**Language Specification**

This document specifies the 16 bit programming language SCPL. Design is with the Z80 in mind, but it is not limited to that processor.

The documentation is inside-out in design, starting with the simplest element and working out.

Terms

Terms can be one of two things , a constant value or a variable address. The following terms are allowed :

|  |  |  |
| --- | --- | --- |
| Name | Example | Notes |
| Integer constant | -104, 104 | Standard integer constant. Which can be prefixed by a minus sign. |
| Hexadecimal constant | ‘C’ | A 7 bit ASCII value from 32-127. Control characters (e.g. \n \t) are not currently supported. |
| String Constant | “hello !” | A string constant returns the address of an ASCIIZ representation of that string. It is not necessarily unique (e.g. two strings with the same text could use the same address … or not) but should be treated as invariable normally. |
| Identifier | Count.x | Any legal identifier, e.g. one containing the characters a-z, 0-9 \_and . , but must begin with the letters \_ a-z. Case is irrelevant. |
| Identifier Address | &Count.x | The address of the identifier count.x ; this could be a variable or procedure address. |

Identifier Scope

The identifier scope is very lazy. Any variable defined in a procedure is local to that procedure *but* there is only one set of variables for each procedure instance. If you want to use recursion variables will have to be placed on a stack. The reason for this is it means that variables can be loaded and saved using single instructions, not (say) relying on slow loading from IX and IY.

Expressions

Expressions are built up from alternate terms and operators. Note that there is no order of precedence and no parentheses ; evaluation is strictly left to right. All operators are binary.

Operators are, at present.

|  |  |
| --- | --- |
| Operator | Notes |
| + | Addition |
| - | Subtraction |
| \* | Multiplication (all 16 bit values, overflow lost) |
| / | Division (no error if zero) |
| % | Modules (no error if zero) |
| ? | Byte indirect (e.g. a?b reads byte at (a+b) |
| ! | Word indirect (e.g. a!b reads word at (a+b) |
| > < == != >= <= | Tests ; return -1 if the test is true and 0 if it is false. |
| & | ^ | Bitwise operations (and, or, xor) |

Assignments

Assignments are built up of l-expressions and expressions. L-expressions can be any of the following

|  |  |
| --- | --- |
| Example | Notes |
| count.y | A variable |
| a?2 | Store the result in the byte at A+2 |
| b!c | Store the result in the word at B+C |

Code Blocks

Code blocks are defined as either (i) a single instruction ending with a semicolon or (ii) a group of instructions surrounded by curly brackets (like in ‘C’).

Instructions are:

<l-expr> = <expression>;

Byte or word Expression

while (<expression>) <code block>

Conditional loop

if (<expression>) <code block>

Conditional test

for (<expression>) <code block>

Repeat <expression> times with an index counting from <expression-1> down to zero.

<procedure name>(p1,p2,p3) <code block>

Call a procedure. Parameters are stored in HL, DE and BC respectively, to a maximum of 3.

Parameters are all terms.

var <name><[size]>;

Define a variable or block of memory. If a block this is done BCPL style e.g. it’s a reference

Program Blocks

Code blocks are collections of procedures and global variable definitions. Note that any identifiers beginning with an underscore are always local to the module they are compiled in. Variables are always local to the module they are compiled in, even the global ones.

Code Generation

Code generation is based around three registers, the A and B registers – the first term is loaded into A, subsequent into B and binary operations are all A <op> B and an index register which holds the target address for a?4 and b!2 left-expressions.

General

Comments are done using double slashes only. Anything on the line after // is ignored.

Case is ignored *except* in string constants.