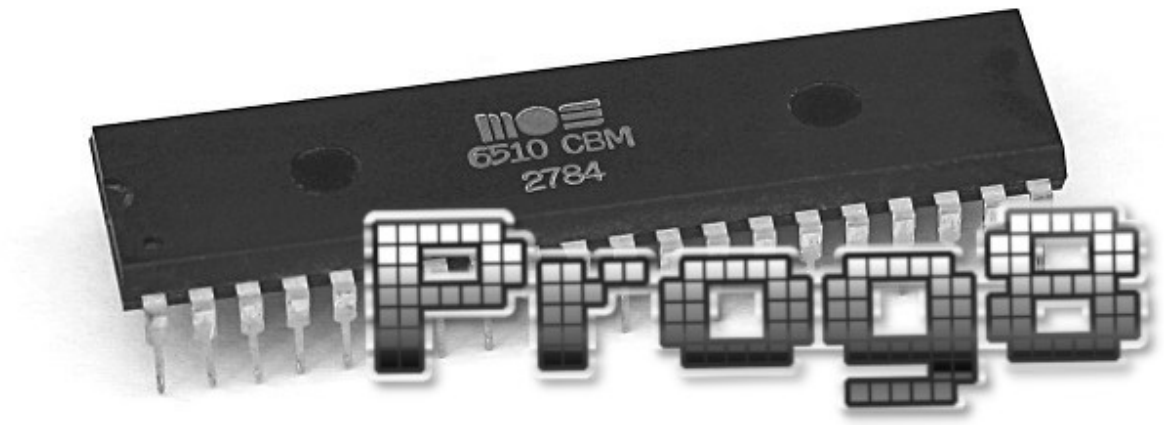
Prog8 Documentation

Release 8.1

Irmen de Jong

May 25, 2022

CONTENTS OF THIS MANUAL:


6510 CBM
2784
prog8

CHAPTER

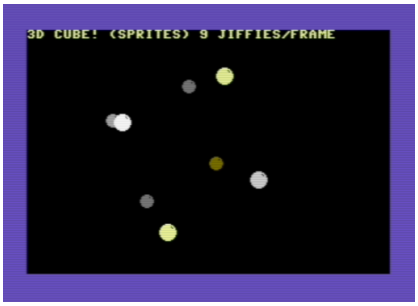
ONE

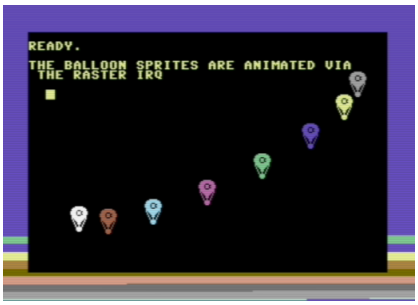
WHAT IS PROG8?

This is a compiled programming language targeting the 8-bit 6502 / 6510 / 65c02 microprocessors. This CPU is from the late 1970's and early 1980's and was used in many home computers from that era, such as the Commodore-64. The language aims to provide many conveniences over raw assembly code (even when using a macro assembler), while still being low level enough to create high performance programs. You can compile programs for various machines with this CPU:

- Commodore 64
- Commander X16
- Commodore 128 (limited support for now)
- Atari 800 XL (limited support for now)

 The project is on github: https://github.com/irmen/prog8.git






IRMEN'S
TEH_TRIZ
HOLD:
NEXT:
LINES:
3
SCORE:
65
CONTROLS:
MOVE
ROTATE
DESCEND
DROP
HOLD
SPEED: 1

CHAPTER

TWO

LANGUAGE FEATURES

- It is a cross-compiler running on modern machines (Linux, MacOS, Windows, ...) It generates a machine code program runnable on actual 8-bit 6502 hardware.
- Fast execution speed due to compilation to native assembly code. It's possible to write certain raster interrupt 'demoscene' effects purely in Prog8.
- Provides a very convenient edit/compile/run cycle by being able to directly launch the compiled program in an emulator and provide debugging information to this emulator.
- Based on simple and familiar imperative structured programming (it looks like a mix of C and Python)
- Modular programming and scoping via modules, code blocks, and subroutines.
- Provide high level programming constructs but at the same time stay close to the metal; still able to directly use memory addresses and ROM subroutines, and inline assembly to have full control when every register, cycle or byte matters
- Subroutines with parameters and return values
- Complex nested expressions are possible
- Variables are allocated statically
- Conditional branches to map directly on processor branch instructions
- `when` statement to avoid if-else chains
- `in` expression for concise and efficient multi-value/containment test
- pipe operator |> to rewrite nested function call expressions in a more readable chained form
- Nested subroutines can access variables from outer scopes to avoids the overhead to pass everything via parameters
- Variable data types include signed and unsigned bytes and words, arrays, strings.
- Floating point math also supported if the target system provides floating point library routines (C64 and Cx16 both do).
- Strings can contain escaped characters but also many symbols directly if they have a petscii equivalent, such as "". Characters like ^, _, \, {, } and | are also accepted and converted to the closest petscii equivalents.
- High-level code optimizations, such as const-folding, expression and statement simplifications/rewriting.
- Many built-in functions, such as `sin`, `cos`, `rnd`, `abs`, `sqrt`, `msb`, `rol`, `ror`, `swap`, `sort` and `reverse`
- Programs can be run multiple times without reloading because of automatic variable (re)initializations.
- Supports the sixteen 'virtual' 16-bit registers R0 .. R15 from the Commander X16, also on the other machines.

- If you only use standard kernal and core prog8 library routines, it is possible to compile the *exact same program* for different machines (just change the compilation target flag)!

CHAPTER

THREE

CODE EXAMPLE

Here is a hello world program:

```
%import textio

main {
    sub start() {
        txt.print("hello world i  prog8\n")
    }
}
```

This code calculates prime numbers using the Sieve of Eratosthenes algorithm:

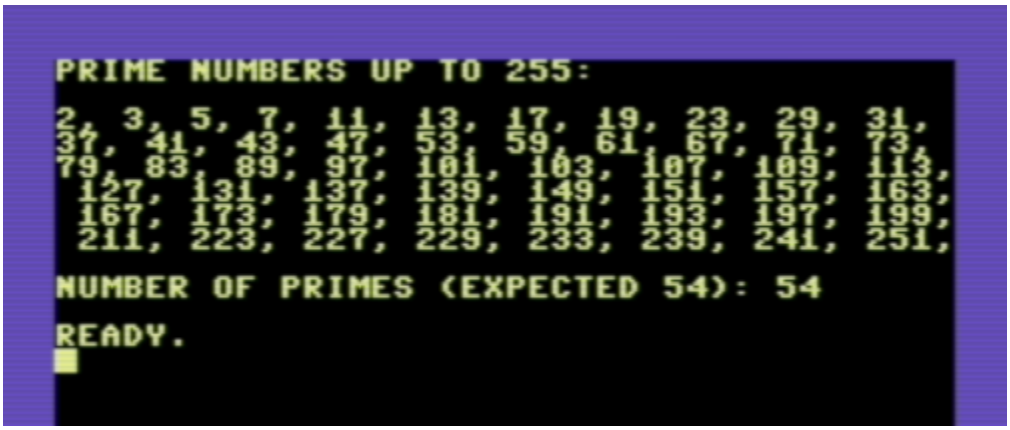
```
%import textio
%zeropage basicsafe

main {
    ubyte[256] sieve
    ubyte candidate_prime = 2       ; is increased in the loop

    sub start() {
        ; clear the sieve, to reset starting situation on subsequent runs
        sys.memset(sieve, 256, false)
        ; calculate primes
        txt.print("prime numbers up to 255:\n\n")
        ubyte amount=0
        repeat {
            ubyte prime = find_next_prime()
            if prime==0
                break
            txt.print_ub(prime)
            txt.print(", ")
            amount++
        }
        txt.nl()
        txt.print("number of primes (expected 54): ")
        txt.print_ub(amount)
        txt.nl()
    }

    sub find_next_prime() -> ubyte {
        while sieve[candidate_prime] {
```

(continues on next page)

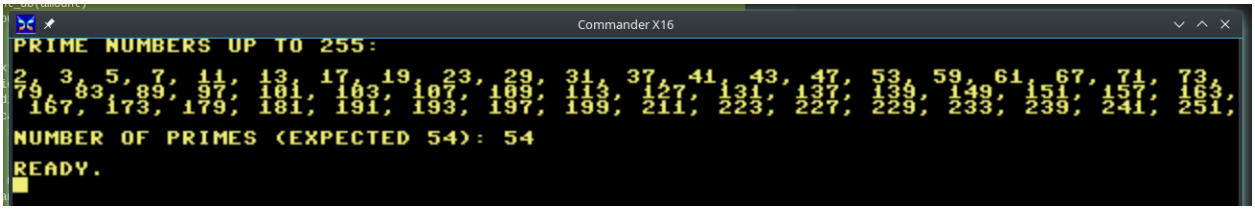
(continued from previous page)

```
            candidate_prime++
            if candidate_prime==0
                return 0        ; we wrapped; no more primes available in the sieve
        }

        ; found next one, mark the multiples and return it.
        sieve[candidate_prime] = true
        uword multiple = candidate_prime

        while multiple < len(sieve) {
            sieve[lsb(multiple)] = true
            multiple += candidate_prime
        }
        return candidate_prime
    }
}
```

when compiled an ran on a C-64 you get this:



when the exact same program is compiled for the Commander X16 target, and run on the emulator, you get this:



CHAPTER

FOUR

GETTING THE COMPILER

Usually you just download a fat jar of an official released version, but you can also build it yourself from source. Detailed instructions on how to obtain a version of the compiler are in *First, getting a working compiler*.

CHAPTER

FIVE

REQUIRED ADDITIONAL TOOLS

64tass - cross assembler. Install this on your shell path. It's very easy to compile yourself. A recent precompiled .exe (only for Windows) can be obtained from my clone of this project. *You need at least version 1.55.2257 of this assembler to correctly use the breakpoints feature.* It's possible to use older versions, but it is very likely that the automatic Vice breakpoints won't work with them.

A **Java runtime (jre or jdk), version 11 or newer** is required to run the prog8 compiler itself. If you're scared of Oracle's licensing terms, most Linux distributions ship OpenJDK in their packages repository instead. For Windows it's possible to get that as well; check out AdoptOpenJDK . For MacOS you can use the Homebrew system to install a recent version of OpenJDK.

Finally: an **emulator** (or a real machine ofcourse) to test and run your programs on. In C64 mode, the compiler assumes the presence of the Vice emulator. If you're targeting the CommanderX16 instead, there's a choice of the official x16emu and the unofficial box16 (you can select which one you want to launch using the `-emu` or `-emu2` command line options)

5.1 Writing and building a program

5.1.1 First, getting a working compiler

Before you can compile Prog8 programs, you'll have to download or build the compiler itself. First make sure you have installed the *Required additional tools*. Then you can choose a few ways to get a compiler:

Download a precompiled version from github:

1. download a recent "fat-jar" (called something like "prog8compiler-all.jar") from the releases on Github
2. run the compiler with "java -jar prog8compiler-all.jar" to see how you can use it.

Using the Gradle build system to build it yourself:

The Gradle build system is used to build the compiler. The most interesting gradle commands to run are probably:

`./gradlew build` Builds the compiler code and runs all available checks and unit-tests. Also automatically runs the installDist and installShadowDist tasks.

`./gradlew installDist` Builds the compiler and installs it with scripts to run it, in the directory `./compiler/build/install/p8compile`

`./gradlew installShadowDist` Creates a 'fat-jar' that contains the compiler and all dependencies, in a single executable .jar file, and includes few start scripts to run it. The output can be found in `.compiler/build/install/compiler-shadow/`

`./gradlew shadowDistZip` Creates a zipfile with the above in it, for easy distribution. This file can be found in `./compiler/build/distributions/`

For normal use, the `installDist` target should suffice and after succesful completion, you can start the compiler with:

```
./compiler/build/install/p8compile/bin/p8compile <options> <sourcefile>
```

(You should probably make an alias...)

Hint: Development and testing is done on Linux using the IntelliJ IDEA IDE, but the compiler should run on most operating systems that provide a fairly modern java runtime (11 or newer). If you do have trouble building or running the compiler on your operating system, please let me know!

5.1.2 What is a Prog8 "Program" anyway?

A "complete runnable program" is a compiled, assembled, and linked together single unit. It contains all of the program's code and data and has a certain file format that allows it to be loaded directly on the target system. Prog8 currently has no built-in support for programs that exceed 64 Kb of memory, nor for multi-part loaders.

For the Commodore-64, most programs will have a tiny BASIC launcher that does a SYS into the generated machine code. This way the user can load it as any other program and simply RUN it to start. (This is a regular ".prg" program). Prog8 can create those, but it is also possible to output plain binary programs that can be loaded into memory anywhere.

5.1.3 Running the compiler

Make sure you have installed the *Required additional tools*.

You run the Prog8 compiler on a main source code module file. Other modules that this code needs will be loaded and processed via imports from within that file. The compiler will link everything together into one output program at the end.

If you start the compiler without arguments, it will print a short usage text. For normal use the compiler can be invoked with the command:

```
$ java -jar prog8compiler-7.3-all.jar sourcefile.p8
```

(Use the appropriate name and version of the jar file downloaded from one of the Git releases. Other ways to invoke the compiler are also available: see the introduction page about how to build and run the compiler yourself)

By default, assembly code is generated and written to `sourcefile.asm`. It is then (automatically) fed to the 64tass assembler tool that creastes the final runnable program.

Command line options

One or more .p8 module files Specify the main module file(s) to compile. Every file specified is a separate program.

`-help`, `-h` Prints short command line usage information.

`-target <compilation target>` Sets the target output of the compiler, currently 'c64' and 'cx16' are valid targets. c64 = Commodore 64, c128 = Commodore 128, cx16 = Commander X16, atari = Atari 800 XL Default = c64

`-srcdirs <pathlist>` Specify a list of extra paths (separated with ':'), to search in for imported modules. Useful if you have library modules somewhere that you want to re-use, or to switch implementations of certain routines via a command line switch.

`-emu`, `-emu2` Auto-starts target system emulator after successful compilation. emu2 starts the alternative emulator if available. The compiled program and the symbol and breakpoint lists (for the machine code monitor) are immediately loaded into the emulator..

-out <directory> sets directory location for output files instead of current directory

-noasm Do not create assembly code and output program. Useful for debugging or doing quick syntax checks.

-noopt Don't perform any code optimizations. Useful for debugging or faster compilation cycles.

-noreinit Don't create code to reinitialize the global (block level) variables on every run of the program. Also means that all such variables are no longer placed in the zero page. Sometimes the program will be a lot shorter when using this, but sometimes the opposite happens. When using this option, it is no longer be possible to run the program correctly more than once! *Experimental feature*: still has some problems!

-optfloatx Also optimize float expressions if optimizations are enabled. Warning: can increase program size significantly if a lot of floating point expressions are used.

-watch Enables continuous compilation mode (watches for file changes). This greatly increases compilation speed on subsequent runs: almost instant compilation times (less than a second) can be achieved in this mode. The compiler will compile your program and then instead of exiting, it waits for any changes in the module source files. As soon as a change happens, the program gets compiled again. Note that it is possible to use the watch mode with multiple modules as well, but it will recompile everything in that list even if only one of the files got updated.

-slowwarn Shows debug warnings about slow or problematic assembly code generation. Ideally, the compiler should use as few stack based evaluations as possible.

-quietasm Don't print assembler output results.

-asmlist Generate an assembler listing file as well.

-expericodegen Use experimental code generation backend (*incomplete*).

5.1.4 Module source code files

A module source file is a text file with the `.p8` suffix, containing the program's source code. It consists of compilation options and other directives, imports of other modules, and source code for one or more code blocks.

Prog8 has various *LIBRARY* modules that are defined in special internal files provided by the compiler. You should not overwrite these or reuse their names. They are embedded into the packaged release version of the compiler so you don't have to worry about where they are, but their names are still reserved.

Importing other source files and specifying search location(s)

You can create multiple source files yourself to modularize your large programs into multiple module files. You can also create "library" modules this way with handy routines, that can be shared among programs. By importing those module files, you can use them in other modules. It is possible to tell the compiler where it should look for these files, by using the `srcdirs` command line option.

5.1.5 Debugging (with Vice)

There's support for using the monitor and debugging capabilities of the rather excellent Vice emulator.

The `%breakpoint` directive (see *Directives*) in the source code instructs the compiler to put a *breakpoint* at that position. Some systems use a BRK instruction for this, but this will usually halt the machine altogether instead of just suspending execution. Prog8 issues a NOP instruction instead and creates a 'virtual' breakpoint at this position. All breakpoints are then written to a file called "programname.vice-mon-list", which is meant to be used by the Vice emulator. It contains a series of commands for Vice's monitor, including source labels and the breakpoint settings. If you use the emulator autostart feature of the compiler, it will take care of this for you. If you launch Vice manually, you'll have to use a command line option to load this file:

```
$ x64 -moncommands programname.vice-mon-list
```

Vice will then use the label names in memory disassembly, and will activate any breakpoints as well. If your running program hits one of the breakpoints, Vice will halt execution and drop you into the monitor.

5.1.6 Troubleshooting

Getting an assembler error about undefined symbols such as `not defined 'floats'`? This happens when your program uses floating point values, and you forgot to import `floats` library. If you use floating points, the compiler needs routines from that library. Fix it by adding an `%import floats`.

5.1.7 Examples

A couple of example programs can be found in the 'examples' directory of the source tree. Make sure you have installed the *Required additional tools*. Then, for instance, to compile and run the rasterbars example program, use this command:

```
$ java -jar prog8compiler.jar -emu examples/rasterbars.p8
```

or:

```
$ ./p8compile.sh -emu examples/rasterbars.p8
```

5.2 Programming in Prog8

This chapter describes a high level overview of the elements that make up a program. Details about the syntax can be found in the *Syntax Reference* chapter.

5.2.1 Elements of a program

Program Consists of one or more *modules*.

Module A file on disk with the `.p8` suffix. It can contain *directives* and *code blocks*. Whitespace and indentation in the source code are arbitrary and can be mixed tabs or spaces. A module file can *import* other modules, including *library modules*.

Comments Everything after a semicolon `;` is a comment and is ignored by the compiler. If the whole line is just a comment, this line will be copied into the resulting assembly source code for reference.

Directive These are special instructions for the compiler, to change how it processes the code and what kind of program it creates. A directive is on its own line in the file, and starts with `%`, optionally followed by some arguments.

Code block A block of actual program code. It has a starting address in memory, and defines a *scope* (also known as 'namespace'). It contains variables and subroutines. More details about this below: *Blocks, Scopes, and accessing Symbols*.

Variable declarations The data that the code works on is stored in variables ('named values that can change'). The compiler allocates the required memory for them. There is *no dynamic memory allocation*. The storage size of all variables is fixed and is determined at compile time. Variable declarations tend to appear at the top of the code block that uses them, but this is not mandatory. They define the name and type of the variable, and its initial value. Prog8 supports a small list of data types, including special 'memory mapped' types that don't allocate storage but instead point to a fixed location in the address space.

Code These are the instructions that make up the program's logic. Code can only occur inside a subroutine. There are different kinds of instructions ('statements' is a better name) such as:

- value assignment
- looping (for, while, do-until, repeat, unconditional jumps)
- conditional execution (if - then - else, when, and conditional jumps)
- subroutine calls
- label definition

Subroutine Defines a piece of code that can be called by its name from different locations in your code. It accepts parameters and can return a value (optional). It can define its own variables, and it is even possible to define subroutines nested inside other subroutines. Their contents is scoped accordingly. Nested subroutines can access the variables from outer scopes. This removes the need and overhead to pass everything via parameters. Subroutines do not have to be declared before they can be called.

Label This is a named position in your code where you can jump to from another place. You can jump to it with a jump statement elsewhere. It is also possible to use a subroutine call to a label (but without parameters and return value).

Scope Also known as 'namespace', this is a named box around the symbols defined in it. This prevents name collisions (or 'namespace pollution'), because the name of the scope is needed as prefix to be able to access the symbols in it. Anything *inside* the scope can refer to symbols in the same scope without using a prefix. There are three scope levels in Prog8:

- global (no prefix)
- code block
- subroutine

While Modules are separate files, they are *not* separate scopes! Everything defined in a module is merged into the global scope. This is different from most other languages that have modules. The global scope can only contain blocks and some directives, while the others can contain variables and subroutines too.

5.2.2 Blocks, Scopes, and accessing Symbols

Blocks are the top level separate pieces of code and data of your program. They have a starting address in memory and will be combined together into a single output program. They can only contain *directives*, *variable declarations*, *subroutines* and *inline assembly code*. Your actual program code can only exist inside these subroutines. (except the occasional inline assembly)

Here's an example:

```
main $c000 {
    ; this is code inside the block...
}
```

The name of a block must be unique in your entire program. Be careful when importing other modules; blocks in your own code cannot have the same name as a block defined in an imported module or library.

If you omit both the name and address, the entire block is *ignored* by the compiler (and a warning is displayed). This is a way to quickly "comment out" a piece of code that is unfinshed or may contain errors that you want to work on later, because the contents of the ignored block are not fully parsed either.

The address can be used to place a block at a specific location in memory. Usually it is omitted, and the compiler will automatically choose the location (usually immediately after the previous block in memory). It must be `>= $0200` (because `$00`–`$ff` is the ZP and `$100`–`$1ff` is the cpu stack).

Scoping rules

Use qualified names ("dotted names") to reference symbols defined elsewhere

In prog8 every symbol is 'public' and can be accessed from anywhere else, given its *full* "dotted name". So, accessing a variable `counter` defined in subroutine `worker` in block `main`, can be done from anywhere by using `main.worker.counter`.

Symbols are names defined in a certain *scope*. Inside the same scope, you can refer to them by their 'short' name directly. If the symbol is not found in the same scope, the enclosing scope is searched for it, and so on, up to the top level block, until the symbol is found. If the symbol was not found the compiler will issue an error message.

Scopes are created using either of these two statements:

- blocks (top-level named scope)
- subroutines (nested named scope)

Important: Unlike most other programming languages, a new scope is *not* created inside for, while, repeat, and do-until statements, the if statement, and the branching conditionals. These all share the same scope from the subroutine they're defined in. You can define variables in these blocks, but these will be treated as if they were defined in the subroutine instead. This can seem a bit restrictive because you have to think harder about what variables you want to use inside the subroutine, to avoid clashes. But this decision was made for a good reason: memory in prog8's target systems is usually very limited and it would be a waste to allocate a lot of variables. The prog8 compiler is not yet advanced enough to be able to share or overlap variables intelligently. So for now that is something you have to think about yourself.

5.2.3 Program Start and Entry Point

Your program must have a single entry point where code execution begins. The compiler expects a `start` subroutine in the `main` block for this, taking no parameters and having no return value.

As any subroutine, it has to end with a `return` statement (or a `goto` call):

```
main {
    sub start ()  {
        ; program entrypoint code here
        return
    }
}
```

The `main` module is always relocated to the start of your programs address space, and the `start` subroutine (the entrypoint) will be on the first address. This will also be the address that the BASIC loader program (if generated) calls with the SYS statement.

5.2.4 Variables and values

Variables are named values that can change during the execution of the program. They can be defined inside any scope (blocks, subroutines etc.) See *Scopes*. When declaring a numeric variable it is possible to specify the initial value, if you don't want it to be zero. For other data types it is required to specify that initial value it should get. Values will usually be part of an expression or assignment statement:

```
12345                ; integer number
$aa43                ; hex integer number
%100101              ; binary integer number (% is also remainder operator so be␣
↪careful)
-33.456e52           ; floating point number
"Hi, I am a string"  ; text string, encoded with default encoding
'a'                  ; byte value (ubyte) for the letter a
sc:"Alternate"       ; text string, encoded with c64 screencode encoding
sc:'a'               ; byte value of the letter a in c64 screencode encoding

byte  counter  = 42  ; variable of size 8 bits, with initial value 42
```

putting a variable in zeropage: If you add the `@zp` tag to the variable declaration, the compiler will prioritize this variable when selecting variables to put into zero page (but no guarantees). If there are enough free locations in the zeropage, it will try to fill it with as much other variables as possible (before they will be put in regular memory pages). Use `@requirezp` tag to *force* the variable into zeropage, but if there is no more free space the compilation will fail. It's possible to put strings, arrays and floats into zeropage too, however because Zp space is really scarce this is not advised as they will eat up the available space very quickly. It's best to only put byte or word variables in Zeropage.

Example:

```
byte   @zp  smallcounter = 42
uword  @requirezp  zppointer = $4000
```

shared tag: If you add the `@shared` tag to the variable declaration, the compiler will know that this variable is a prog8 variable shared with some assembly code elsewhere. This means that the assembly code can refer to the variable even if it's otherwise not used in prog8 code itself. (usually, these kinds of 'unused' variables are optimized away by the compiler, resulting in an error when assembling the rest of the code). Example:

```
byte  @shared  assemblyVariable = 42
```

Integers

Integers are 8 or 16 bit numbers and can be written in normal decimal notation, in hexadecimal and in binary notation. A single character in single quotes such as `'a'` is translated into a byte integer, which is the Petscii value for that character.

Unsigned integers are in the range 0-255 for unsigned byte types, and 0-65535 for unsigned word types. The signed integers integers are in the range -128..127 for bytes, and -32768..32767 for words.

Floating point numbers

Floats are stored in the 5-byte 'MFLPT' format that is used on CBM machines, and currently all floating point operations are specific to the Commodore-64. This is because routines in the C-64 BASIC and KERNAL ROMs are used for that. So floating point operations will only work if the C-64 BASIC ROM (and KERNAL ROM) are banked in.

Also your code needs to import the `floats` library to enable floating point support in the compiler, and to gain access to the floating point routines. (this library contains the directive to enable floating points, you don't have to worry about this yourself)

The largest 5-byte MFLPT float that can be stored is: **1.7014118345e+38** (negative: **-1.7014118345e+38**)

Note: On the Commander X16, to use floating point operations, ROM bank 4 has to be enabled (BASIC). Importing the `floats` library will do this for you if needed.

Arrays

Array types are also supported. They can be formed from a list of bytes, words, floats, or addresses of other variables (such as explicit address-of expressions, strings, or other array variables) - values in an array literal always have to be constants. Putting variables inside an array has to be done on a value-by-value basis. Here are some examples of arrays:

```
byte[10]  array                   ; array of 10 bytes, initially set to 0
byte[]  array = [1, 2, 3, 4]      ; initialize the array, size taken from value
byte[99] array = 255              ; initialize array with 99 times 255 [255, 255, 255,␣
↪255, ...]
byte[] array = 100 to 199         ; initialize array with [100, 101, ..., 198, 199]
str[] names = ["ally", "pete"]    ; array of string pointers/addresses (equivalent to␣
↪uword)
uword[] others = [names, array]   ; array of pointers/addresses to other arrays

value = array[3]            ; the fourth value in the array (index is 0-based)
char = string[4]            ; the fifth character (=byte) in the string
```

Note: Right now, the array should be small enough to be indexable by a single byte index. This means byte arrays should be <= 256 elements, word arrays <= 128 elements, and float arrays <= 51 elements.

You can split an array initializer list over several lines if you want.

Note that the various keywords for the data type and variable type (`byte`, `word`, `const`, etc.) can't be used as *identifiers* elsewhere. You can't make a variable, block or subroutine with the name `byte` for instance.

It's possible to assign a new array to another array, this will overwrite all elements in the original array with those in the value array. The number and types of elements have to match. For large arrays this is a slow operation because every element is copied over. It should probably be avoided.

Using the `in` operator you can easily check if a value is present in an array, example: `if choice in [1,2,3,4] {....}`

Arrays at a specific memory location: Using the memory-mapped syntax it is possible to define an array to be located at a specific memory location. For instance to reference the first 5 rows of the Commodore 64's screen matrix as an array, you can define:

```
&ubyte[5*40]  top5screenrows = $0400
```

This way you can set the second character on the second row from the top like this:

```
top5screenrows[41] = '!'
```

Array indexing on a pointer variable: An uword variable can be used in limited scenarios as a 'pointer' to a byte in memory at a specific, dynamic, location. You can use array indexing on a pointer variable to use it as a byte array at a dynamic location in memory: currently this is equivalent to directly referencing the bytes in memory at the given index. See also *Direct access to memory locations*

Strings

Strings are a sequence of characters enclosed in " quotes. The length is limited to 255 characters. They're stored and treated much the same as a byte array, but they have some special properties because they are considered to be *text*. Strings (without encoding prefix) will be encoded (translated from ASCII/UTF-8) into bytes via the *default encoding* for the target platform. On the CBM machines, this is CBM PETSCII.

Alternative encodings can be specified with a `encodingname:` prefix to the string or character literal. The following encodings are currently recognised:

- `petscii` Petscii, the default encoding on CBM machines (c64, c128, cx16)
- `sc` CBM-screencodes aka 'poke' codes (c64, c128, cx16)
- `iso` iso-8859-15 text (supported on cx16)

So the following is a string literal that will be encoded into memory bytes using the iso encoding. It can be correctly displayed on the screen only if a iso-8859-15 charset has been activated first (the Commander X16 has this feature built in):

```
iso:"Käse, Straße"
```

You can concatenate two string literals using '+', which can be useful to split long strings over separate lines. But remember that the length of the total string still cannot exceed 255 characaters. A string literal can also be repeated a given number of times using '*', where the repeat number must be a constant value. And a new string value can be assigned to another string, but no bounds check is done so be sure the destination string is large enough to contain the new value (it is overwritten in memory):

```
str string1 = "first part" + "second part"
str string2 = "hello!" * 10

string1 = string2
string1 = "new value"
```

There are several 'escape sequences' to help you put special characters into strings, such as newlines, quote characters themselves, and so on. The ones used most often are `\\`, `\"`, `\n`, `\r`. For a detailed description of all of them and what they mean, read the syntax reference on strings.

Using the `in` operator you can easily check if a characater is present in a string, example: `if '@' in email_address {....}` (however this gives no clue about the location in the string where the character is present, if you need that, use the `string.find()` library function instead)

Hint: Strings/arrays and uwords (=memory address) can often be interchanged. An array of strings is actually an array of uwords where every element is the memory address of the string. You can pass a memory address to assembly functions that require a string as an argument. For regular assignments you still need to use an explicit `&` (address-of) to take the address of the string or array.

Note: Strings and their (im)mutability

String literals outside of a string variable's initialization value, are considered to be "constant", i.e. the string isn't going to change during the execution of the program. The compiler takes advantage of this in certain ways. For instance, multiple identical occurrences of a string literal are folded into just one string allocation in memory. Examples of such strings are the string literals passed to a subroutine as arguments.

Strings that aren't such string literals are considered to be unique, even if they are the same as a string defined elsewhere. This includes the strings assigned to a string variable in its declaration! These kind of strings are not deduplicated and are just copied into the program in their own unique part of memory. This means that it is okay to treat those strings as mutable; you can safely change the contents of such a string without destroying other occurrences (as long as you stay within the size of the allocated string!)

Special types: const and memory-mapped

When using `const`, the value of the 'variable' can no longer be changed. You'll have to specify the initial value expression. This value is then used by the compiler everywhere you refer to the constant (and no storage is allocated for the constant itself). This is only valid for the simple numeric types (byte, word, float).

When using `&` (the address-of operator but now applied to a datatype), the variable will point to specific location in memory, rather than being newly allocated. The initial value (mandatory) must be a valid memory address. Reading the variable will read the given data type from the address you specified, and setting the varible will directly modify that memory location(s):

```
const  byte  max_age = 2000 - 1974      ; max_age will be the constant value 26
&word  SCREENCOLORS = $d020             ; a 16-bit word at the addres $d020-$d021
```

Direct access to memory locations

Normally memory locations are accessed by a *memory mapped* name, such as `c64.BGCOL0` that is defined as the memory mapped address $d021.

If you want to access a memory location directly (by using the address itself or via an uword pointer variable), without defining a memory mapped location, you can do so by enclosing the address in `@(...)`:

```
color = @($d020)  ; set the variable 'color' to the current c64 screen border color (
↪"peek(53280)")
@($d020) = 0      ; set the c64 screen border to black ("poke 53280,0")
@(vic+$20) = 6    ; you can also use expressions to 'calculate' the address
```

This is the official syntax to 'dereference a pointer' as it is often named in other languages. You can actually also use the array indexing notation for this. It will be silently converted into the direct memory access expression as explained above. Note that this also means that unlike regular arrays, the index is not limited to an ubyte value. You can use a full uword to index a pointer variable like this:

```
pointervar[999] = 0     ; set memory byte to zero at location pointervar + 999.
```

Converting types into other types

Sometimes you need an unsigned word where you have an unsigned byte, or you need some other type conversion. Many type conversions are possible by just writing `as <type>` at the end of an expression:

```
uword  uw = $ea31
ubyte  ub = uw as ubyte     ; ub will be $31, identical to lsb(uw)
float  f = uw as float      ; f will be 59953, but this conversion can be omitted in␣
↪this case
word   w = uw as word       ; w will be -5583 (simply reinterpret $ea31 as 2-complement␣
↪negative number)
f = 56.777
ub = f as ubyte             ; ub will be 56
```

Sometimes it is a straight 'type cast' where the value is simply interpreted as being of the other type, sometimes an actual value conversion is done to convert it into the targe type. Try to avoid type conversions as much as possible.

Initial values across multiple runs of the program

When declaring values with an initial value, this value will be set into the variable each time the program reaches the declaration again. This can be in loops, multiple subroutine calls, or even multiple invocations of the entire program. If you omit the initial value, zero will be used instead.

This only works for simple types, *and not for string variables and arrays*. It is assumed these are left unchanged by the program; they are not re-initialized on a second run. If you do modify them in-place, you should take care yourself that they work as expected when the program is restarted. (This is an optimization choice to avoid having to store two copies of every string and array)

5.2.5 Loops

The *for*-loop is used to let a variable iterate over a range of values. Iteration is done in steps of 1, but you can change this. The loop variable must be declared separately as byte or word earlier, so that you can reuse it for multiple occasions. Iterating with a floating point variable is not supported. If you want to loop over a floating-point array, use a loop with an integer index variable instead.

The *while*-loop is used to repeat a piece of code while a certain condition is still true. The *do–until* loop is used to repeat a piece of code until a certain condition is true. The *repeat* loop is used as a short notation of a for loop where the loop variable doesn't matter and you're only interested in the number of iterations. (without iteration count specified it simply loops forever).

You can also create loops by using the `goto` statement, but this should usually be avoided.

Breaking out of a loop prematurely is possible with the `break` statement.

Attention: The value of the loop variable after executing the loop *is undefined*. Don't use it immediately after the loop without first assigning a new value to it! (this is an optimization issue to avoid having to deal with mostly useless post-loop logic to adjust the loop variable's value)

Warning: For efficiency reasons, it is assumed that the ending value of the for loop is actually >= the starting value (or <= if the step is negative). This means that for loops in prog8 behave differently than in other languages if this is *not* the case! A for loop from ubyte 10 to ubyte 2, for example, will iterate through all values 10, 11, 12, 13,

.... 254, 255, 0 (wrapped), 1, 2. In other languages the entire loop will be skipped in such cases. But prog8 omits the overhead of an extra loop range check and/or branch for every for loop by assuming the normal ranges.

5.2.6 Conditional Execution

if statements

Conditional execution means that the flow of execution changes based on certiain conditions, rather than having fixed gotos or subroutine calls:

```
if aa>4 goto overflow

if xx==3  yy = 4
if xx==3  yy = 4 else  aa = 2

if xx==5 {
        yy = 99
} else {
        aa = 3
}
```

Conditional jumps (`if condition goto label`) are compiled using 6502's branching instructions (such as `bne` and `bcc`) so the rather strict limit on how *far* it can jump applies. The compiler itself can't figure this out unfortunately, so it is entirely possible to create code that cannot be assembled successfully. Thankfully the `64tass` assembler that is used has the option to automatically convert such branches to their opposite + a normal jmp. This is slower and takes up more space and you will get warning printed if this happens. You may then want to restructure your branches (place target labels closer to the branch, or reduce code complexity).

There is a special form of the if-statement that immediately translates into one of the 6502's branching instructions. This allows you to write a conditional jump or block execution directly acting on the current values of the CPU's status register bits. The eight branching instructions of the CPU each have an if-equivalent (and there are some easier to understand aliases):

condition	meaning
`if_cs`	if carry status is set
`if_cc`	if carry status is clear
`if_vs`	if overflow status is set
`if_vc`	if overflow status is clear
`if_eq` / `if_z`	if result is equal to zero
`if_ne` / `if_nz`	if result is not equal to zero
`if_pl` / `if_pos`	if result is 'plus' (>= zero)
`if_mi` / `if_neg`	if result is 'minus' (< zero)

So `if_cc goto target` will directly translate into the single CPU instruction `BCC target`.

Caution: These special `if_XX` branching statements are only useful in certain specific situations where you are *certain* that the status register (still) contains the correct status bits. This is not always the case after a fuction call or other operations! If in doubt, check the generated assembly code!

Note: For now, the symbols used or declared in the statement block(s) are shared with the same scope the if statement itself is in. Maybe in the future this will be a separate nested scope, but for now, that is only possible when defining a subroutine.

when statement ('jump table')

Instead of writing a bunch of sequential if-elseif statements, it is more readable to use a `when` statement. (It will also result in greatly improved assembly code generation) Use a `when` statement if you have a set of fixed choices that each should result in a certain action. It is possible to combine several choices to result in the same action:

```
when value {
    4 -> txt.print("four")
    5 -> txt.print("five")
    10,20,30 -> {
        txt.print("ten or twenty or thirty")
    }
    else -> txt.print("don't know")
}
```

The when-*value* can be any expression but the choice values have to evaluate to compile-time constant integers (bytes or words). They also have to be the same datatype as the when-value, otherwise no efficient comparison can be done.

Note: Instead of chaining several value equality checks together using `or` (ex.: `if x==1 or xx==5 or xx==9`), consider using a `when` statement or `in` containment check instead. These are more efficient.

5.2.7 Assignments

Assignment statements assign a single value to a target variable or memory location. Augmented assignments (such as `aa += xx`) are also available, but these are just shorthands for normal assignments (`aa = aa + xx`).

Only variables of type byte, word and float can be assigned a new value. It's not possible to set a new value to string or array variables etc, because they get allocated a fixed amount of memory which will not change. (You *can* change the value of elements in a string or array though).

Attention: Data type conversion (in assignments): When assigning a value with a 'smaller' datatype to variable with a 'larger' datatype, the value will be automatically converted to the target datatype: byte –> word –> float. So assigning a byte to a word variable, or a word to a floating point variable, is fine. The reverse is *not* true: it is *not* possible to assign a value of a 'larger' datatype to a variable of a smaller datatype without an explicit conversion. Otherwise you'll get an error telling you that there is a loss of precision. You can use builtin functions such as `round` and `lsb` to convert to a smaller datatype, or revert to integer arithmetic.

5.2.8 Expressions

Expressions tell the program to *calculate* something. They consist of values, variables, operators such as + and -, function calls, type casts, or other expressions. Here is an example that calculates to number of seconds in a certain time period:

```
num_hours * 3600 + num_minutes * 60 + num_seconds
```

Long expressions can be split over multiple lines by inserting a line break before or after an operator:

```
num_hours * 3600
 + num_minutes * 60
 + num_seconds
```

In most places where a number or other value is expected, you can use just the number, or a constant expression. If possible, the expression is parsed and evaluated by the compiler itself at compile time, and the (constant) resulting value is used in its place. Expressions that cannot be compile-time evaluated will result in code that calculates them at runtime. Expressions can contain procedure and function calls. There are various built-in functions such as sin(), cos() that can be used in expressions (see *Built-in Functions*). You can also reference idendifiers defined elsewhere in your code.

Read the *Syntax Reference* chapter for all details on the available operators and kinds of expressions you can write.

Note: Order of evaluation:

The order of evaluation of expression operands is *unspecified* and should not be relied upon. There is no guarantee of a left-to-right or right-to-left evaluation. But don't confuse this with operator precedence order (multiplication comes before addition etcetera).

Attention: Floating point values used in expressions:

When a floating point value is used in a calculation, the result will be a floating point, and byte or word values will be automatically converted into floats in this case. The compiler will issue a warning though when this happens, because floating point calculations are very slow and possibly unintended!

Calculations with integer variables will not result in floating point values. if you divide two integer variables say 32500 and 99 the result will be the integer floor division (328) rather than the floating point result (328.2828282828283). If you need the full precision, you'll have to make sure at least the first operand is a floating point. You can do this by using a floating point value or variable, or use a type cast. When the compiler can calculate the result during compile-time, it will try to avoid loss of precision though and gives an error if you may be losing a floating point result.

Arithmetic and Logical expressions

Arithmetic expressions are expressions that calculate a numeric result (integer or floating point). Many common arithmetic operators can be used and follow the regular precedence rules. Logical expressions are expressions that calculate a boolean result: true or false (which in reality are just a 1 or 0 integer value).

You can use parentheses to group parts of an expresion to change the precedence. Usually the normal precedence rules apply (* goes before + etc.) but subexpressions within parentheses will be evaluated first. So `(4 + 8) * 2` is 24 and not 20, and `(true or false) and false` is false instead of true.

Attention: calculations keep their datatype even if the target variable is larger: When you do calculations on a BYTE type, the result will remain a BYTE. When you do calculations on a WORD type, the result will remain a WORD. For instance:

```
byte b = 44
word w = b*55   ; the result will be 116! (even though the target variable is a word)
w *= 999        ; the result will be -15188  (the multiplication stays within a word,␣
↪but overflows)
```

The compiler does NOT warn about this! It's doing this for performance reasons - so you won't get sudden 16 bit (or even float) calculations where you needed only simple fast byte arithmetic. If you do need the extended resulting value, cast at least one of the operands explicitly to the larger datatype. For example:

```
byte b = 44
w = (b as word)*55
w = b*(55 as word)
```

5.2.9 Subroutines

Defining a subroutine

Subroutines are parts of the code that can be repeatedly invoked using a subroutine call from elsewhere. Their definition, using the `sub` statement, includes the specification of the required parameters and return value. Subroutines can be defined in a Block, but also nested inside another subroutine. Everything is scoped accordingly. With `asmsub` you can define a low-level subroutine that is implemented directly in assembly and takes parameters directly in registers.

Trivial `asmsub` routines can be tagged as `inline` to tell the compiler to copy their code in-place to the locations where the subroutine is called, rather than inserting an actual call and return to the subroutine. This may increase code size significantly and can only be used in limited scenarios, so YMMV. Note that the routine's code is copied verbatim into the place of the subroutine call in this case, so pay attention to any jumps and rts instructions in the inlined code! Inlining regular Prog8 subroutines is at the discretion of the compiler.

Calling a subroutine

The arguments in parentheses after the function name, should match the parameters in the subroutine definition. If you want to ignore a return value of a subroutine, you should prefix the call with the `void` keyword. Otherwise the compiler will issue a warning about discarding a result value.

Deeply nested function calls can be rewritten as a chain using the *pipe operator* `|>` as long as they are unary functions (taking a single argument). Various possibilities of using this operator are explained in the syntax reference for this operator.

Note: Order of evaluation:

The order of evaluation of arguments to a single function call is *unspecified* and should not be relied upon. There is no guarantee of a left-to-right or right-to-left evaluation of the call arguments.

Caution: Note that due to the way parameters are processed by the compiler, subroutines are *non-reentrant*. This means you cannot create recursive calls. If you do need a recursive algorithm, you'll have to hand code it in embedded assembly for now, or rewrite it into an iterative algorithm. Also, subroutines used in the main program

should not be used from an IRQ handler. This is because the subroutine may be interrupted, and will then call itself from the IRQ handler. Results are then undefined because the variables will get overwritten.

5.2.10 Built-in Functions

There's a set of predefined functions in the language. These are fixed and can't be redefined in user code. You can use them in expressions and the compiler will evaluate them at compile-time if possible.

Math

abs(x) Absolute value of an integer. For floating point numbers, use `floats.fabs()` instead.

sgn(x) Get the sign of the value. Result is -1, 0 or 1 (negative, zero, positive).

sqrt16(w) 16 bit unsigned integer Square root. Result is unsigned byte. To do the reverse, squaring an integer, just write `x*x`.

Array operations

any(x) 1 ('true') if any of the values in the array value x is 'true' (not zero), else 0 ('false')

all(x) 1 ('true') if all of the values in the array value x are 'true' (not zero), else 0 ('false')

len(x) Number of values in the array value x, or the number of characters in a string (excluding the 0-byte). Note: this can be different from the number of *bytes* in memory if the datatype isn't a byte. See sizeof(). Note: lengths of strings and arrays are determined at compile-time! If your program modifies the actual length of the string during execution, the value of len(s) may no longer be correct! (use the `string.length` routine if you want to dynamically determine the length by counting to the first 0-byte)

reverse(array) Reverse the values in the array (in-place). Can be used after sort() to sort an array in descending order.

sort(array) Sort the array in ascending order (in-place) Supported are arrays of bytes or word values. Sorting a floating-point array is not supported right now, as a general sorting routine for this will be extremely slow. Either build one yourself or find another solution that doesn't require sorting. Finally, note that sorting an array with strings in it will not do what you might think; it considers the array as just an array of integer words and sorts the string *pointers* accordingly. Sorting strings alphabetically has to be programmed yourself if you need it.

Miscellaneous

cmp(x,y) Compare the integer value x to integer value y. Doesn't return a value or boolean result, only sets the processor's status bits! You can use a conditional jumps (`if_cc` etcetera) to act on this. Normally you should just use a comparison expression (`x < y`)

lsb(x) Get the least significant byte of the word x. Equivalent to the cast "x as ubyte".

msb(x) Get the most significant byte of the word x.

mkword(msb, lsb) Efficiently create a word value from two bytes (the msb and the lsb). Avoids multiplication and shifting. So mkword($80, $22) results in $8022.

Note: The arguments to the mkword() function are in 'natural' order that is first the msb then the lsb. Don't get confused by how the system actually stores this 16-bit word value in memory (which is in little-endian format, so lsb first then msb)

---

peek(address) same as @(address) - reads the byte at the given address in memory.

peekw(address) reads the word value at the given address in memory. Word is read as usual little-endian lsb/msb byte order.

poke(address, value) same as @(address)=value - writes the byte value at the given address in memory.

pokew(address, value) writes the word value at the given address in memory, in usual little-endian lsb/msb byte order.

pokemon(address, value) Attempts to write a byte to a ROM at a location in machine language monitor bank. Doesn't have anything to do with a certain video game.

push(value) pushes a byte value on the CPU hardware stack. Lowlevel function that should normally not be used.

pushw(value) pushes a 16-bit word value on the CPU hardware stack. Lowlevel function that should normally not be used.

pop(variable) pops a byte value off the CPU hardware stack into the given variable. Only variables can be used. Lowlevel function that should normally not be used.

popw(value) pops a 16-bit word value off the CPU hardware stack into the given variable. Only variables can be used. Lowlevel function that should normally not be used.

rnd() returns a pseudo-random byte from 0..255

rndw() returns a pseudo-random word from 0..65535

rol(x) Rotate the bits in x (byte or word) one position to the left. This uses the CPU's rotate semantics: bit 0 will be set to the current value of the Carry flag, while the highest bit will become the new Carry flag value. (essentially, it is a 9-bit or 17-bit rotation) Modifies in-place, doesn't return a value (so can't be used in an expression). You can rol a memory location directly by using the direct memory access syntax, so like `rol(@($5000))`

rol2(x) Like `rol` but now as 8-bit or 16-bit rotation. It uses some extra logic to not consider the carry flag as extra rotation bit. Modifies in-place, doesn't return a value (so can't be used in an expression). You can rol a memory location directly by using the direct memory access syntax, so like `rol2(@($5000))`

ror(x) Rotate the bits in x (byte or word) one position to the right. This uses the CPU's rotate semantics: the highest bit will be set to the current value of the Carry flag, while bit 0 will become the new Carry flag value. (essentially, it is a 9-bit or 17-bit rotation) Modifies in-place, doesn't return a value (so can't be used in an expression). You can ror a memory location directly by using the direct memory access syntax, so like `ror(@($5000))`

ror2(x) Like `ror` but now as 8-bit or 16-bit rotation. It uses some extra logic to not consider the carry flag as extra rotation bit. Modifies in-place, doesn't return a value (so can't be used in an expression). You can ror a memory location directly by using the direct memory access syntax, so like `ror2(@($5000))`

sizeof(name) Number of bytes that the object 'name' occupies in memory. This is a constant determined by the data type of the object. For instance, for a variable of type uword, the sizeof is 2. For an 10 element array of floats, it is 50 (on the C-64, where a float is 5 bytes). Note: usually you will be interested in the number of elements in an array, use len() for that.

swap(x, y) Swap the values of numerical variables (or memory locations) x and y in a fast way. You can swap two memory locations directly by using the direct memory access syntax, so like `swap(@($5000), @($5001))`

memory(name, size, alignment) Returns the address of the first location of a statically "reserved" block of memory of the given size in bytes, with the given name. If you specify an alignment value >1, it means the block of memory will be aligned to such a dividable address in memory, for instance an alignment of $100 means the memory block is aligned on a page boundary, and $2 means word aligned (even addresses). Requesting the address of such a named memory block again later with the same name, will result in the same address as before. When reusing blocks in that way, it is required that the size argument is the same, otherwise you'll get a compilation error. This routine can be used to "reserve" parts of the memory where a normal byte array variable would not suffice; for instance if you need more than 256 consecutive bytes. The return value is just a simple uword address so it cannot be used as an array in your program. You can only treat it as a pointer or use it in inline assembly.

callfar(bank, address, argumentaddress) ; NOTE: specific to cx16 target for now Calls an assembly routine in another ram-bank on the CommanderX16 (using the `jsrfar` routine) The banked RAM is located in the address range $A000-$BFFF (8 kilobyte). Notice that bank $00 is used by the Kernal and should not be used by user code. The third argument can be used to designate the memory address of an argument for the routine; it will be loaded into the A register and will receive the result value returned by the routine in the A register. If you leave this at zero, no argument passing will be done. If the routine requires different arguments or return values, `callfar` cannot be used and you'll have to set up a call to `jsrfar` yourself to process this.

callrom(bank, address, argumentaddress) ; NOTE: specific to cx16 target for now Calls an assembly routine in another rom-bank on the CommanderX16 The banked ROM is located in the address range $C000-$FFFF (16 kilobyte). There are 32 banks (0 to 31). The third argument can be used to designate the memory address of an argument for the routine; it will be loaded into the A register and will receive the result value returned by the routine in the A register. If you leave this at zero, no argument passing will be done. If the routine requires different arguments or return values, `callrom` cannot be used and you'll have to set up a call in assembly code yourself that handles the banking and argument/returnvalues.

syscall(callnr), syscall1(callnr, arg), syscall2(callnr, arg1, arg2), syscall3(callnr, arg1, arg2, arg3) Functions for doing a system call on targets that support this. Currently no actual target uses this though except, possibly, the experimental code generation target! The regular 6502 based compiler targets just use a gosub to asmsub kernal routines at specific memory locations. So these builtin function calls are not useful yet except for experimentation in new code generation targets.

rsave, rsavex Saves all registers including status (or only X) on the stack It's not needed to rsave()/rsavex() before an asm subroutine that clobbers the X register (which is used by prog8 as the internal evaluation stack pointer); the compiler will take care of this situation automatically. Note: the 16 bit 'virtual' registers of the Commander X16 are *not* saved.

rrestore, rrestorex Restore all registers including status (or only X) back from the cpu hardware stack Note: the 16 bit 'virtual' registers of the Commander X16 are *not* restored.

5.2.11 Library routines

There are many routines available in the compiler libraries. Some are used internally by the compiler as well. There's too many to list here, just have a look through the source code of the library modules to see what's there. (They can be found in the compiler/res directory) The example programs also use a small set of the library routines, you can study their source code to see how they might be used.

5.3 Syntax Reference

5.3.1 Module file

This is a file with the `.p8` suffix, containing *directives* and *code blocks*, described below. The file is a text file wich can also contain:

Lines, whitespace, indentation

Line endings are significant because *only one* declaration, statement or other instruction can occur on every line. Other whitespace and line indentation is arbitrary and ignored by the compiler. You can use tabs or spaces as you wish.

Source code comments

Everything after a semicolon `;` is a comment and is ignored. If the whole line is just a comment, it will be copied into the resulting assembly source code. This makes it easier to understand and relate the generated code. Examples:

```
counter = 42    ; set the initial value to 42
; next is the code that...
```

5.3.2 Directives

`%output <type>`

Level: module. Global setting, selects program output type. Default is `prg`.

- type `raw` : no header at all, just the raw machine code data
- type `prg` : C64 program (with load address header)

`%launcher <type>`

Level: module. Global setting, selects the program launcher stub to use. Only relevant when using the `prg` output type. Defaults to `basic`.

- type `basic` : add a tiny C64 BASIC program, whith a SYS statement calling into the machine code
- type `none` : no launcher logic is added at all

`%zeropage <style>`

Level: module. Global setting, select ZeroPage handling style. Defaults to `kernalsafe`.

- style `kernalsafe` – use the part of the ZP that is 'free' or only used by BASIC routines, and don't change anything else. This allows full use of KERNAL ROM routines (but not BASIC routines), including default IRQs during normal system operation. It's not possible to return cleanly to BASIC when the program exits. The only choice is to perform a system reset. (A `system_reset` subroutine is available in the syslib to help you do this)
- style `floatsafe` – like the previous one but also reserves the addresses that are required to perform floating point operations (from the BASIC kernal). No clean exit is possible.
- style `basicsafe` – the most restricted mode; only use the handful 'free' addresses in the ZP, and don't touch change anything else. This allows full use of BASIC and KERNAL ROM routines including default IRQs during normal system operation. When the program exits, it simply returns to the BASIC ready prompt.
- style `full` – claim the whole ZP for variables for the program, overwriting everything, except the few addresses mentioned above that are used by the system's IRQ routine. Even though the default IRQ routine is still active, it is impossible to use most BASIC and KERNAL ROM routines. This includes many floating point operations and several utility routines that do I/O, such as `print`. This option makes programs smaller and faster because even more variables can be stored in the ZP (which allows for more efficient assembly code). It's not possible to return cleanly to BASIC when the program exits. The only choice is to perform a system reset. (A `system_reset` subroutine is available in the syslib to help you do this)
- style `dontuse` – don't use *any* location in the zeropage.

Also read *ZeroPage ("ZP")*.

%zpreserved <fromaddress>,<toaddress>

Level: module. Global setting, can occur multiple times. It allows you to reserve or 'block' a part of the zeropage so that it will not be used by the compiler.

%address <address>

Level: module. Global setting, set the program's start memory address. It's usually fixed at `$0801` because the default launcher type is a CBM-basic program. But you have to specify this address yourself when you don't use a CBM-basic launcher.

%import <name>

Level: module. This reads and compiles the named module source file as part of your current program. Symbols from the imported module become available in your code, without a module or filename prefix. You can import modules one at a time, and importing a module more than once has no effect.

%option <option> [, <option> ...]

Level: module, block. Sets special compiler options.

- `enable_floats` (module level) tells the compiler to deal with floating point numbers (by using various subroutines from the Commodore-64 kernal). Otherwise, floating point support is not enabled. Normally you don't have to use this yourself as importing the `floats` library is required anyway and that will enable it for you automatically.
- `no_sysinit` (module level) which cause the resulting program to *not* include the system re-initialization logic of clearing the screen, resetting I/O config etc. You'll have to take care of that yourself. The program will just start running from whatever state the machine is in when the program was launched.
- `force_output` (in a block) will force the block to be outputted in the final program. Can be useful to make sure some data is generated that would otherwise be discarded because the compiler thinks it's not referenced (such as sprite data)
- `align_word` (in a block) will make the assembler align the start address of this block on a word boundary in memory (so, an even memory address).
- `align_page` (in a block) will make the assembler align the start address of this block on a page boundary in memory (so, the LSB of the address is 0).
- `merge` (in a block) will merge this block's contents into an already existing block with the same name. Useful in library scenarios.

%asmbinary "<filename>" [, <offset>[, <length>]]

Level: not at module scope. This directive can only be used inside a block. The assembler will include the file as binary bytes at this point, prog8 will not process this at all. The optional offset and length can be used to select a particular piece of the file. The file is located relative to the current working directory! To reference the contents of the included binary data, you can put a label in your prog8 code just before the %asmbinary. An example program for this can be found below at the description of %asminclude.

%asminclude "<filename>"

Level: not at module scope. This directive can only be used inside a block. The assembler will include the file as raw assembly source text at this point, prog8 will not process this at all. Symbols defined in the included assembly can not be referenced from prog8 code. However they can be referenced from other assembly code if properly prefixed. You can ofcourse use a label in your prog8 code just before the %asminclude directive, and reference that particular label to get to (the start of) the included assembly. Be careful: you risk symbol redefinitions or duplications if you include a piece of assembly into a prog8 block that already defines symbols itself. The compiler first looks for the file relative to the same directory as the module containing this statement is in, if the file can't be found there it is searched relative to the current directory. Here is a small example program to show how to use labels to reference the included contents from prog8 code:

```
%import textio
%zeropage basicsafe

main {

    sub start() {
        txt.print("first three bytes of included asm:\n")
        uword included_addr = &included_asm
        txt.print_ub(@(included_addr))
        txt.spc()
        txt.print_ub(@(included_addr+1))
        txt.spc()
        txt.print_ub(@(included_addr+2))

        txt.print("\nfirst three bytes of included binary:\n")
        included_addr = &included_bin
        txt.print_ub(@(included_addr))
        txt.spc()
        txt.print_ub(@(included_addr+1))
        txt.spc()
        txt.print_ub(@(included_addr+2))
        txt.nl()
        return

included_asm:
        %asminclude "inc.asm"

included_bin:
        %asmbinary "inc.bin"

    }
}
```

%breakpoint

Level: not at module scope. Defines a debugging breakpoint at this location. See *Debugging (with Vice)*

%asm {{ ... }}

Level: not at module scope. Declares that a piece of *assembly code* is inside the curly braces. This code will be copied as-is into the generated output assembly source file. The assembler syntax used should be for the 3rd party cross assembler tool that Prog8 uses (64tass). Note that the start and end markers are both *double curly braces* to minimize the chance that the assembly code itself contains either of those. If it does contain a `}}`, it will confuse the parser.

5.3.3 Identifiers

Naming things in Prog8 is done via valid *identifiers*. They start with a letter, and after that, a combination of letters, numbers, or underscores. Examples of valid identifiers:

```
a
A
monkey
COUNTER
Better_Name_2
something_strange__
```

5.3.4 Code blocks

A named block of actual program code. Itefines a *scope* (also known as 'namespace') and can only contain *directives*, *variable declarations*, *subroutines* or *inline assembly*:

```
<blockname> [<address>] {
    <directives>
    <variables>
    <subroutines>
    <inline asm>
}
```

The <blockname> must be a valid identifier. The <address> is optional. If specified it must be a valid memory address such as `$c000`. It's used to tell the compiler to put the block at a certain position in memory. Also read *Blocks, Scopes, and accessing Symbols*. Here is an example of a code block, to be loaded at `$c000`:

```
main $c000 {
        ; this is code inside the block...
}
```

5.3.5 Labels

To label a position in your code where you can jump to from another place, you use a label:

```
nice_place:
                ; code ...
```

It's just an identifier followed by a colon `:`. It's allowed to put the next statement on the same line, after the label.

5.3.6 Variables and value literals

The data that the code works on is stored in variables. Variable names have to be valid identifiers. Values in the source code are written using *value literals*. In the table of the supported data types below you can see how they should be written.

Variable declarations

Variables should be declared with their exact type and size so the compiler can allocate storage for them. You can give them an initial value as well. That value can be a simple literal value, or an expression. If you don't provide an intial value yourself, zero will be used. You can add a `@zp` zeropage-tag, to tell the compiler to prioritize it when selecting variables to be put into zeropage (but no guarantees). If the ZP is full, the variable will be allocated in normal memory elsewhere. Use the `@requirezp` tag to force the variable in zeropage, but if the ZP is full, the compilation will fail. You can add a `@shared` shared-tag, to tell the compiler that the variable is shared with some assembly code and that it should not be optimized away if not used elsewhere. The syntax is:

```
<datatype>  [ @shared ] [ @zp ] [ @requirezp ]  <variable name>   [ = <initial value> ]
```

Various examples:

```
word        thing   = 0
byte        counter = len([1, 2, 3]) * 20
byte        age     = 2018 - 1974
float       wallet  = 55.25
str         name    = "my name is Alice"
uword       address = &counter
byte[]      values  = [11, 22, 33, 44, 55]
byte[5]     values                  ; array of 5 bytes, initially set to zero
byte[5]     values  = 255           ; initialize with five 255 bytes

word  @zp         zpword = 9999     ; prioritize this when selecting vars for zeropage␣
↪storage
uword @requirezp  zpaddr = $3000    ; we require this variable in Zeropage
word  @shared asmvar                ; variable is used in assembly code but not elsewhere
```

Data types

Prog8 supports the following data types:

type identifier	type	storage size	example var declaration and literal value
`byte`	signed byte	1 byte = 8 bits	`byte myvar = -22`
`ubyte`	unsigned byte	1 byte = 8 bits	`ubyte myvar = $8f`, `ubyte c = 'a'`, `ubyte c2 = @'a'`
–	boolean	1 byte = 8 bits	`byte myvar = true` or `byte myvar == false` The true and false are actually just aliases for the byte values 1 and 0.
`word`	signed word	2 bytes = 16 bits	`word myvar = -12345`
`uword`	unsigned word	2 bytes = 16 bits	`uword myvar = $8fee`
`float`	floating-point	5 bytes = 40 bits	`float myvar = 1.2345` stored in 5-byte cbm MFLPT format
`byte[x]`	signed byte array	x bytes	`byte[4] myvar`
`ubyte[x]`	unsigned byte array	x bytes	`ubyte[4] myvar`
`word[x]`	signed word array	2*x bytes	`word[4] myvar`
`uword[x]`	unsigned word array	2*x bytes	`uword[4] myvar`
`float[x]`	floating-point array	5*x bytes	`float[4] myvar`
`byte[]`	signed byte array	depends on value	`byte[] myvar = [1, 2, 3, 4]`
`ubyte[]`	unsigned byte array	depends on value	`ubyte[] myvar = [1, 2, 3, 4]`
`word[]`	signed word array	depends on value	`word[] myvar = [1, 2, 3, 4]`
`uword[]`	unsigned word array	depends on value	`uword[] myvar = [1, 2, 3, 4]`
`float[]`	floating-point array	depends on value	`float[] myvar = [1.1, 2.2, 3.3, 4.4]`
`str[]`	array with string ptrs	2*x bytes + strs	`str[] names = ["ally", "pete"]`
`str`	string (petscii)	varies	`str myvar = "hello."` implicitly terminated by a 0-byte

arrays: you can split an array initializer list over several lines if you want. When an initialization value is given, the array size in the declaration can be omitted.

hexadecimal numbers: you can use a dollar prefix to write hexadecimal numbers: `$20ac`

binary numbers: you can use a percent prefix to write binary numbers: `%10010011` Note that `%` is also the remainder operator so be careful: if you want to take the remainder of something with an operand starting with 1 or 0, you'll have to add a space in between.

character values: you can use a single character in quotes like this `'a'` for the Petscii byte value of that character.

``byte`` versus ``word`` values:

- When an integer value ranges from 0..255 the compiler sees it as a `ubyte`. For -128..127 it's a `byte`.

- When an integer value ranges from 256..65535 the compiler sees it as a `uword`. For -32768..32767 it's a `word`.
- When a hex number has 3 or 4 digits, for example `$0004`, it is seen as a `word` otherwise as a `byte`.
- When a binary number has 9 to 16 digits, for example `%1100110011`, it is seen as a `word` otherwise as a `byte`.
- If the number fits in a byte but you really require it as a word value, you'll have to explicitly cast it: `60 as uword` or you can use the full word hexadecimal notation `$003c`.

Data type conversion

Many type conversions are possible by just writing `as <type>` at the end of an expression, for example `ubyte ub = floatvalue as ubyte` will convert the floating point value to an unsigned byte.

Memory mapped variables

The & (address-of operator) used in front of a data type keyword, indicates that no storage should be allocated by the compiler. Instead, the (mandatory) value assigned to the variable should be the *memory address* where the value is located:

```
&byte BORDERCOLOR = $d020
&ubyte[5*40]  top5screenrows = $0400        ; works for array as well
```

Direct access to memory locations

Instead of defining a memory mapped name for a specific memory location, you can also directly access the memory. Enclose a numeric expression or literal with `@(...)` to do that:

```
color = @($d020)  ; set the variable 'color' to the current c64 screen border color (
↪"peek(53280)")
@($d020) = 0      ; set the c64 screen border to black ("poke 53280,0")
@(vic+$20) = 6    ; a dynamic expression to 'calculate' the address
```

The array indexing notation on a uword 'pointer variable' is syntactic sugar for such a direct memory access expression:

```
pointervar[999] = 0     ; equivalent to @(pointervar+999) = 0
```

Constants

All variables can be assigned new values unless you use the `const` keyword. The initial value must be known at compile time (it must be a compile time constant expression). This is only valid for the simple numeric types (byte, word, float):

```
const  byte  max_age = 99
```

Reserved names

The following names are reserved, they have a special meaning:

```
true  false             ; boolean values 1 and 0
```

Range expression

A special value is the *range expression* which represents a range of integer numbers or characters, from the starting value to (and including) the ending value:

```
<start>  to  <end>   [ step  <step> ]
<start>  downto  <end>   [ step  <step> ]
```

You an provide a step value if you need something else than the default increment which is one (or, in case of downto, a decrement of one). Because a step of minus one is so common you can just use the downto variant to avoid having to specify the step as well.

If used in the place of a literal value, it expands into the actual array of integer values:

```
byte[] array = 100 to 199     ; initialize array with [100, 101, ..., 198, 199]
```

Array indexing

Strings and arrays are a sequence of values. You can access the individual values by indexing. Syntax is familiar with brackets: `arrayvar[x]`

```
array[2]        ; the third byte in the array (index is 0-based)
string[4]       ; the fifth character (=byte) in the string
```

Note: you can also use array indexing on a 'pointer variable', which is basically an uword variable containing a memory address. Currently this is equivalent to directly referencing the bytes in memory at the given index. See *Direct access to memory locations*

String

A string literal can occur with or without an encoding prefix (encoding followed by ':' followed by the string itself). When this is omitted, the string is stored in the machine's default character encoding (which is PETSCII on the CBM machines). You can choose to store the string in other encodings such as `sc` (screencodes) or `iso` (iso-8859-15). String length is limited to 255 characters. Here are several examples:

- `"hello"` a string translated into the default character encoding (PETSCII)
- `petscii:"hello"` same as the above, on CBM machines.
- `sc:"my name is Alice"` string with screencode encoding (new syntax)
- `iso:"Ich heiße François"` string in iso encoding

There are several escape sequences available to put special characters into your string value:

- `\\` - the backslash itself, has to be escaped because it is the escape symbol by itself
- `\n` - newline character (move cursor down and to beginning of next line)
- `\r` - carriage return character (more or less the same as newline if printing to the screen)

- \" - quote character (otherwise it would terminate the string)
- \' - apostrophe character (has to be escaped in character literals, is okay inside a string)
- \uHHHH - a unicode codepoint u0000 - uffff (16-bit hexadecimal)
- \xHH - 8-bit hex value that will be copied verbatim *without encoding*
- String literals can contain many symbols directly if they have a petscii equivalent, such as "". Characters like ^, _, \, {, } and | (that have no direct PETSCII counterpart) are still accepted and converted to the closest PETSCII equivalents. (Make sure you save the source file in UTF-8 encoding if you use this.)

5.3.7 Operators

arithmetic: + - * / % +, -, *, / are the familiar arithmetic operations. / is division (will result in integer division when using on integer operands, and a floating point division when at least one of the operands is a float) % is the remainder operator: `25 % 7` is 4. Be careful: without a space, %10 will be parsed as the binary number 2. Remainder is only supported on integer operands (not floats).

bitwise arithmetic: & | ^ ~ << >> & is bitwise and, | is bitwise or, ^ is bitwise xor, ~ is bitwise invert (this one is an unary operator) << is bitwise left shift and >> is bitwise right shift (both will not change the datatype of the value)

assignment: = Sets the target on the LHS (left hand side) of the operator to the value of the expression on the RHS (right hand side). Note that an assignment sometimes is not possible or supported.

augmented assignment: += -= *= /= **= &= |= ^= <<= >>= This is syntactic sugar; `aa += xx` is equivalent to `aa = aa + xx`

postfix increment and decrement: ++ -- Syntactic sugar; aa++ is equivalent to `aa = aa + 1`, and aa-- is equivalent to `aa = aa - 1`. Because these operations are so common, we have these short forms.

comparison: != < > <= >= Equality, Inequality, Less-than, Greater-than, Less-or-Equal-than, Greater-or-Equal-than comparisons. The result is a 'boolean' value 'true' or 'false' (which in reality is just a byte value of 1 or 0).

logical: not and or xor These operators are the usual logical operations that are part of a logical expression to reason about truths (boolean values). The result of such an expression is a 'boolean' value 'true' or 'false' (which in reality is just a byte value of 1 or 0).

Note: Unlike most other programming languages, there is no short-cirquit or McCarthy-evaluation for the and and or operators at this time. This means that prog8 currently always evaluates all operands from these logical expressions, even when one of them already determines the outcome. This may be changed in a future language version.

---

range creation: to Creates a range of values from the LHS value to the RHS value, inclusive. These are mainly used in for loops to set the loop range. Example:

```
    0 to 7          ; range of values 0, 1, 2, 3, 4, 5, 6, 7  (constant)

    aa = 5
    aa = 10
aa to xx            ; range of 5, 6, 7, 8, 9, 10

    byte[] array = 10 to 13   ; sets the array to [1, 2, 3, 4]

    for  i  in  0 to 127  {
```

(continues on next page)

(continued from previous page)

```
            ; i loops 0, 1, 2, ... 127
    }
```

containment check: `in` Tests if a value is present in a list of values, which can be a string or an array. The result is a simple boolean `true` or `false`. Consider using this instead of chaining multiple value tests with `or`, because the containment check is more efficient. Examples:

```
ubyte cc
if cc in [' ', '@', 0] {
    txt.print("cc is one of the values")
}

str  email_address = "?????????"
if '@' in email_address {
    txt.print("email address seems ok")
}
```

pipe: `|>` Used as an alternative to nesting function calls. The pipe operator is used to 'pipe' the value into the next function. You write a pipe as a sequence of function calls. You don't write the arguments to the functions though: the value of one segment in the pipe, will be used as the argument for the next function call in the sequence.

note: It only works on unary functions (taking a single argument) for now.

For example, this: `txt.print_uw(add_bonus(determine_score(get_player(1))))` can be rewritten as:

```
get_player(1)
    |> determine_score()
    |> add_bonus()
    |> txt.print_uw()
```

A pipe can also be written as an expression that returns a value, for example `uword score = add_bonus(determine_score(get_player(1)))`

```
uword score = get_player(1)
                |> determine_score()
                |> add_bonus()
```

address of: `&` This is a prefix operator that can be applied to a string or array variable or literal value. It results in the memory address (UWORD) of that string or array in memory: `uword a = &stringvar` Sometimes the compiler silently inserts this operator to make it easier for instance to pass strings or arrays as subroutine call arguments. This operator can also be used as a prefix to a variable's data type keyword to indicate that it is a memory mapped variable (for instance: `&ubyte screencolor = $d021`)

precedence grouping in expressions, or subroutine parameter list: `(` *expression* `)` Parentheses are used to group parts of an expression to change the order of evaluation. (the subexpression inside the parentheses will be evaluated first): `(4 + 8) * 2` is 24 instead of 20.

Parentheses are also used in a subroutine call, they follow the name of the subroutine and contain the list of arguments to pass to the subroutine: `big_function(1, 99)`

5.3.8 Subroutine / function calls

You call a subroutine like this:

```
[ void / result = ] subroutinename_or_address ( [argument...] )

; example:
resultvariable = subroutine(arg1, arg2, arg3)
void noresultvaluesub(arg)
```

Arguments are separated by commas. The argument list can also be empty if the subroutine takes no parameters. If the subroutine returns a value, usually you assign it to a variable. If you're not interested in the return value, prefix the function call with the `void` keyword. Otherwise the compiler will warn you about discarding the result of the call.

Multiple return values

Normal subroutines can only return zero or one return values. However, the special `asmsub` routines (implemented in assembly code) or `romsub` routines (referencing a routine in kernal ROM) can return more than one return value. For example a status in the carry bit and a number in A, or a 16-bit value in A/Y registers. It is not possible to process the results of a call to these kind of routines directly from the language, because only single value assignments are possible. You can still call the subroutine and not store the results.

There is an exception: if there's just one return value in a register, and one or more others that are returned as bits in the status register (such as the Carry bit), the compiler allows you to call the subroutine. It will then store the result value in a variable if required, and *try to keep the status register untouched after the call* so you can often use a conditional branch statement for that. But the latter is tricky, make sure you check the generated assembly code.

If there really are multiple relevant return values (other than a combined 16 bit return value in 2 registers), you'll have to write a small block of custom inline assembly that does the call and stores the values appropriately. Don't forget to save/restore any registers that are modified.

5.3.9 Subroutine definitions

The syntax is:

```
sub   <identifier>  ( [parameters] )  [ -> returntype ]  {
        ... statements ...
}

; example:
sub  triple_something (word amount) -> word  {
        return  X * 3
}
```

The open curly brace must immediately follow the subroutine result specification on the same line, and can have nothing following it. The close curly brace must be on its own line as well. The parameters is a (possibly empty) comma separated list of "<datatype> <parametername>" pairs specifying the input parameters. The return type has to be specified if the subroutine returns a value.

Assembly / ROM subroutines

Subroutines implemented in ROM are usually defined by compiler library files, with the following syntax:

```
romsub $FFD5 = LOAD(ubyte verify @ A, uword address @ XY) -> clobbers() -> ubyte @Pc,␣
↪ubyte @ A, ubyte @ X, ubyte @ Y
```

This defines the `LOAD` subroutine at ROM memory address $FFD5, taking arguments in all three registers A, X and Y, and returning stuff in several registers as well. The `clobbers` clause is used to signify to the compiler what CPU registers are clobbered by the call instead of being unchanged or returning a meaningful result value.

User subroutines in the program source code that are implemented purely in assembly and which have an assembly calling convention (i.e. the parameters are strictly passed via cpu registers), are defined with `asmsub` like this:

```
asmsub  clear_screenchars (ubyte char @ A) clobbers(Y)  {
    %asm {{
        ldy  #0
_loop   sta  c64.Screen,y
        sta  c64.Screen+$0100,y
        sta  c64.Screen+$0200,y
        sta  c64.Screen+$02e8,y
        iny
        bne  _loop
        rts
        }}
}
```

the statement body of such a subroutine should consist of just an inline assembly block.

The `@ <register>` part is required for rom and assembly-subroutines, as it specifies for the compiler what cpu registers should take the routine's arguments. You can use the regular set of registers (A, X, Y), the special 16-bit register pairs to take word values (AX, AY and XY) and even a processor status flag such as Carry (Pc).

Note: Asmsubs can also be tagged as `inline asmsub` to make trivial pieces of assembly inserted directly instead of a call to them. Note that it is literal copy-paste of code that is done, so make sure the assembly is actually written to behave like such - which probably means you don't want a `rts` or `jmp` or `bra` in it!

Note: The 'virtual' 16-bit registers from the Commander X16 can also be specified as `R0` .. `R15` . This means you don't have to set them up manually before calling a subroutine that takes one or more parameters in those 'registers'. You can just list the arguments directly. *This also works on the Commodore-64!* (however they are not as efficient there because they're not in zeropage) In prog8 and assembly code these 'registers' are directly accessible too via `cx16.r0` .. `cx16.r15` (these are memory mapped uword values), `cx16.r0s` .. `cx16.r15s` (these are memory mapped word values), and `L` / `H` variants are also available to directly access the low and high bytes of these.

5.3.10 Expressions

Expressions calculate a value and can be used almost everywhere a value is expected. They consist of values, variables, operators, function calls, type casts, direct memory reads, and can be combined into other expressions. Long expressions can be split over multiple lines by inserting a line break before or after an operator:

```
num_hours * 3600
 + num_minutes * 60
 + num_seconds
```

5.3.11 Loops

for loop

The loop variable must be a byte or word variable, and it must be defined separately first. The expression that you loop over can be anything that supports iteration (such as ranges like `0 to 100`, array variables and strings) *except* floating-point arrays (because a floating-point loop variable is not supported).

You can use a single statement, or a statement block like in the example below:

```
for <loopvar>  in  <expression>  [ step <amount> ]   {
        ; do something...
        break           ; break out of the loop
}
```

For example, this is a for loop using a byte variable `i`, defined before, to loop over a certain range of numbers:

```
ubyte i

...

for i in 20 to 155 {
    ; do something
}
```

And this is a loop over the values of the array `fibonacci_numbers`:

```
uword[] fibonacci_numbers = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,
↪ 987, 1597, 2584, 4181]

uword number
for number in fibonacci_numbers {
    ; do something with number
}
```

while loop

As long as the condition is true (1), repeat the given statement(s). You can use a single statement, or a statement block like in the example below:

```
while  <condition>  {
        ; do something...
        break           ; break out of the loop
}
```

do-until loop

Until the given condition is true (1), repeat the given statement(s). You can use a single statement, or a statement block like in the example below:

```
do  {
        ; do something...
        break           ; break out of the loop
} until  <condition>
```

repeat loop

When you're only interested in repeating something a given number of times. It's a short hand for a for loop without an explicit loop variable:

```
repeat 15 {
    ; do something...
    break           ; you can break out of the loop
}
```

If you omit the iteration count, it simply loops forever. You can still `break` out of such a loop if you want though.

5.3.12 Conditional Execution and Jumps

Unconditional jump

To jump to another part of the program, you use a `goto` statement with an addres or the name of a label or subroutine:

```
    goto  $c000             ; address
    goto  name              ; label or subroutine

uword address = $4000
goto  address   ; jump via address variable
```

Notice that this is a valid way to end a subroutine (you can either `return` from it, or jump to another piece of code that eventually returns).

If you jump to an address variable (uword), it is doing an 'indirect' jump: the jump will be done to the address that's currently in the variable.

Conditional execution

With the 'if' / 'else' statement you can execute code depending on the value of a condition:

```
if  <expression>  <statements>  [else  <statements> ]
```

where <statements> can be just a single statement for instance just a `goto`, or it can be a block such as this:

```
if  <expression> {
        <statements>
} else {
        <alternative statements>
}
```

Special status register branch form:

There is a special form of the if-statement that immediately translates into one of the 6502's branching instructions. It is almost the same as the regular if-statement but it lacks a contional expression part, because the if-statement itself defines on what status register bit it should branch on:

```
if_XX  <statements>  [else  <statements> ]
```

where <statements> can be just a single statement for instance just a `goto`, or it can be a block such as this:

```
if_XX {
        <statements>
} else {
        <alternative statements>
}
```

The XX corresponds to one of the processor's branching instructions, so the possibilities are: `if_cs`, `if_cc`, `if_eq`, `if_ne`, `if_pl`, `if_mi`, `if_vs` and `if_vc`. It can also be one of the four aliases that are easier to read: `if_z`, `if_nz`, `if_pos` and `if_neg`.

Caution: These special `if_XX` branching statements are only useful in certain specific situations where you are *certain* that the status register (still) contains the correct status bits. This is not always the case after a fuction call or other operations! If in doubt, check the generated assembly code!

when statement ('jump table')

The structure of a when statement is like this:

```
when <expression> {
    <value(s)> -> <statement(s)>
    <value(s)> -> <statement(s)>
    ...
    [ else -> <statement(s)> ]
}
```

The when-*value* can be any expression but the choice values have to evaluate to compile-time constant integers (bytes or words). The else part is optional. Choices can result in a single statement or a block of multiple statements in which case you have to use { } to enclose them:

```
when value {
    4 -> txt.print("four")
    5 -> txt.print("five")
    10,20,30 -> {
        txt.print("ten or twenty or thirty")
    }
    else -> txt.print("don't know")
}
```

5.4 Compiler library modules

The compiler provides several "built-in" library modules with useful subroutine and variables.

Some of these may be specific for a certain compilation target, or work slightly different, but some effort is put into making them available across compilation targets.

This means that as long as your program is only using the subroutines from these libraries and not using hardware- and/or system dependent code, and isn't hardcoding certain assumptions like the screen size, the exact same source program can be compiled for multiple different target platforms. Many of the example programs that come with Prog8 are written like this.

You can `%import` and use these modules explicitly, but the compiler may also import one or more of these library modules automatically as required.

Note: For full details on what is available in the libraries, please study their source code here: https://github.com/irmen/prog8/tree/master/compiler/res/prog8lib

---

Caution: The resulting compiled binary program *only works on the target machine it was compiled for*. You must recompile the program for every target you want to run it on.

5.4.1 syslib

The "system library" for your target machine. It contains many system-specific definitions such as ROM/kernal subroutine definitions, memory location constants, and utility subroutines.

Depending on the compilation target, other routines may also be available in here specific to that target. Best is to check the source code of the correct syslib module.

Many of these definitions overlap for the C64 and Commander X16 targets so it is still possible to write programs that work on both targets without modifications.

5.4.2 sys (part of syslib)

target A constant ubyte value designating the target machine that the program is compiled for. Notice that this is a compile-time constant value and is not determined on the system when the program is running. The following return values are currently defined:

- 16 = compiled for CommanderX16 with 65C02 CPU
- 64 = compiled for Commodore-64 with 6502/6510 CPU

exit(returncode) Immediately stops the program and exits it, with the returncode in the A register. Note: custom interrupt handlers remain active unless manually cleared first!

memcopy(from, to, numbytes) Efficiently copy a number of bytes from a memory location to another. *Warning:* can only copy *non-overlapping* memory areas correctly! Because this function imposes some overhead to handle the parameters, it is only faster if the number of bytes is larger than a certain threshold. Compare the generated code to see if it was beneficial or not. The most efficient will often be to write a specialized copy routine in assembly yourself!

memset(address, numbytes, bytevalue) Efficiently set a part of memory to the given (u)byte value. But the most efficient will always be to write a specialized fill routine in assembly yourself! Note that for clearing the screen, very fast specialized subroutines are available in the `textio` and `graphics` library modules.

memsetw(address, numwords, wordvalue) Efficiently set a part of memory to the given (u)word value. But the most efficient will always be to write a specialized fill routine in assembly yourself!

read_flags() -> ubyte Returns the current value of the CPU status register.

set_carry() Sets the CPU status register Carry flag.

clear_carry() Clears the CPU status register Carry flag.

set_irqd() Sets the CPU status register Interrupt Disable flag.

clear_irqd() Clears the CPU status register Interrupt Disable flag.

progend() Returns the last address of the program in memory + 1. Can be used to load dynamic data after the program, instead of hardcoding something.

wait(uword jiffies) wait approximately the given number of jiffies (1/60th seconds) note: the system irq handler has to be active for this to work as it depends on the system jiffy clock

waitvsync() busy wait till the next vsync has occurred (approximately), without depending on custom irq handling. can be used to avoid screen flicker/tearing when updating screen contents. note: a more accurate way to wait for vsync is to set up a vsync irq handler instead. note for cx16: the system irq handler has to be active for this to work (this is not required on c64)

waitrastborder() (c64/c128 targets only) busy wait till the raster position has reached the bottom screen border (approximately) can be used to avoid screen flicker/tearing when updating screen contents. note: a more accurate way to do this is by using a raster irq handler instead.

reset_system() Soft-reset the system back to initial power-on Basic prompt. (called automatically by Prog8 when the main subroutine returns and the program is not using basicsafe zeropage option)

5.4.3 conv

Routines to convert strings to numbers or vice versa.

- numbers to strings, in various formats (binary, hex, decimal)
- strings in decimal, hex and binary format into numbers (bytes, words)

5.4.4 textio (txt.*)

This will probably be the most used library module. It contains a whole lot of routines dealing with text-based input and output (to the screen). Such as

- printing strings and numbers
- reading text input from the user via the keyboard
- filling or clearing the screen and colors
- scrolling the text on the screen
- placing individual characters on the screen

5.4.5 diskio

Provides several routines that deal with disk drive I/O, such as:

- list files on disk, optionally filtering by a simple pattern with ? and *
- show disk directory as-is
- display disk drive status
- load and save data from and to the disk
- delete and rename files on the disk

5.4.6 string

Provides string manipulation routines.

`length(str) -> ubyte length` Number of bytes in the string. This value is determined during runtime and counts upto the first terminating 0 byte in the string, regardless of the size of the string during compilation time. Don't confuse this with `len` and `sizeof`

`left(source, length, target)` Copies the left side of the source string of the given length to target string. It is assumed the target string buffer is large enough to contain the result. Also, you have to make sure yourself that length is smaller or equal to the length of the source string. Modifies in-place, doesn't return a value (so can't be used in an expression).

`right(source, length, target)` Copies the right side of the source string of the given length to target string. It is assumed the target string buffer is large enough to contain the result. Also, you have to make sure yourself that length is smaller or equal to the length of the source string. Modifies in-place, doesn't return a value (so can't be used in an expression).

`slice(source, start, length, target)` Copies a segment from the source string, starting at the given index, and of the given length to target string. It is assumed the target string buffer is large enough to contain the result. Also, you have to make sure yourself that start and length are within bounds of the strings. Modifies in-place, doesn't return a value (so can't be used in an expression).

find(string, char) -> ubyte index + carry bit Locates the first position of the given character in the string, returns carry bit set if found and the index in the string. Or carry bit clear if the character was not found.

compare(string1, string2) -> ubyte result Returns -1, 0 or 1 depeding on wether string1 sorts before, equal or after string2. Note that you can also directly compare strings and string values with eachother using ==, < etcetera (it will use string.compare for you under water automatically).

copy(from, to) -> ubyte length Copy a string to another, overwriting that one. Returns the length of the string that was copied. Often you don't have to call this explicitly and can just write `string1 = string2` but this function is useful if you're dealing with addresses for instance.

lower(string) Lowercases the petscii-string in place.

upper(string) Uppercases the petscii-string in place.

5.4.7 floats

Provides definitions for the ROM/kernal subroutines and utility routines dealing with floating point variables. This includes `print_f`, the routine used to print floating point numbers, `fabs` to get the absolute value of a floating point number, and a dozen or so floating point math routines.

atan(x) Arctangent.

ceil(x) Rounds the floating point up to an integer towards positive infinity.

cos(x) Cosine. If you want a fast integer cosine, have a look at examples/cx16/sincos.p8 that contains various lookup tables generated by the 64tass assembler.

deg(x) Radians to degrees.

floor (x) Rounds the floating point down to an integer towards minus infinity.

ln(x) Natural logarithm (base e).

log2(x) Base 2 logarithm.

rad(x) Degrees to radians.

round(x) Rounds the floating point to the closest integer.

sin(x) Sine. If you want a fast integer sine, have a look at examples/cx16/sincos.p8 that contains various lookup tables generated by the 64tass assembler.

sqrt(x) Floating point Square root. To do the reverse, squaring a floating point number, just write `x*x` or `x**2`.

tan(x) Tangent.

rndf() returns a pseudo-random float between 0.0 and 1.0

5.4.8 graphics

Monochrome bitmap graphics routines, fixed 320*200 resolution:

- clearing the screen
- drawing individual pixels
- drawing lines, rectangles, filled rectangles, circles, discs

This library is available both on the C64 and the Cx16. It uses the ROM based graphics routines on the latter, and it is a very small library because of that. That also means though that it is constrained to 320*200 resolution on the Cx16 as well. Use the `gfx2` library if you want full-screen graphics or non-monochrome drawing (only on Cx16).

5.4.9 math

Low level math routines. You should not normally have to bother with this directly. The compiler needs it to implement most of the math operations in your programs.

However there's a bunch of integer trig functions in here too that use lookup tables to quickly calculate sine and cosines. Usually a custom lookup table is the way to go if your application needs this, but perhaps the provided ones can be of service too:

sin8u(x) Fast 8-bit ubyte sine of angle 0..255, result is in range 0..255

sin8(x) Fast 8-bit byte sine of angle 0..255, result is in range -127..127

sin16u(x) Fast 16-bit uword sine of angle 0..255, result is in range 0..65535

sin16(x) Fast 16-bit word sine of angle 0..255, result is in range -32767..32767

sinr8u(x) Fast 8-bit ubyte sine of angle 0..179 (each is a 2 degree step), result is in range 0..255 Angles 180..255 will yield a garbage result!

sinr8(x) Fast 8-bit byte sine of angle 0..179 (each is a 2 degree step), result is in range -127..127 Angles 180..255 will yield a garbage result!

sinr16u(x) Fast 16-bit uword sine of angle 0..179 (each is a 2 degree step), result is in range 0..65535 Angles 180..255 will yield a garbage result!

sinr16(x) Fast 16-bit word sine of angle 0..179 (each is a 2 degree step), result is in range -32767..32767 Angles 180..255 will yield a garbage result!

cos8u(x) Fast 8-bit ubyte cosine of angle 0..255, result is in range 0..255

cos8(x) Fast 8-bit byte cosine of angle 0..255, result is in range -127..127

cos16u(x) Fast 16-bit uword cosine of angle 0..255, result is in range 0..65535

cos16(x) Fast 16-bit word cosine of angle 0..255, result is in range -32767..32767

cosr8u(x) Fast 8-bit ubyte cosine of angle 0..179 (each is a 2 degree step), result is in range 0..255 Angles 180..255 will yield a garbage result!

cosr8(x) Fast 8-bit byte cosine of angle 0..179 (each is a 2 degree step), result is in range -127..127 Angles 180..255 will yield a garbage result!

cosr16u(x) Fast 16-bit uword cosine of angle 0..179 (each is a 2 degree step), result is in range 0..65535 Angles 180..255 will yield a garbage result!

cosr16(x) Fast 16-bit word cosine of angle 0..179 (each is a 2 degree step), result is in range -32767..32767 Angles 180..255 will yield a garbage result!

5.4.10 cx16logo

Just a fun module that contains the Commander X16 logo in PETSCII graphics and allows you to print it anywhere on the screen.

5.4.11 prog8_lib

Low level language support. You should not normally have to bother with this directly. The compiler needs it for various built-in system routines.

5.4.12 gfx2 (cx16 only)

Full-screen multicolor bitmap graphics routines, available on the Cx16 machine only.

- multiple full-screen resolutions: 640 * 480 monochrome, and 320 * 240 monochrome and 256 colors
- clearing screen, switching screen mode, also back to text mode is possible.
- drawing individual pixels
- drawing lines, rectangles, filled rectangles, circles, discs
- drawing text inside the bitmap
- in monochrome mode, it's possible to use a stippled drawing pattern to simulate a shade of gray.

5.4.13 palette (cx16 only)

Available for the Cx16 target. Various routines to set the display color palette. There are also a few better looking Commodore-64 color palettes available here, because the Commander X16's default colors for this (the first 16 colors) are too saturated and are quite different than how they looked on a VIC-II chip in a C-64.

5.4.14 cx16diskio (cx16 only)

Available for the Cx16 target. Contains extensions to the load and load_raw routines from the regular diskio module, to deal with loading of potentially large files in to banked ram (HiRam). Also contains a helper function to calculate the file size of a loaded file (although that is truncated to 16 bits, 64Kb)

5.5 Target system specification

Prog8 targets the following hardware:

- 8 bit MOS 6502/65c02/6510 CPU
- 64 Kb addressable memory (RAM or ROM)
- optional use of memory mapped I/O registers
- optional use of system ROM routines

Currently these machines can be selected as a compilation target (via the `-target` compiler argument):

- 'c64': the Commodore 64
- 'cx16': the Commander X16
- 'c128': the Commodore 128 (*limited support*)
- 'atari': the Atari 800 XL (*experimental support*)

This chapter explains some relevant system details of the c64 and cx16 machines.

Hint: If you only use standard kernal and prog8 library routines, it is possible to compile the *exact same program* for both machines (just change the compilation target flag)!

5.5.1 Memory Model

Physical address space layout

The 6502 CPU can address 64 kilobyte of memory. Most of the 64 kilobyte address space can be used by Prog8 programs. This is a hard limit: there is no built-in support for RAM expansions or bank switching.

memory area	type	note
`$00–$ff`	ZeroPage	contains many sensitive system variables
`$100–$1ff`	Hardware stack	used by the CPU, normally not accessed directly
`$0200–$ffff`	Free RAM or ROM	free to use memory area, often a mix of RAM and ROM

A few of these memory addresses are reserved and cannot be used for arbitrary data. They have a special hardware function, or are reserved for internal use in the code generated by the compiler:

reserved address	in use for
`$00`	data direction (CPU hw)
`$01`	bank select (CPU hw)
`$02`	internal scratch variable
`$03`	internal scratch variable
`$fb - $fc`	internal scratch variable
`$fd - $fe`	internal scratch variable
`$fffa - $fffb`	NMI vector (CPU hw)
`$fffc - $fffd`	RESET vector (CPU hw)
`$fffe - $ffff`	IRQ vector (CPU hw)

The actual machine will often have many other special addresses as well, For example, the Commodore-64 has:

- ROMs installed in the machine: BASIC, kernal and character roms. Occupying `$a000–$bfff` and `$e000–$ffff`.
- memory-mapped I/O registers, for the video and sound chips, and the CIA's. Occupying `$d000–$dfff`.
- RAM areas that are used for screen graphics and sprite data: usually at `$0400–$07ff`.

Prog8 programs can access all of those special memory locations but it will have a special meaning.

ZeroPage ("ZP")

The ZeroPage memory block `$02–$ff` can be regarded as 254 CPU 'registers', because they take less clock cycles to access and need fewer instruction bytes than accessing other memory locations outside of the ZP. Theoretically they can all be used in a program, with the follwoing limitations:

- several addresses (`$02`, `$03`, `$fb - $fc`, `$fd - $fe`) are reserved for internal use
- most other addresses will already be in use by the machine's operating system or kernal, and overwriting them will probably crash the machine. It is possible to use all of these yourself, but only if the program takes over the entire system (and seizes control from the regular kernal). This means it can no longer use (most) BASIC and kernal routines from ROM.
- it's more convenient and safe to let the compiler allocate these addresses for you and just use symbolic names in the program code.

Prog8 knows what addresses are safe to use in the various ZP handling configurations. It will use the free ZP addresses to place its ZP variables in, until they're all used up. If instructed to output a program that takes over the entire machine, (almost) all of the ZP addresses are suddenly available and will be used.

ZeroPage handling is configurable: There's a global program directive to specify the way the compiler treats the ZP for the program. The default is to be reasonably restrictive to use the part of the ZP that is not used by the C64's kernal routines. It's possible to claim the whole ZP as well (by disabling the operating system or kernal). If you want, it's also possible to be more restricive and stay clear of the addresses used by BASIC routines too. This allows the program to exit cleanly back to a BASIC ready prompt - something that is not possible in the other modes.

IRQs and the ZeroPage

The normal IRQ routine in the C-64's kernal will read and write several addresses in the ZP (such as the system's software jiffy clock which sits in `$a0 - $a2`):

`$a0 - $a2`; `$91`; `$c0`; `$c5`; `$cb`; `$f5 - $f6`

These addresses will *never* be used by the compiler for ZP variables, so variables will not interfere with the IRQ routine and vice versa. This is true for the normal ZP mode but also for the mode where the whole system and ZP have been taken over. So the normal IRQ vector can still run and will be when the program is started!

5.5.2 CPU

Directly Usable Registers

The hardware CPU registers are not directly accessible from regular Prog8 code. If you need to mess with them, you'll have to use inline assembly. Be extra wary of the `X` register because it is used as an evaluation stack pointer and changing its value you will destroy the evaluation stack and likely crash the program.

The status register (P) carry flag and interrupt disable flag can be written via a couple of special builtin functions (`set_carry()`, `clear_carry()`, `set_irqd()`, `clear_irqd()`), and read via the `read_flags()` function.

The 16 'virtual' 16-bit registers that are defined on the Commander X16 machine are not real hardware registers and are just 16 memory-mapped word values that you *can* access directly.

5.5.3 IRQ Handling

Normally, the system's default IRQ handling is not interfered with. You can however install your own IRQ handler (for clean separation, it is advised to define it inside its own block). There are a few library routines available to make setting up C-64 60hz IRQs and Raster IRQs a lot easier (no assembly code required).

For the C64 these routines are:

```
c64.set_irq(uword handler_address, boolean useKernal)
c64.set_rasterirq(uword handler_address, uword rasterline, boolean useKernal)
c64.restore_irq()     ; set everything back to the systems default irq handler
```

And for the Commander X16:

```
cx16.set_irq(uword handler_address, boolean useKernal)          ; vsync irq
cx16.set_rasterirq(uword handler_address, uword rasterline)     ; note: disables kernal␣
↪irq handler! sys.wait() won't work anymore
cx16.restore_irq()     ; set everything back to the systems default irq handler
```

5.6 Technical details

5.6.1 All variables are static in memory

All variables are allocated statically, there is no concept of dynamic heap or stack frames. Essentially all variables are global (but scoped) and can be accessed and modified anywhere, but care should be taken ofcourse to avoid unexpected side effects.

Especially when you're dealing with interrupts or re-entrant routines: don't modify variables that you not own or else you will break stuff.

5.6.2 Software stack for expression evaluation

Prog8 uses a software stack to evaluate complex expressions that it can't calculate in-place or directly into the target variable, register, or memory location.

'software stack' means: seperated and not using the processor's hardware stack.

The software stack is implemented as follows:

- 2 pages of memory are allocated for this, exact locations vary per machine target. For the C-64 they are set at $ce00 and $cf00 (so $ce00-$cfff is reserved). For the Commander X16 they are set at $0400 and $0500 (so $0400-$05ff are reserved).
- these are the high and low bytes of the values on the stack (it's a 'split 16 bit word stack')
- for byte values just the lsb page is used, for word values both pages
- float values (5 bytes) are chopped up into 2 words and 1 byte on this stack.
- the X register is permanently allocated to be the stack pointer in the software stack.
- you can use the X register as long as you're not using the software stack. But you *must* make sure it is saved and restored after the code that modifies it, otherwise the evaluation stack gets corrupted.

5.6.3 Subroutine Calling Convention

Calling a subroutine requires three steps:

1. preparing the arguments (if any) and passing them to the routine
2. calling the routine
3. preparig the return value (if any) and returning that from the call.

Calling the routine is just a simple JSR instruction, but the other two work like this:

`asmsub` routines

These are usually declarations of kernal (ROM) routines or low-level assembly only routines, that have their arguments solely passed into specific registers. Sometimes even via a processor status flag such as the Carry flag. Return values also via designated registers. The processor status flag is preserved on returning so you can immediately act on that for instance via a special branch instruction such as `if_z` or `if_cs` etc.

regular subroutines

- subroutine parameters are just variables scoped to the subroutine.
- the arguments passed in a call are evaluated (using the eval-stack if needed) and then copied into those variables. Using variables for this sometimes can seem inefficient but it's required to allow subroutines to work locally with their parameters and allow them to modify them as required, without changing the variables used in the call's arguments. If you want to get rid of this overhead you'll have to make an `asmsub` routine in assembly instead.
- the order of evaluation of subroutine call arguments *is unspecified* and should not be relied upon.
- the return value is passed back to the caller via cpu register(s): Byte values will be put in `A` . Word values will be put in `A` + `Y` register pair. Float values will be put in the `FAC1` float 'register' (Basic allocated this somewhere in ram).

Calls to builtin functions are treated in a special way: Generally if they have a single argument it's passed in a register or register pair. Multiple arguments are passed like a normal subroutine, into variables. Some builtin functions have a fully custom implementation.

The compiler will warn about routines that are called and that return a value, if you're not doing something with that returnvalue. This can be on purpuse if you're simply not interested in it. Use the `void` keyword in front of the subroutine call to get rid of the warning in that case.

5.6.4 The 6502 CPU's X-register: off-limits

Prog8 uses the cpu's X-register as a pointer in its internal expression evaluation stack. When only writing code in Prog8, this is taken care of behind the scenes for you by the compiler. However when you are including or linking with assembly routines or kernal/ROM calls that *do* use the X register (either clobbering it internally, or using it as a parameter, or return value register), those calls will destroy Prog8's stack pointer and this will result in invalid calculations.

You should avoid using the X register in your assembly code, or take preparations. If you make sure that the value of the X register is preserved before calling a routine that uses it, and restored when the routine is done, you'll be ok.

Routines that return a value in the X register can be called from Prog8 but the return value is inaccessible unless you write a short piece of inline assembly code to deal with it yourself, such as:

```
ubyte returnvalue

%asm {{
    stx  P8ZP_SCRATCH_REG       ; use 'phx/plx' if using 65c02 cpu
    ldx  #10
    jsr  routine_using_x
    stx  returnvalue
    ldx  P8ZP_SCRATCH_REG
}}
; now use 'returnvalue' variable
```

Prog8 also provides some help to deal with this:

- you should use a `clobbers(X)` specification for asmsub routines that modify the X register; the compiler will preserve it for you automatically when such a routine is called
- the `rsavex()` and `rrestorex()` builtin functions can preserve and restore the X register
- the `rsave()` and `rrestore()` builtin functions can preserve and restore *all* registers (but this is very slow and overkill if you only need to save X)

5.7 Porting Guide

Here is a guide for porting Prog8 to other compilation targets. Answers to the questions below are used to configure the new target and supporting libraries.

5.7.1 CPU

1. 6502 or 65C02? (or strictly compatible with one of these)
2. can the **64tass** cross assembler create programs for the system? (if not, bad luck atm)

5.7.2 Memory Map

Zero page

1. *Absolute requirement:* Provide three times 2 consecutive bytes (i.e. three 16-bit pointers) in the Zero page that are free to use at all times.
2. Provide list of any additional free Zero page locations for a normal running system (basic + kernal enabled)
3. Provide list of any additional free Zero page locations when basic is off, but floating point routines should still work
4. Provide list of any additional free Zero page locations when only the kernal remains enabled

Only the three 16-bit pointers are absolutely required to be able to use prog8 on the system. But more known available Zero page locations mean smaller and faster programs.

RAM, ROM, I/O

1. what part(s) of the address space is RAM? What parts of the RAM can be used by user programs?
2. what is the usual starting memory address of programs?
3. what is the best place to put 2 pages (512 bytes total) of scratch area data in RAM?
4. what part(s) of the address space is ROM?
5. what part(s) of the address space is memory mapped I/O registers?
6. is there a banking system? How does it work (how do you select Ram/Rom banks)? How is the default bank configuration set?

5.7.3 Character encodings

1. if not Petscii or CBM screencodes: provide the primary character encoding table that the system uses (i.e. how is text represented in memory)
2. provide alternate character encodings (if any)
3. what are the system's standard character screen dimensions?
4. is there a screen character matrix directly accessible in Ram? What's it address? Same for color attributes if any.

5.7.4 ROM routines

1. provide a list of the core ROM routines on the system, with names, addresses, and call signatures.

Ideally there are at least some routines to manipulate the screen and get some user input(clear, print text, print numbers, input strings from the keyboard) Routines to initialize the system to a sane state and to do a warm reset are useful too. The more the merrier.

Floating point

Prog8 supports floating point math *if* the target system has floating point math routines in ROM. If the machine has this:

1. what is the binary representation format of the floating point numbers? (how many bytes, how the bits are set up)
2. what are the valid minimum negative and maximum positive floating point values?
3. provide a list of the floating point math routines in ROM: name, address, call signature.

5.7.5 Support libraries

The most important libraries are `syslib` and `textio`. `syslib` *has* to provide several system level functions such as how to initialize the machine to a sane state, and how to warm reset it, etc. `textio` contains the text output and input routines, it's very welcome if they are implemented also for the new target system.

There are several other support libraries that you may want to port (`diskio`, `graphics` to name a few).

Also ofcourse if there are unique things available on the new target system, don't hesitate to provide extensions to the `syslib` or perhaps a new special custom library altogether.

5.8 TODO

5.8.1 For next release

...

5.8.2 Need help with

- c128 target: various machine specific things (free zp locations, how banking works, getting the floating point routines working, ...)
- atari target: more details details about the machine, fixing library routines. I have no clue whatsoever.
- see the *Porting Guide* for details on what information is needed.

5.8.3 Future Things and Ideas

Compiler:

- add McCarthy evaluation to shortcircuit and/or expressions. First do ifs by splitting them up? Then do expressions that compute a value?
- Inliner: also inline function call expressions, and remove it from the StatementOptimizer
- vm: implement remaining sin/cos functions in math.p8
- vm: somehow deal with asmsubs otherwise the vm IR can't fully encode all of prog8
- vm: don't store symbol names in instructions to make optimizing the IR easier? but what about jumps to labels. And it's no longer readable by humans.
- vm: how to remove all unused subroutines? (in the 6502 assembly codegen, we let 64tass solve this for us)
- vm: rather than being able to jump to any 'address' (IPTR), use 'blocks' that have entry and exit points -> even better dead code elimination possible too
- when the vm is stable and *if* its language can get promoted to prog8 IL, the variable allocation should be changed. It's now done before the vm code generation, but the IL should probably not depend on the allocations already performed. So the CodeGen doesn't do VariableAlloc *before* the codegen, but as a last step.
- createAssemblyAndAssemble(): make it possible to actually get rid of the VarDecl nodes by fixing the rest of the code mentioned there. but probably better to rewrite the 6502 codegen on top of the new Ast.
- simplifyConditionalExpression() should not split expression if it still results in stack-based evaluation, but how does it know?
- simplifyConditionalExpression() sometimes introduces needless assignment to r9 tempvar (what scenarios?)
- make it possible to use cpu opcodes such as 'nop' as variable names by prefixing all asm vars with something such as `p8v_`? Or not worth it (most 3 letter opcodes as variables are nonsensical anyway) then we can get rid of the instruction lists in the machinedefinitions as well?
- [problematic due to using 64tass:] add a compiler option to not remove unused subroutines. this allows for building library programs. But this won't work with 64tass's .proc ... Perhaps replace all uses of .proc/.pend by .block/.bend will fix that? (but we lose the optimizing aspect of the assembler where it strips out unused code. There's not really a dynamic switch possible as all assembly lib code is static and uses one or the other)
- Zig-like try-based error handling where the V flag could indicate error condition? and/or BRK to jump into monitor on failure? (has to set BRK vector for that)

- add special (u)word array type (or modifier?) that puts the array into memory as 2 separate byte-arrays 1 for LSB 1 for MSB -> allows for word arrays of length 256

Libraries:

- fix the problems in c128 target, and flesh out its libraries.
- fix the problems in atari target, and flesh out its libraries.
- c64: make the graphics.BITMAP_ADDRESS configurable (VIC banking)
- optimize several inner loops in gfx2 even further?
- add modes 2 and 3 to gfx2 (lowres 4 color and 16 color)?
- add a flood fill routine to gfx2?
- diskio: use cx16 MACPTR() in f_read() to load stuff faster? (see its use in X16edit to fast load blocks) note that it might fail on non sdcard files so have to make graceful degradation

Expressions:

- pipe operator: (targets other than 'Virtual'): allow non-unary function calls in the pipe that specify the other argument(s) in the calls.
- rethink the whole "isAugmentable" business. Because the way this is determined, should always also be exactly mirrorred in the AugmentableAssignmentAsmGen or you'll get a crash at code gen time. note: new ast PtAssignment already has no knowledge about this anymore.
- can we get rid of pieces of asmgen.AssignmentAsmGen by just reusing the AugmentableAssignment ? generated code should not suffer
- rewrite expression tree evaluation suchthat it doesn't use an eval stack but flatten the tree into linear code that uses a fixed number of predetermined value 'variables'? "Three address code" was mentioned. https://en.wikipedia.org/wiki/Three-address_code these variables have to be unique for each subroutine because they could otherwise be interfered with from irq routines etc.
- this removes the need for the BinExprSplitter? (which is problematic and very limited now) and perhaps as well the assignment splitting in BeforeAsmAstChanger too

Optimizations:

- various optimizers skip stuff if compTarget.name==VMTarget.NAME. Once (if?) 6502-codegen is no longer done from the old CompilerAst, those checks should probably be removed, or be made permanent
- VariableAllocator: can we think of a smarter strategy for allocating variables into zeropage, rather than first-come-first-served
- **translateUnaryFunctioncall() in BuiltinFunctionsAsmGen: should be able to assign parameters to a builtin function direc**
  compare `aa = startvalue(1) |> sin8u() |> cos8u() |> sin8u() |> cos8u()` versus: `aa = cos8u(sin8u(cos8u(sin8u(startvalue(1)))))` the second one contains no sta cx16.r9L in between.
- AssignmentAsmGen.assignExpression() -> better code gen for assigning boolean comparison expressions
- when a for loop's loopvariable isn't referenced in the body, and the iterations are known, replace the loop by a repeatloop but we have no efficient way right now to see if the body references a variable.
- introduce byte-index operator to avoid index multiplications in loops over arrays? see github issue #4

CHAPTER

SIX

INDEX

- genindex

INDEX

W