The Moncky programming language

Kris Demuynck

17th August 2022

Abstract

This document describes the Moncky programming language. It was created to make programming the Moncky computer easier. The home of the Moncky project is

https://hackaday.io/project/181269-the-moncky-project

and

https://gitlab.com/big-bat/moncky

for all source code.

Copyright

Kris Demuynck

The Moncky programming language

Contents

Chapter 1

Introduction

Programming the Moncky computer in assembly language is quite tedious. Therefore a higher level programming language was created. This is the Moncky programming language. It is loosly based on Pascal and C. The main goal was to create an easy to learn language that can be parsed in a simple way. The compiler is able to create very compact assembly from the source.

Here are some design considerations when the Moncky programming language was constructed:

- comments can be written in C-style /* */ or with //
- there are only integer data types (int16, int32, uint16 and uint32). The types uint8 and int8 are still available but will be deprecated in the future. Floating point values are stored in 32-bit integers. Calculations with floating point values have to be programmed (they are available in the firmware). Characters can be either 8-bit or 16-bit integers, depending on the choice of the programmer. Booleans are stored as 16 bit integers. The value of 'false' is 0 and the value of 'true' is equal to -1.
- the compiler doesn't check the assignment between different types. If the programmer assigns a uint32 to a uint16, 16 bits are lost, without warning.
- there is no for-loop (for the moment). There is only a while-loop
- the if- and while-statements are always followed by a compound statement between curly brackets. The programmer is not allowed to put one single statement
- the if-statement does not need parantheses except when there is an expression
- curly braces ({}) are used to indicate compound statements
- there is no need to put a semicolon at the end of a statement. The compiler knows when a statement is done
- comma's are not needed to seperate parameters or values in an array
- expressions always have to be between parantheses, except if the expression is a constant or an identifier. The order of the operations is always defined by parantheses
- boolean operators will render 0 (false) or -1 (true) as a result. In this way there is no need to have a separate operator for booleans. And booleans can be used as masks.
- all firmware functions are available to the programmer
- all local variables must be declared at the start of the function. This enables the compiler to optimise the usage of the stack

- there are constants, records, arrays, and pointers
- only global variables can have an initial value. Local variables must always be assigned separately (at the moment)
- there are no pointers to functions possible (at the moment)
- array literals can only contain primitive values
- arrays of int8 or uint8 are stored compact in 16-bit numbers (LSB first)
- there is currently no support for modules. All code needs to be in one source file.

The home page for the Moncky project is:

https://hackaday.io/project/181269-the-moncky-project

All code and documentation can be found in gitlab:

https://gitlab.com/big-bat/moncky

Chapter 2

The syntax

The syntax of the Moncky programming language is defined by the following EBNF:

```
monckySourceFile = { declaration } mainDeclaration EOF
mainDeclaration = MAIN { localVarDeclaration } compoundStatement
declaration = constDeclaration
            | recordDeclaration
            | globalVarDeclaration
            | funDeclaration
constDeclaration = CONST IDENTIFIER EQUALS EXPRESSION
recordDeclaration = RECORD IDENTIFIER LEFT_CURLY { recordDeclarationLine }
                    RIGHT_CURLY
globalVarDeclaration = VAR IDENTIFIER COLON TYPE [ EQUALS literal ]
functionDeclaration = FUN IDENTIFIER LEFT_BRACKET { parameterDeclaration }
                      RIGHT_BRACKET [ COLON TYPE ] { localVarDeclaration }
                      compoundStatement
recordDeclarationLine = IDENTIFIER COLON type
parameterDeclaration = IDENTIFIER COLON type
localVarDeclaration = IDENTIFIER COLON type
compoundStatement = LEFT_CURLY { statement } RIGHT_CURLY
statement = assignmentStatement
          | functionCall
          | returnStatement
          | ifStatement
          | whileStatement
assignmentStatement = LET destination EQUALS expression
returnStatement = RETURN [ expression ]
ifStatement = IF expression compoundStatement [ ELSE compoundStatement ]
whileStatement = WHILE expression compoundStatement
expression = literal
           | IDENTIFIER expression2
           | functionCall
           | LEFT_BRACKET ternaryExpression RIGHT_BRACKET
           | LEFT_BRACKET unaryExpression RIGHT_BRACKET
           | LEFT_BRACKET binaryExpression RIGHT_BRACKET
expression2 = POINT IDENTIFIER [ expression2 ]
            | LEFT_SQUARE expression RIGHT_SQUARE [ expression2 ]
ternaryExpression = QUESTION_MARK expression COLON expression COLON
```

```
                    expression
unaryExpression = unaryOperation expression
                | AMPERSAND destination
                | CIRCUMFLEX destination
binaryExpression = expression binaryOperation expression
functionCall = CALL IDENTIFIER LEFT_BRACKET [ parameterList ] RIGHT_BRACKET
parameterList = expression [ parameterList ]
literal = INTEGER_LITERAL
        | FLOAT_LITERAL
        | CHAR_LITERAL
        | STRING_LITERAL
        | TRUE
        | FALSE
        | arrayLiteral
arrayLiteral = LEFT_CURLY primitiveType arrayValues RIGHT_CURLY
arrayValues = INTEGER_LITERAL [ COMMA arrayValues ]
            | FLOAT_LITERAL [ COMMA arrayValues ]
            | CHAR_LITERAL [ COMMA arrayValues ]
            | TRUE [ COMMA arrayValues ]
            | FALSE  [ COMMA arrayValues ]
            | IDENTIFIER [ COMMA arrayValues ]
binaryOperation = PLUS | MINUS | ASTERISK | SLASH | PERCENT | PIPE | AMPERSAND
                | CIRCUMFLEX | SHL | SHR | ASHR | BOOL_EQUALS | BOOL_NOT_EQUALS
                | SMALLER_THAN | GREATER_THAN | SMALLER_THAN_OR_EQUAL
                | GREATER_THAN_OR_EQUAL
unaryOperation = EXCLAMATION | TILDE
type = primitiveType | pointerType | recordType | arrayType
pointerType = POINTER TO type
recordType = IDENTIFIER
arrayType = ARRAY [ LEFT_SQUARE INTEGER_LITERAL RIGHT_SQUARE ] OF type
primitiveType = INT8 | INT16 | INT32 | UINT8 | UINT16 | UINT32
```

Chapter 3

Example programs

This chapter contains some simple programs to demonstrate the syntax of the Moncky programming language. The reader is assumed to be already acquainted with another programming language.

3.1 Hello world

The first program is, of course, the hello world program:

```
main {
  call graphics_setBgCol(0) // set the background colour to black
  call graphics_clrscr() // clear the screen
  call graphics_setCol(0xFF) // set the foreground colour to white
  call text_setCursor(0 0) // set the cursor in the top left
  call text_printString("Hello, World!") // print the string
}
```

Remark that a function call always begins with the keyword 'call'. In this way the compiler knows what is coming. A future version of the language might get rid of this, but it is required now.

Parameters do not have comma's between them. This is not needed because the compiler can figure out itself where parameters begin and end.

There is no semicolon after a statement. It is never needed, even if multiple statements are put on one line.

You can use hexadecimal values, beginning with '0x'.

All firmware functions are available which makes this program very small.

There must always be at least a main function, without any parameters.

3.2 Variables

In the Moncky programming language variables can be declared both globaly and localy. Global variables are defines outside the scope of any function. They must be defined before they are used. Local variables are declared in a function, just before the left curly brace.

Here you see an example:

```
var a: uint16 = 42
main
b : int16
{
  let b = a
  call text_printDecimal(b)
}
```

The variable a is a global variable and the variable b is a local one. Variable b can only be used within the function it is defined in.

Global variables can have initial values as can be seen in the example above.

3.2.1 Primitive types

Each variable has a type. In the Moncky programming language there are 4 primitive types: int16, uint16, int32, and uint32. They can be used to store signed and unsigned integers of 16 and 32 bits. Note that there is no floating point type. However, one is allowed to write the following:

```
var f: uint32 = 0.6072529350088814
```

The given value will be translated to a float32 by the compiler and then interpreted as an integer. This means that the above code is equivalent to:

```
var f: uint32 = 0x3F1B74EE
```

The magic number 0x3F1B74EE corresponds to the bits formed by the IEEE 754 single precision format.

There are also no boolean values. The value 0 is interpreted als 'false' and any other value is considered to be 'true'. However, if the compiler needs to represent 'true', it will always use the value -1 which corresponds to all-ones. The following two statements are therefore equivalent:

```
var b: uint16 = true
var b: uint16 = 0xFFFF
```

Characters can be stored in a uint16 or int16 variable. It is possible to use character literals as follows:

```
var c: uint16 = 'A'
```

which is exactly the same as:

```
var c: uint16 = 65
```

3.2.2 Arrays

The Moncky programming language supports the use of arrays. An array can be declared as follows:

```
var a: array[42] of uint16
```

In this case, variable a will be an array of 42 words of 16 bits. The space is already allocated in memory and the values in it are not initialized.

If one wants to initialize the values, an array literal can be used. Here you see an example:

```
var a: array[3] of uint16 = { uint16 1 2 3 }
```

In this case the array is initialized with the given values. Note that one needs to specify the type of the elements and this does not need to correspond with the type of the variable a. To show how this can be interesting, the following example is given:

```
var a: array[4] of uint16 = { uint32 500000 400000 }
```

In this example the literal is given as two 32-bit integers but they will be accessible as 4 16-bit words. The Moncky compiler will alway put the least significant bits first.

If one initializes the array using a literal, it is not necessary to specify the length. The following is valid:

```
var a: array of uint16 = { int32 -1 -2 }
```

Sometimes an array consists of bytes instead of 16-bit words. In this case it could be interesting to store the array more compact to that each 16-bit word contains two bytes. This is possible as follows:

```
var a: array of uint16 = { uint8 0xC1, 0xE2, 0xA3, 0x64 }
```

The array will now consist of the numbers 0xE2C1 and 0x64A3.

This is also the case when using a string literal. Here you see an example:

```
var s: array of uint16 = "Hello, world!"
```

This will store the string as a null-terminated string of uint8 into the array.

Arrays are indexed from 0 and can be accessed as follows:

```
let i = a[0]
let j = a[i]
```

Arrays can have multiple dimensions. They are defined as follows:

```
var m: array[4] of array[6] of uint16
```

These are accessed as follows:

```
let i = m[3][2]
```

Note that the Moncky programming language does not check the boundaries of an array. The compiler will check only if the index is given as a constant. It is up to the programmer to make sure the array boundaries are not crossed.

3.2.3 Records

Next to arrays, it is also possible to specify a record which bundles data together. Records must first be defined as a type before they can be used. Here is an example:

```
record Point {
    x : int16
    y : int16
}
var point: Point
var points: array[4] of Point
```

The record can be used with a dot as follows:

```
let point.x = 5
let point.y = 4
let points[0].x = 7
let points[1].y = 8
```

Note that it is not allowed to write the following:

```
let point = points[2]
```

This is because the data cannot fit into the registers (in this case it could, but more generaly that is not the case). A memory-copy should be used to transfer big amounts of data.

3.2.4 Pointers

Pointers can be very useful to transfer data to a function without having to copy all data. In the Moncky programming language a pointer is declared as follows:

```
var p: pointer to Point
```

One can now write the following code:

```
let p = (& points[2])
let (^p).x = 42
```

The first statement copies the address of points[2] to the pointer and the second statement assigns the value 42 to the x-value of the Point. The code is equivalent to:

```
let points[2].x = 42
```

Note that parantheses are needed here. More about these parantheses is explained in the next subsection.

3.3 Calculations and operations

In the Moncky programming language it is possible to do perform operations on variables and constants.

There are three types of operations:

- unary operations: these operations operate on a single argument
- binary operations: these operations operate on two arguments
- ternary operation: this operation needs three arguments

When using any of these operations, one must always put parantheses around it so the compiler knows exactly what is coming.

3.3.1 Unary operations

There are 4 unary operations: !, ~, ^, and &.

The ! operator will calculate the logical not of a boolean. This is: if the value was 0, the result will be 'true' and if the value was not 0, the result will be 'false'.

The ~ operator will calculate the inverse of a value. Every bit of the value will be inverted. Note that if one only uses 0 and -1 for booleans, this operation will be equivalent to the ! operator on booleans. It is slightly faster than the ! operator

The ^operator was already shown before. It will result in the value to which a pointer is pointing to.

The & operator will give the address of a variable as a result.

Here is a small example of the usage of a unary operation:

```
let inverse = (!value)
```

3.3.2 Binary operations

There are many binary operations:

- arithmetic operations: +, -, *, /, % (this is the modulo operator)
- bitwise operations: |, &, ^ (or, and, and xor)
- shift operations: <<, >>, >>> (shift left, shift right, and arithmetic shift right)
- logical operations (resulting in a boolean value): ==, !=, <, >, <=, and >=

Here is an example:

```
let c = (a + b)
```

The operations can be nested, always using parentheses:

```
let c = (((a + b) / (c - d)) & e)
```

3.3.3 Ternary operation

There is one ternary operation: ?

This is used to create a short version of an if-statement. It is followed by three arguments.

An example is given here:

```
let a = (? (b < 7) 42 (b - 2))
```

In this case the value of a will depend on the value of b. If b is smaller than 7, a will get the value 42. Otherwise a will be set to b-2.

The generated code will not contain a normal 'if' statement but rather a calculation. It will be equivalent to:

```
let a = ((b < 7) & 42) | ((b >= 7) & (b - 2)))
```

This works because the boolean (b $<$ 7) is stored as 0 (all zeroes) or -1 (all ones).

3.4 Constants

A programmer can easily define constants in the program. This is done outside any function. Constants must be declared before being used.

A simple example will make this clear:

```
const PI = 3.14
```

Notice that there is no need to specify the type. It will be automatically set (uint32 in this case).

3.5 Conditions

In a normal program flow there is a need to take decisions. This is done with an if-statement. The syntax is as follows:

```
if (a > 5) {
  let b = 42
}
```

The curly braces are needed. The compiler will not accept a single statement.

The code will be executed when the result of the expression after 'if' is not zero (false).

One can also add an 'else' statement:

```
if (a > 5) {
  let b = 42
} else {
  let b = 7
}
```

The parentheses after the 'if' are only necessary because a binary operation is used. The following is also correct:

```
let bool = (a > 5)
if bool {
  let b = 42
} else {
  let b = 7
}
```

3.6 Loops

In order to implement a loop, only one statement is implemented at this moment: the while loop.

Here is an example:

```
while (a >= 4) {
  let b = (b >> 1)
  let a = (a - 1)
}
```

The loop is executed as long as the expression next to while is not 0 ('false').

3.7 Functions

In order to reuse software, it is important to be able to create functions. Here is an example of a function: it will calculate the n-th fibonacci number:

```
fun fib(n: uint32):uint32 {
  if (n <= 1) {
    return 1
  } else {
    return (n * call fib((n-1)))
  }
}
main
i: uint16
{
  call graphics_setBgCol(0)
  call graphics_clrscr()
  call graphics_setCol(0xFF)
  call text_setCursor(0 0)
  call text_printString("fib(5) = ")
  let i = fib(5)
  call text_printDecimal(i)
}
```

Recursive functions are supported as a call-stack is used for passing parameters and for local variables.

The type of a variable is always put after the identifier. Local variables are declared between the function declaration and the compound statement.

Chapter 4

The compiler

This chapter describes the compiler that was written for the Moncky programming language. The compiler is programmed in Java. It consists of different parts:

- LexicalAnalyzer: this is used to transform the input file into tokens.
- Compiler: this is the parser that gets its input from the Lexical analyzer and will check the syntax of the file
- Resource managers: these manage the symbol tables and the usage of the registers
- CodeGenerator: this will generate Moncky assembly code
- Types: these classes contain the information of each primitive and user-defined type

The compiler links all generated code with the firmware.

The compiler will generate a file called 'MONCKYOS'. This can be written on the first partition of a SD card. The Moncky computer can read this file and execute the program within it.

The following subsections describe each part of the compiler in more detail.

4.1 LexicalAnalyzer

The lexical analyzer takes a file and transforms it into a stream of tokens. The file is read sequentially. The lexical analyzer keeps track of the current line number.

One token can be put back if needed. This allows the compiler to peek for the next token.

The lexical analyzer will automatically skip comments in C++ style.

The following tokens are defined:

UNKNOWN, EOF, SEMICOLON, IDENTIFIER, COMMA, COLON, FLOAT_LITERAL, INTEGER_LITERAL, STRING_LITERAL, CHAR_LITERAL, FALSE, TRUE, EQUALS, BOOL_EQUALS, BOOL_NOT_EQUALS, BOOL_OR, BOOL_AND, SMALLER_THAN, GREATER_THAN, SMALLER_THAN_OR_EQUAL, GREATER_THAN_OR_EQUAL, EXCLAMATION, POINT, TILDE, QUESTION_MARK, PIPE, AMPERSAND, CIRCUMFLEX, PLUS, MINUS, ASTERISK, SLASH, PERCENT, SHL, SHR, ASHR, LEFT_BRACKET, RIGHT_BRACKET, LEFT_CURLY, RIGHT_CURLY, LEFT_SQUARE, RIGHT_SQUARE, UINT8, UINT16, UINT32, INT8, INT16, INT32, WHILE, IF, ELSE, RETURN, POINTER, TO, ARRAY, OF, CALL, LET, FUN, VAR, CONST, MAIN, RECORD

If a FLOAT_LITERAL, INTEGER_LITERAL, STRING_LITERAL, or CHAR_LITERAL was read, the value is stored in the LexicalAnalyzer so the compiler can read it directly from there.

The main two methods in this class are: nextToken(), putBack(), and lineNumber().

4.2 Compiler

This class contains the parser. It reads the tokens from the LexicalAnalyzer. The grammar is parsed using a technique described by Niklaus Wirth. Every rule of the EBNF is represented by a method (starting with the word 'parse'). This is a very elegant way to parse.

The compiler can check if a token is a valid start for a rule using the FirstChecker class. This utility class contains a map that maps the name of a rule to all valid starting tokens for that rule.

The Compiler class does not create code. It parses the file and checks the grammar. It uses the resource managers to register all encountered entities and the code generator to generate the assembly code.

The compiler will issue an error if an unexpected token is encountered.

4.3 Resource managers

There are different resource managers available:

- The VariableManager contains all global and local variables currently defined. A Variable has a name and a type. A variable can have an initial value (currently only used with global variables) and an address. Global variables have a string as their address. This is the label given to them. Local variables have an integer as address which indicates the offset from the base pointer.
- The LabelManager can generate unique label names
- The ConstantManager has a map of all constants defined so far
- The FunctionManager contains all defined functions. A function has a name, a list of parameters, a return type, a list of local variables, and a flag indicating if it is a firmware function
- The LiteralManager contains all literal strings and arrays encountered so far. This allows to reuse literals and to put them together at the end of the code
- The RecordTypeManager contains all records that are defined so far
- The RegisterManager is used to keep track which registers are in use and which are still free. Registers r0, r1, r2, r12, r13, r14, and r15 are reserved for special purposes.

All managers are accessible through one class: ResourceManager.

4.4 CodeGenerator

The code generator is responsible for generating the resulting assembly code. It is split into two classes: CodeGenerator and CalculationGenerator. It is a rather ugly construction (both classes refer to each other) but at the time this seemed convenient.

The code generator works with objects of type 'Destination' and 'Value'. There is also a class called 'DestinationValue' which can either a Destination or a Value.

A Destination is a pointer to a variable. It represents an address in memory to which a value can be written. The address of the destination can be stored into a register but it can also be stored in the Destination object. The compiler will try to keep the address as long as possible in the object. Only when really needed, the destination will be saved into a register. This allows for much more compact generated code.

A Value is the value of a variable, a constant, or a literal. The value can also be stored in a register or in the object. By keeping the value as long as possible in the object, output code can be made more dense. For instance, if a calculation is done with two constants, the compiler is able to do the calculation. In this case no code is needed to perform the calculation.

4.5 Types

All Moncky data types have a corresponding class in Java. The class hierarchy can be seen here: