# XCPL Language Description

XCPL is a 6502 based cross-compiled programming language. It runs code in a virtual machine called “Sour 16”. This is a virtual machine not dissimilar to Sweet 16, but it is largely unusable for general programming. 6502 code routines are supported.

## Code

Code ignores case, *except* in quoted strings and character constants. # is a comment and marks the rest of the line as to be ignored.

The general syntax is like that of C, with semicolons used to end lines and curly brackets { } to group multiple commands together. However it shares BCPL typelessness and indirection (though it supports byte and word indirection). The latter is useful for ASCII strings and dense structures but doesn’t offer significant speed increases in the VM.

A major variation is that all local variables are static

## Types

Types are all 16-bit unsigned integers. It is possible to access byte data using indirection, but not have byte variables.

## Term

A term can be one of the following :

* A decimal integer (192)
* A hexadecimal constant prefixed with $ ($FA74)
* A character constant (‘x’)
* A constant which is the address of an identifier prefixed with @ (@count)
* A Double-Quoted String. This evaluates to an address of the string in memory, which is an ASCIIZ string (e.g. the last character is byte $00)
* An identifier representing a variable.
* An expression in parenthesis ( (count+7) )
* A negated term (-count)
* A byte indirection. This specifies the byte value at an address (?$F4) – this is equivalent to PEEK($F4)
* A word indirection. This specifies the word value at an address (?$F7) – this is equivalent to DEEK($F7) in some BASICs.
* A table. These are created using the keywords bytes and words – which take a list of constants (integer, character, string, identifier address, table) and puts them in a table of bytes or words. E.g. words(1,10,100,1000,10000) creates a 10 byte 5 word table with the powers of 10 in it.

## Expressions

Expressions are chains of terms separated by binary operators (the terms can include parenthesised expressions). All operators operate in the same way irrespective of what data they represent.

The following operators are supported :

* Standard arithmetic operators + - \* / % (% is modulus). These are all unsigned arithmetic. Division and Modulus by zero is indeterminate but does not (currently) cause an error. If you want to multiply or divide by a power of 2 use the << and >> operators which are faster as the 6502 does multiply/divide the hard way.
* Binary operators & (and) | (or) ^ (exclusive-or). These are binary operators *not* logical operators. If you write 3 & 7 it is not the same as (3 <> 0) & (7 <> 0)
* Comparison operators == <> >= <= > and <. These all operate in an unsigned arithmetic mode. They evaluate to $FFFF if true and 0 if false. In comparisons (e.g. while and if tests) the value does not matter other than it is non-zero ; so, you can use if (a) as a shorthand for if (a<>0)
* Shift operators << and >>, do a logical shift left or right of the data (e.g. count << 2 is the value in count shifted left twice). Zeros are shifted in.
* Indirection operators ? and !. These are like the term indirection operators, except the address is the sum of the left- and right-hand side. So, if a = 10 then a!4 reads the word at offset 10+4, 14. Note that in BCPL n!1 reads the nth word, in XCPL it reads the nth byte. So, if you use word data in structures, it must be done in steps of 2.

## Precedence

The precedence rules are as follows (lowest to highest) :

* Logic operators : & | ^
* Comparison operators : < == > <= <> >=
* Additive operators : + -
* Multiplicative operators : \* % >> <<
* Indirection operators : ! ?

## Assignment

Assignment is of the form <lexpr> = <expression> . The left-hand side can be either a variable, or one of the four types of indirection : ?term !term term!term term?term

Generally ?term and !term are used for accessing specific memory locations – control registers and so on, and the binary form is used for structures. One may allocate a block of memory to an object and access it indirectly e.g. if missile points to a block of 4 bytes then one can say:

missile!0 = x;

missile!2 = y;

This is the same as the BASIC DOKE missile+0,x:DOKE missile+2,y

(DOKE is a double byte POKE, so missile?5 = count is POKE missile+5,count)

## Variables

Variables can be declared either globally or locally. Local variables are declared in a procedure body.

The var command is the keyword var followed by a list of commas separated variables. Each variable can have a block of memory allocated by specifying the bytes in square brackets.

var count, scores[4];

declares two variables, one called ‘count’, and one called ‘scores’. The scores variable is initialised to the address of a block of 4 bytes.

All locals are static. This means that if you allocate memory to a local variable it will have that value when the routine is first call and will maintain it over subsequent invocations ; memory is only allocated *once*. Code *can* recurse but you must handle the variables yourselves.

Similarly, uninitialized variables or “array elements” are cleared when the program is first run, but *not* cleared on each invocation of a procedure, because they are static. *Everything* is static. It’s a 6502.

## Increment and Decrement

Variables *only* can be incremented or decremented using ++ and – e.g.

count++;

score--;

This is a command *not* a term. You cannot write count = n++;

## Procedure calls

There are no functions. Procedures are called with a list of values, up to a maximum of 12. You shouldn’t have that many anyway. The compiler does keep track of the number of parameters in a call.

Parameters are all word values, but you can pass references using @ e.g.

get Square(@v,12);

which gives a method for returning values (other than returning in a static ; it’s arguable that you could just have a global variable called ‘RETURN’ as you can’t nest calls ; you just have to deal with the return immediately. You pays your money … or you would if this wasn’t MIT licensed.

## Repeat Loops

Repeat loops have the following form

repeat <code block> until <expression>

The loop will terminate when the expression is non-zero, it does not have to be a comparison expression.

var count;

count = 10;

Repeat { count = count – 1;print.hex(count) until (count == 0);

## While Loops

While loops have a very similar form to repeat loops, but the test is at the top:

while <expr> <code block>

e.g.

var n;

n = 100;

while (n)

{

n--;

}

## Do Loop

Do is XCPLs equivalent of for loops in C. It has the same basic function – execute code a fixed number of times but is significantly simplified. It has two forms, one has an index variable, the other does not.

Do (<count>) <code block>

Do (<count>,<variable>) <code block>

The loop is executed <count> times ; count is any valid expression. If the variable is defined, then it contains the value of that expression-1 to 0.

Do (10) { Print.String(“Hello, world”); }

Do (4,idx} ( Print.Hex(idx); }

In the second example idx will have the values 3,2,1 and 0 in the loop. The first prints “Hello, world” ten times.

## If Statements

If statements have two forms, the same as those in C

If <expr> <code block>

If <expr> <code block> else <code block>

e.g.

if (n % 2 == 0) { print.string(“Even”); } else { print.string(“Odd”); }

## Macros

Macros are textual substitutions rather like #define in C, except they do not allow parameters. The syntax requires the define keyword to be followed by a quoted string e.g.

define count “4”

will cause the identifier count to be replaced by 4 in all code.

## Program Files

Program files just consist of lists of procedure definitions and variable definitions. A procedure definition is like C but has no type (there is only one type). The program is started by running the last defined procedure

var n;

define c.to.print “42”

lotsChar(n,c) {

do(n) { print.char(c); }

}

main() {

do(40,n) {

lotsChar(n+1,c.to.print);

print.char(13);

}

}

This is a complete working program. Note that n would normally be declared locally in the main() routine , it’s just moved out of there to show the syntax.