAL` table.

```
.FUNCT <symbol>{,<symbol>[=<any>]...}
```

Begins generation of a function and starts a new local symbol block. The first symbol is the function name. Any other symbols specified become local variables. Default values may be given or will default to zero.

3.6 Flow Control Pseudo-ops

```
.INSERT <string>
```

Logically inserts the contents of file `<string>` into the current program at this point.

```
.ENDI
```

Ends the current `.INSERT` file and returns control to the previous input source. Everything after this in the file will be ignored.

```
.END
```

Signifies the end of the program. Everything after this in the input source will be ignored.

4 Program Structure

4.1 Special Labels

To satisfy the requirements of ZIP, pointers to certain locations will automatically be assembled into a table at the beginning of the program. (See the ZIP manual for the exact format.) To this end, certain symbols must be defined by the user as global labels at the appropriate positions in the program. The following symbols are required:

VOCAB:: Vocabulary table.

OBJECT:: Object table.

GLOBAL:: Global symbol table. This table must be the proper length, as specified in the ZIP manual.

FWORDS:: Frequently used word table. If omitted, will be automatically generated by the assembler.

PURBOT:: Beginning of pure (read-only) data and code.

ENDLOD:: End of the preloaded data and code.

START:: First instruction to be executed when game is started. Must actually point to an instruction, not a function.

4.2 Program Order

The program should be arranged in the following order:

1. `GLOBAL`, modifiable tables, and other impure data.
2. `PURBOT::`
3. `VOCAB`, `OBJECT`, `FWORDS`, and other pure, preloaded tables, strings, and functions.
4. `ENDLOD::`
5. Non-preloaded tables, strings, and functions.
6. `.END`

`START::` must be specified in front of some instruction.