Cross-Platform Low Level Language (CPL^3)
Language Reference

Brian Westphal • University of Nevada, Reno – 2003

Comments are denoted by a leading pound sign (#) and continue until the newline character is reached.

Programs are defined by a global variable declaration block followed by at least one function definition. The main function must be called `.func_main`.

Global Variables are defined with the following syntax:

```
vars:(type1)var1,(type2)var2,…,(type3)var3;
```

Global Functions are defined with the following syntax:

```
.func_functionname
{
	return:(type);
	params:(type1)var1,(type2)var2,…,(type3)var3;
	vars:(type1)var1,(type2)var2,…,(type3)var3;

	statements
}
```

If a defined function does not return a value, then `return:void;` should be used (parentheses are not needed as void is not considered to be a formal type). See *User Functions* for more information.

Local Functions are defined in the same way as global functions, except `.func_` should be replaced with `.localfunc_`.

Variable Types are nearly always enclosed in parentheses; the only exception is with the `func_types:` directive. The following basic types are available: label, integer, floating-point, character, and various types of pointers (the data of which is an address).

Labels are defined with a leading dot (`.`) followed by a name consisting of letters and/or underscore characters. They are used only for function names and formal labels (labels cannot be stored as values of variables). Much like with C/C++, labels are only accessible within their current block.

Integers are defined with the following syntax:

1. `(subtypeibitdepth)numericvalue`
2. `(subtypeibitdepth)var`

The subtype is optional and may be denoted by a ‘u’ or ‘s’ character (meaning unsigned or signed). The bit-depth is also optional and may be denoted by an integer value of 8, 16, 32, or 64.

The bit-depth describes the size of the resulting integer variable.

Syntax Opt 1 If a value is specified (this cannot be done for a declaration) then an integer value should be supplied. An integer value may contain a leading sign (+/-) followed by any number of decimal digits (0-9). See *Scientific Notation* for more information.

Syntax Opt 2 A variable name can only be specified in declaration mode. See *Variable Names* for more information.

```
(i32)1
```

Floating-Point Numbers are defined in the same way as integers except that the `i` should be replaced with an `f`. Floating-point bit-depths may be either 32 or 64. It is important to note that most modern architectures do not support unsigned floating-point numbers. A floating-point value may contain a leading sign (+/-) followed by any number of decimal digits (0-9), followed by a dot (`.`), followed by any number of decimal digits. See *Scientific Notation* and *Variable Names* for more information.

```
(f32)-3.1415927
```

Characters are defined similarly to integers and floating-point numbers:

1. `(c`*`bitdepth`*`)`*`hex`*
2. `(c)'`*`literal`*`'`

The bit-depth is optional for Syntax Option 1 and not allowed for Syntax Option 2 (Syntax Option 2 assumes a bit-depth of 8).

Syntax Opt 1 If a hex value is specified, it represents the ASCII (for 8-bit) or UNICODE (for 16-bit) value of a character. Hex values are defined with at least one hexadecimal digit (0-F).

Syntax Opt 2 If a character literal is specified it is enclosed in single-quotes.

```
(c16)0020
(c)'X'
```

Pointers are available for integer, floating-point, and character types. A void pointer type is also available. Pointers are defined by appending an asterisk to a type specification. The value of a pointer is an address. A literal address may be defined using the following syntax:

`(`*`type`*`)a`*`address`*

The address portion can be any mix of hexadecimal digits (0-9) and colons (`:`) used for readability.

```
(c8*)a0000:B800
(void*)a0000:A000
```

Arrays are defined much as they are in C/C++. Square brackets enclose the size of an array during declaration and the index of an element during use. Unlike C/C++ however, only single dimensional arrays are allowed and all sizes must be literal numeric values. All indices must be either literal numeric values or variables.

```
vars:(c)$string[1024];

MOV $string[$index] (c8)0000;
```

Strings are also defined similarly to the way they are defined in C/C++. Double-quotes enclose a string which may contain the escape sequence `\"` used to denote a literal double-quote. Strings in CPL[3] are valid only when used as the second parameter of a `MOV` function. One should note that the space for strings is not automatically allocated, except in temporary space, so in the following example the variable `$string` must have memory allocated for it prior to the `MOV` function call.

```
MOV $string “Hello, World!”;
```

Scientific Notation may be used to define integer and floating-point literal values. Integer scientific notation may include a leading sign (+/-) and always includes any number of decimal digits (0-9) followed by an `E`, followed by an additional optional sign (+/-), followed by any number of decimal digits. This format represents

the typical form of scientific notation as is used in C/C++. Floating-point scientific notation is also similar in this way.

```
(i32)10E6
(f32)1.0E-6
```

Variable Names are defined with leading dollar signs (`$`) followed by an underscore or letter, optionally followed by any number of underscores, letters, or digits.

Statements may include the following if..else structures, while and do..while loops, built-in function calls, user function calls, and directives.

if...else structures are defined with the following syntax:

```
if condition
{
     …
}
else
{
     …
}
```

The `else` portion is optional. The condition is made up of a variable or literal followed by a relational operator, followed by another variable or literal. See *Relational Operators* for more information.

```
if $lhs < $rhs
{
     RET $lhs;
}
```

while loops are defined with the following syntax:

```
wloop condition
{
     …
}
```

The condition has the same format as with if...else structures.

```
wloop $index < (ui32)10
{
     ADD $index $index (ui32)1;
}
```

do...while loops are defined similarly to while loops except that `wloop` is replaced with `dwloop`.

See *Function Calls* for more information on built-in function calls and *User Functions* for more information on user function calls and directives.

Relational Operators include the following: `<`, `<=`, `==`, `>=`, `>`, and `!=` (as defined in C/C++).

Function Calls. CPL[3] supports the following built-in function calls: `ADD`, `SUB`, `MULT`, `DIV`, `REM`, `NEG`, `EQ`, `GT`, `GE`, `LT`, `LE`, `NE`, `AND`, `NOT`, `NOR`, `OR`, `XOR`, `SHL`, `SHR`, `BAND`, `BOR`, `BXOR`, `RET`, `CALL`, `JMP`, `JEZ`, `MOV`, `HALT`, `ALLOC`, and `DEALLOC`.

ADD	Adds two values together and stores the result in a specified location. CPL3 `ADD dest source source;` C++ `dest = source + source;`
SUB	Subtracts one value from another and stores the result in a specified location. CPL3 `SUB dest source source;` C++ `dest = source - source;`
MULT	Multiplies two values together and stores the result in a specified location. CPL3 `MULT dest source source;` C++ `dest = source * source;`
DIV	Divides one value over another and stores the result in a specified location. CPL3 `DIV dest source source;` C++ `dest = source / source;`
MOD	Calculates the integer remainder of dividing one value over another and stores the result in a specified location. CPL3 `MOD dest source source;` C++ `dest = source % source;`
NEG	Multiplies a value by negative one and stores the result in a specified location. CPL3 `NEG dest source;` C++ `dest = -1 * source;`
EQ	Determines the equality of two values and stores the result in a specified location. CPL3 `EQ dest source source;` C++ `dest = source == source;`
GT	Determines if the one value is greater than another and stores the result in a specified location. CPL3 `GT dest source source;` C++ `dest = source > source;`
GE	Determines if the one value is greater than or equal to another and stores the result in a specified location. CPL3 `GE dest source source;` C++ `dest = source >= source;`
LT	Determines if the one value is less than another and stores the result in a specified location. CPL3 `LT dest source source;` C++ `dest = source < source;`
LE	Determines if the one value is less than or equal to another and stores the result in a specified location. CPL3 `LE dest source source;` C++ `dest = source <= source;`
NE	Determines the inequality of two values and stores the result in a specified location. CPL3 `NE dest source source;` C++ `dest = source != source;`
AND	Determines the result of the logical and operation on two values and stores the result in a specified location. CPL3 `AND dest source source;` C++ `dest = source && source;`
NOT	Determines the result of the logical not operation on a values and stores the result in a specified location. CPL3 `NOT dest source;` C++ `dest = !source;`
NOR	Determines the result of the logical nor (not-or) operation on two values and stores the result in a specified location. CPL3 `NOR dest source source;`

	C++ `dest = !(source || source);`
OR	Determines the result of the logical or operation on two values and stores the result in a specified location. CPL3 `OR dest source source;` C++ `dest = source || source;`
XOR	Determines the result of the logical xor (exclusive or) operation on two values and stores the result in a specified location. CPL3 `XOR dest source source;` C++ `dest = (source || source) && !(source && source);`
SHL	Shifts a value to the left by a specified number of bits and stores the result in a specified location. CPL3 `SHL dest source source;` C++ `dest = source << source;`
SHR	Shifts a value to the right by a specified number of bits and stores the result in a specified location. CPL3 `SHR dest source source;` C++ `dest = source >> source;`
BAND	Determines the result of the bitwise and operation on two values and stores the result in a specified location. CPL3 `BAND dest source source;` C++ `dest = source & source;`
BOR	Determines the result of the bitwise or operation on two values and stores the result in a specified location. CPL3 `BOR dest source source;` C++ `dest = source | source;`
BXOR	Determines the result of the bitwise xor (exclusive or) operation on two values and stores the result in a specified location. CPL3 `BXOR dest source source;` C++ `dest = source ^ source;`
RET	Returns a value from a function. See *User Functions* for more information. CPL3 `RET source;` C++ `return source;`
CALL	Calls a specified user function. See *User Functions* for more information. CPL3 `CALL source;` C++ A standard function call.
JMP	Jumps to a specified label. CPL3 `JMP label;` C++ `goto label;`
JEZ	Jumps to a specified label if the specified value is zero. CPL3 `JEZ label source;` C++ `if (source == 0) goto label;`
MOV	Copies one value into a specified location. CPL3 `MOV dest source;` C++ `dest = source;`
HALT	Exits the program with a zero status. CPL3 `HALT;` C++ `exit (0); //requires stdlib.h`
ALLOC	Allocates a block of memory and returns the address. CPL3 `ALLOC dest source source;` C++ `dest = calloc (source, source); //requires stdlib.h`
DEALLOC	Deallocates a block of memory. CPL3 `DEALLOC source;`

```
C++ free (source); //requires stdlib.h
```

User Functions are defined as mentioned in *Functions*, however there are a few specific items to mention when defining and calling user functions:

1. Nested functions are permitted.
2. Recursion is also permitted.
3. Before calling a function, the parameters must be setup (even if no parameters will be used).

Setting up parameters can be done using the `func_types:` directive. The syntax for this directive is as follows:

```
func_types:((returntype):type1,type2,…,type3);
```

This will setup an appropriate number of parameter variables named `$1`, `$2`, …, `$3` as well as a return value named `$return`. If the function does not return a value then `void` should be used as the return type (without being enclosed in parentheses). If the function does not take parameters, the parameter type list should be marked as `void`. These variables are valid until one statement (or the end of the block) after the function call. That is, if the return value of the function must be stored, it should be stored immediately after calling the function.

```
func_types:((i32):i32);
MOV $1 (i32)5;
CALL .func_factorial;
MOV $tmp $return;
```

Additional Features include a size of operator, address of (`&`) operator, and dereference (`>`) operator. Following are examples of use for each:

```
MOV $tmp sizeof(i);
MOV $tmpaddr &$tmp;
MOV $tmp >$tmpaddr;
```

The following covers several examples of CPL[3] code (infused with C++ code for reference).

Factorial:

```
vars:;

#int factorial (int number)
.func_factorial
{
     return:(i);
     params:(i)$number;
     vars:(i)$temp;

     #if (number <= 1) return 1;
     if $number <= (i)1
     {
          RET (i)1;
     }

     #return number * factorial (number – 1);
     SUB $temp $number (i)1;
     func_types:((i):i);
     MOV $1 $temp;
     CALL .func factorial;
```

```
        MULT $temp $number $return;
        RET $temp;
    }

    #int main ()
    .func_main
    {
        return:(i);
        params:;
        vars:;

        #return factorial (5);
        func_types:((i):i);
        MOV $1 (i)5;
        CALL .func_factorial;
        RET $return;
    }
```

Factorial with Local Functions:

```
    vars:;

    #int main ()
    .func_main
    {
        return:(i);
        params:;
        vars:;

        #int factorial (int number)
        .localfunc_factorial
        {
            return:(i);
            params:(i)$number;
            vars:(i)$temp;

            #if (number <= 1) return 1;
            if $number <= (i)1
            {
                RET (i)1;
            }

            #return number * factorial (number – 1);
            SUB $temp $number (i)1;
            func_types:((i):i);
            MOV $1 $temp;
            CALL .func_factorial;
            MULT $temp $number $return;
            RET $temp;
        }

        #return factorial (5);
        func_types:((i):i);
        MOV $1 (i)5;
        CALL .func_factorial;
```

```
        RET $return;
    }
```

Insertion Sort (function only):

```
vars:;

#void sort (int* data, int* sorted, int length)
#int max = -1;
#int maxindex;
.func_insertionsort
{
     return:void;
     params:(i*)$data,(i*)$sorted,(i)$length;
     vars:(i)$index,(i)$subindex,(i)$max,(i)$maxindex,(i*)$tempOf
fset;

     #for (int index = 0; index < length; index++)
     MOV $index (i)0;
     wloop $index < $length
     {
          #int max = -1;
          MOV $max (i)-1;

          #for (int subindex = 0; subindex < length; subindex++)
          MOV $subindex (i)0;
          wloop $subindex < $length
          {
               #if (data[subindex] > max)
               ADD $tempOffset $data $subindex;
               if >$tempOffset > $max
               {
                    #max = data[subindex];
                    MOV $max >$tempOffset;
                    #maxindex = subindex;
                    MOV $maxindex $subindex;
               }

               ADD $subindex $subindex (i)1;
          }

          #sorted[index] = max;
          ADD $tempOffset $sorted $index;
          MOV >$tempOffset $max;
          #data[maxindex] = -1;
          ADD $tempOffset $data $maxindex;
          MOV >$tempOffset (i)-1;

          ADD $index $index (i)1;
     }
}
```

Annotations

04/13/2003 – Pointer chains can now be used. Following is an example:

```
.func_main
{
     return:(i);
     params:(i)$argv,(c**)$argv;
     vars:;

     ...
}
```