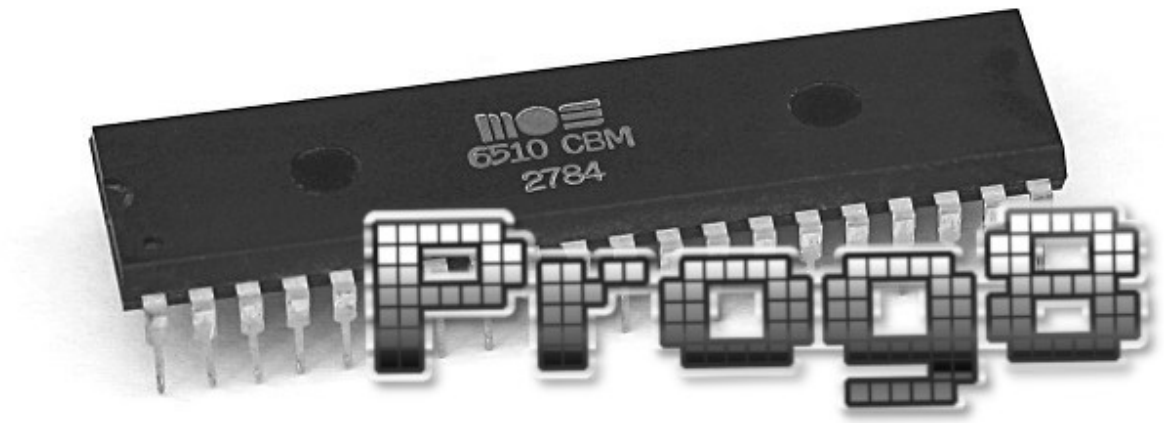
Prog8 Documentation

Release 10.3

Irmen de Jong

Apr 18, 2024

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:

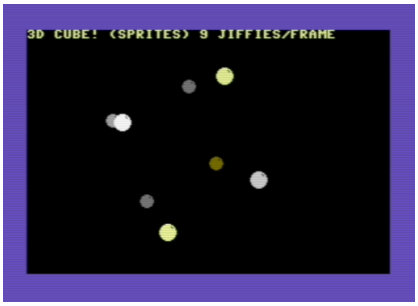
- Commander X16
- Commodore 64
- Commodore 128 (limited support)
- Commodore PET (limited support)
- Atari 800 XL (limited support)

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

1.1 Software License

1.2 Want to buy me a coffee or a pizza perhaps?

This project was created over the last couple of years by dedicating thousands of hours of my free time to it, to make it the best I possibly can. If you like Prog8, and think it's worth a nice cup of hot coffee or a delicious pizza, you can help me out a little bit over at https://ko-fi.com/irmen .


3D CUBE! (SPRITES) 9 JIFFIES/FRAME


READY.
THE BALLOON SPRITES ARE ANIMATED VIA
THE RASTER IRQ


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


Commander X16
200
00000


Tools
draW
Line
Rectangle
Box
Circle
Disc
Erase
Fill
Zoom
Palette
Commands
Undo
Clear
Load
Save
Drive: 8
Quit
Info
Tool hotkeys are underlined.
Hold CTRL to show coordinates.
TAB to toggle menu.


A B C D E F G H
It is your turn, white.

CHAPTER

TWO

FEATURES

- it is a cross-compiler running on modern machines (Linux, MacOS, Windows, ...)
- the compiled programs run very fast, because compilation to highly efficient native machine code.
- Provides a convenient and fast edit/compile/run cycle by being able to directly launch the compiled program in an emulator and provide debugging information to this emulator.
- the language looks like a mix of Python and C so should be quite easy to learn
- Modular programming, scoping via modules, code blocks, and subroutines. No need for forward declarations.
- 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 of various types
- Complex nested expressions are possible
- Variables are all allocated statically, no memory allocator overhead
- Conditional branches for status flags that map 1:1 to processor branch instructions for optimal efficiency
- `when` statement to avoid if-else chains
- `in` expression for concise and efficient multi-value/containment test
- Several powerful built-in functions, such as `lsb`, `msb`, `min`, `max`, `rol`, `ror`, `sort` and `reverse`
- Variable data types include signed and unsigned bytes and words, arrays, strings.
- Various powerful built-in libraries to do I/O, number conversions, graphics and more
- Floating point math is supported on select compiler targets.
- Easy and highly efficient integration with external subroutines and ROM routines on the target systems.
- 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.
- Encode strings and characters into petscii or screencodes or even other encodings, as desired (C64/Cx16)
- Identifiers can contain Unicode Letters, so `knäckebröd`, , and are all valid identifiers.
- Advanced code optimizations to make the resulting program smaller and faster
- Programs can be restarted after exiting (i.e. run them multiple times without having to reload everything), due to automatic variable (re)initializations.
- Supports the sixteen ‘virtual’ 16-bit registers R0 to R15 as defined on the Commander X16. These are also available on the other compilation targets!

- On the Commander X16: Support for low level system features such as Vera Fx, which includes 16x16 bits multiplication in hardware and fast memory copy and fill.
- Many library routines are available across compiler targets. This means that as long as you only use standard Kernal and core prog8 library routines, it is sometimes 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
%zeropage basicsafe

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

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

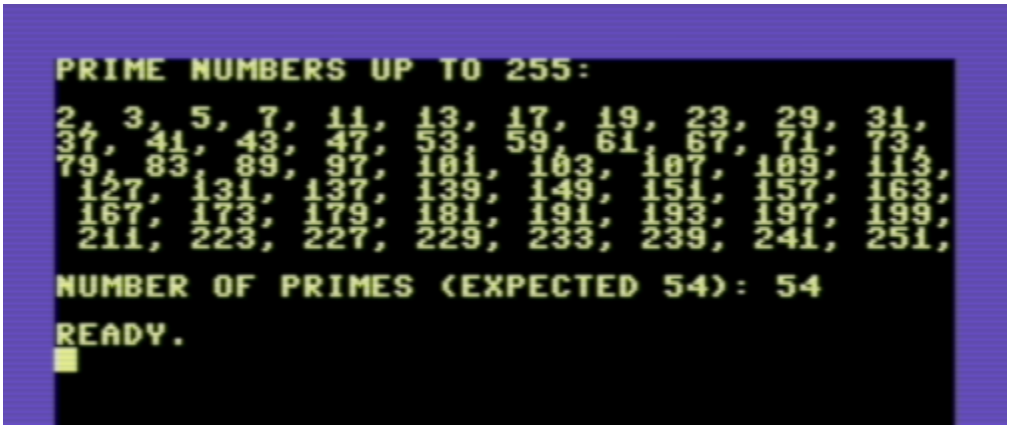
```
%import textio
%zeropage basicsafe

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

    sub start() {
        sys.memset(sieve, 256, 0)   ; clear the sieve
        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] {
            candidate_prime++
```

(continues on next page)

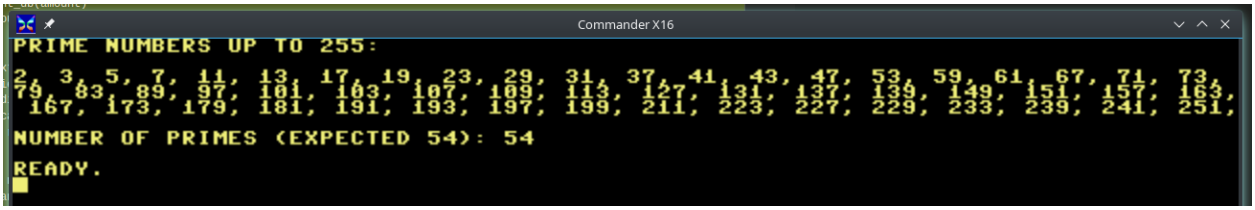
(continued from previous page)

```
            if candidate_prime==0
                return 0        ; we wrapped; no more primes
        }

        ; 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 C64 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 program somewhere on your shell's search path. It's easy to compile yourself, but a recent precompiled .exe (only for Windows) can be obtained from the files section in the official project on sourceforge. *You need at least version 1.58.0 of this assembler.* If you are on Linux, there's probably a "64tass" package in the repositories, but check if it is a recent enough version.

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, get one of the versions of another vendor. Even Microsoft provides their own version. Other OpenJDK builds can be found at Adoptium . For MacOS you can also use the Homebrew system to install a recent version of OpenJDK.

Finally: an **emulator** (or a real machine of course) to test and run your programs on. For the PET, C64 and C128 targets, the compiler assumes the presence of the VICE emulator. If you're targeting the Commander X16 instead, download a recent emulator version for the CommanderX16, such as x16emu (preferred, this is the official emulator. If required, source code is here. There is also Box16 which has powerful debugging features. For the Atari target, it assumes the "atari800" or "altirra" emulator. If multiple options are listed above, you can select which one you want to launch using the `-emu` or `-emu2` command line options.

Syntax highlighting: for a few different editors, syntax highlighting definition files are provided. Look in the syntax-files directory in the github repository to find them.

5.1 Compiling 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 an official release 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.jar" to see how you can use it (use the correct name and version of the jar file you've downloaded).

Or, install via a Package Manager:

Currently, it's only available on AUR for Arch Linux and compatible systems. The package is called "prog8".

This package, alongside the compiler itself, also globally installs syntax highlighting for `vim` and `nano`. In order to run compiler, you can type either `p8compile` or `prog8c`. The usage of those commands is exatcly the same as with the `java -jar` method.

In case you prefer to install AUR packages in a traditional manner, make sure to install "tass64" package before installing prog8, as makepkg itself doesn't fetch AUR dependencies.

Or, download a bleeding edge development version from Github:

1. find the latest CI build on the actions page on Github
2. download the zipped jar artifact from that build, and unzip it.
3. run the compiler with "java -jar prog8compiler.jar" (use the correct name and version of the jar file you've downloaded).

Or, use the Gradle build system to build it yourself from source:

The Gradle build system is used to build the compiler. The most interesting gradle commands to run are probably the ones listed below. (Note: if you have a recent gradle installed on your system already, you can probably replace the `./gradlew` wrapper commands with just the regular `gradle` command.)

`./gradlew build`
: Builds the compiler code and runs all available checks and unit-tests. Also automatically runs the installDist and installShadowDist tasks. Read below at those tasks for where the resulting compiler jar file gets written.

`./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` task should suffice and after succesful completion, you can start the compiler with:

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

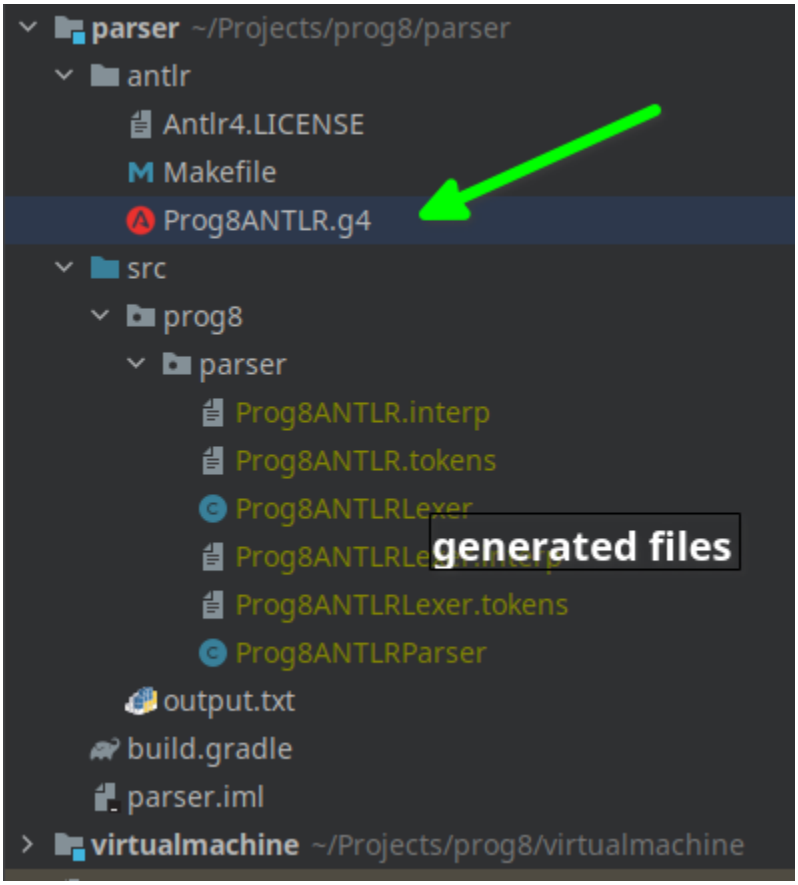
(You should probably make an alias or link...)

Hint: Development and testing is done on Linux using the IntelliJ IDEA IDE, but the actual prog8 compiler should run on all operating systems that provide a java runtime (version 11 or newer). If you do have trouble building or running the compiler on your operating system, please let me know!

To successfully build and debug in IDEA, you have to manually generate the Antlr-parser classes first. The easiest way to do this is the following:

1. make sure you have the Antlr4 plugin installed in IDEA
2. right click the grammar file Prog8ANTLR.g4 in the parser project, and choose "Generate Antlr Recognizer" from the menu.
3. rebuild the full project.

Alternatively you can also use the Makefile in the antlr directory to generate the parser, but for development the Antlr4 plugin provides several extremely handy features so you'll probably want to have it installed anyway.



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.jar -target cx16 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. The -target option is required, in this case we tell it to compile a program for the Commander X16)

By default, assembly code is generated and written to `sourcefile.asm`. It is then (automatically) fed to the 64tass assembler tool that creates 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. This option is required. `c64` = Commodore 64, `c128` = Commodore 128, `cx16` = Commander X16, `pet32` - Commodore PET model 4032, `atari` = Atari 800 XL, `virtual` = builtin virtual machine.

-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 (if it supports them)

-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.

-nostrictbool
Relax the strict boolean type checks: bytes and booleans can be interchanged again without explicit type casts. *This option will likely disappear in a future prog8 version, so you may want to prepare for that in your code!*

-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.

-warnshadow
Tells the assembler to issue warning messages about symbol shadowing. These *can* be problematic, but usually aren't because prog8 has different scoping rules than the assembler has. You may want to watch out for shadowing of builtin names though. Especially 'a', 'x' and 'y' as those are the cpu register names and if you shadow those, the assembler might interpret certain instructions differently and produce unexpected opcodes (like LDA X getting turned into TXA, or not, depending on the symbol 'x' being defined in your own assembly code or not)

-quietasm
Don't print assembler output results.

-asmlist
Generate an assembler listing file as well.

-check
Quickly check the program for errors. No output will be produced.

-breakinstr <instruction>
Also output the specified CPU instruction for a `%breakpoint`, as well as the entry in the vice monitor list file. This can be useful on emulators/systems that don't parse the breakpoint information in the list file, such as the X16Emu emulator for the Commander X16. Useful instructions to consider are `brk` and `stp`. For example for the Commander X16 emulator, `stp` is useful because it can actually tyrigger a breakpoint halt in the debugger when this is enabled by running the emulator with -debug.

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

-printast1
Prints the "compiler AST" (the internal representation of the program) after all processing steps.

-printast2
Prints the "intermediate AST" which is the reduced representation of the program. This is what is used in the code generators, to generate the executable code from.

-dumpvars
print a dump of the variables in the program

-dumpsymbols
print a dump of the variable declarations and subroutine signatures

-sourcelines
Also include the original prog8 source code lines as comments in the generated assembly code file, mixed in between the actual generated assembly code. This can be useful for debugging purposes to see what assembly was generated for what prog8 source code.

-splitarrays
Treat all word arrays as tagged with @split so they are all lsb/msb split into memory. This removes the need to add @split yourself but some programs may fail to compile with this option as not all array operations are implemented yet on split arrays.

-vm
load and run a p8-virt or p8-ir listing in the internal VirtualMachine instead of compiling a prog8 program file..

-D SYMBOLNAME=VALUE
Add this user-defined symbol directly to the beginning of the generated assembly file. Can be repeated to define multiple symbols.

-varshigh <rambank>
Places uninitialized non-zeropage variables in a separate memory area, instead of inside the program itself. This increases the amount of system ram available for program code. The size of the increase depends on the program but can be several hundreds of bytes or more. The location of the memory area for these variables depends on the compilation target machine:

c64: $C000 - $CFFF ; 4 kB, and the specified rambank number is ignored

cx16: $A000 - $BFFF ; 8 kB in the specified HiRAM bank (note: no auto bank switching is done, you must make sure yourself that this HiRAM bank is active when accessing these variables!)

If you use this option, you can no longer use the part of the above memory area that is alotted to the variables, for your own purposes. The output of the 64tass assembler step at the end of compilation shows precise details of

where and how much memory is used by the variables (it's called 'BSS' section or Gap at the address mentioned above). Assembling the program will fail if there are too many variables to fit in a single high ram bank.

`-varsgolden`
: Like `-varshigh`, but places the variables in the $0400-$07FF "golden ram" area instead. Because this is in normal system memory, there are no bank switching issues. This mode is only available on the Commander X16.

`-slabshigh`
: put memory() slabs in high memory area instead of at the end of the program. On the cx16 target the value specifies the HiRAM bank to use, on other systems this value is ignored.

`-slabsgolden`
: put memory() slabs in 'golden ram' memory area instead of at the end of the program. On the cx16 target this is $0400-07ff. This is unavailable on other systems.

`-bytes2float <bytes>`
: convert a comma separated list of bytes from the target system to a float value. NOTE: you need to supply a target option too, and also still have to supply a dummy module file name as well!

`-float2bytes <number>`
: convert floating point number to a list of bytes for the target system. NOTE: you need to supply a target option too, and also still have to supply a dummy module file name as well!

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. This can also be a lo-fi way to use different source files for different compilation targets if you wish. Which is useful as currently the compiler doesn't have conditional compilation like #ifdef/#endif in C.

5.1.5 Debugging (with VICE or Box16)

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 and Box16 emulators. 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.

Box16 is the alternative emulator for the Commander X16 and it also includes debugging facilities that support these symbol and breakpoint lists.

5.1.6 Troubleshooting

Compiler doesn't run, complains about "UnsupportedClassVersionError"

You need to install and use JDK version 11 or newer to run the prog8 compiler. Check this with "java -version". See *Required additional tools*.

The computer just resets (at the end of the program)

In the default compiler configuration, it is not safely possible to return back to the BASIC prompt when your program exits. The only reliable thing to do is to reboot the system. This is due to the fact that in this mode, prog8 will overwrite important BASIC and Kernal variables in zero page memory. To avoid the reset from happening, use an empty `repeat` loop at the end of your program to keep it from exiting. Alternatively, if you want your program to exit cleanly back to the BASIC prompt, you have to use `%zeropage basicsafe`, see *Directives*. The reason this is not the default is that it is very beneficial to have more zeropage space available to the program, and programs that have to return cleanly to the BASIC prompt are considered to be the exception.

Odd text and screen colors at start

Prog8 will reset the screen mode and colors to a uniform well-known state. If you don't like the default text and screen colors, you can simply change them yourself to whatever you want at the start of your program. It depends on the computer system how you do this but there are some routines in the textio module to help you with this. Alternatively you can choose to disable this re-initialization altogether using `%option no_sysinit`, see *Directives*.

Floats error

Are you 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`.

Gradle error when building the compiler yourself

If you get a gradle build error containing the line "No matching toolchains found for requested specification" somewhere, it means that the Gradle build tool can't locate the correct version of the JDK to use. The file "gradle.properties" contains a line like this: `javaVersion=11`. You can do one of two things to fix the build error:

- install a JDK with that version,
- or change the version number to match the JDK version that *is* installed on your system (must be >= 11)

Strange assembler errors

If the compilation of your program fails in the assembly step, please check that you have the required version of the 64tass assembler installed. See *Required additional tools*. Also make sure that inside hand-written inlined assembly, you don't use symbols named just a single letter (especially 'a', 'x' and 'y'). Sometimes these are interpreted as the CPU register of that name. To avoid such confusions, always use 2 or more letters for symbols in your assembly code.

'shadowing' warnings form the assembler

Avoid using 'a', 'x' or 'y' as symbols in your inlined assembly code. Also avoid using 64tass' built-in function or type names as symbols in your inlined assembly code. The 64tass manual contains a list of those.

5.1.7 Community

Most of the development on Prog8 and the use of it is currently centered around the Commander X16 retro computer. Their Discord server contains a small channel dedicated to Prog8. Other than that, use the issue tracker on github.

5.1.8 Examples

A bunch of example programs can be found in the 'examples' directory of the source tree. There are cross-platform examples that can be compiled for various systems unaltered, and there are also examples specific to certain computers (C64, X16, etcetera). So for instance, to compile and run the Commodore 64 rasterbars example program, use this command:

```
$ java -jar prog8compiler.jar -target c64 -emu examples/c64/rasterbars.p8
```

or:

```
$ /path/to/p8compile -target c64 -emu examples/c64/rasterbars.p8
```

5.2 What makes a Prog8 program

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*. It should be saved in UTF-8 encoding.

Comments
: Everything on the line 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. There's also a block-comment: everything surrounded with `/*` and `*/` is ignored and this can span multiple lines. This block

comment is experimental for now: it may change or even be removed again in a future compiler version. The recommended way to comment out a bunch of lines remains to just bulk comment them individually with `;`.

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 also possible to define subroutines within other subroutines. Nested subroutines can access the variables from outer scopes easily, which removes the need and overhead to pass everything via parameters all the time. Subroutines do not have to be declared in the source code 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), everything in a module file goes in here;
- block;
- subroutine, can be nested in another subroutine.

Even though 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. Some more details about how to deal with scopes and names is discussed below.

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.

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

Every symbol is 'public' and can be accessed from anywhere else, when 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`. Unlike most other programming langues, as soon as a name is scoped, Prog8 treats it as a name starting in the *global* namespace. Relative name lookup is only performed for *non-scoped* names.

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).

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.

Subroutines create a new scope. All variables inside a subroutine are hoisted up to the scope of the subroutine they are declared in. Note that you can define **nested subroutines** in Prog8, and such a nested subroutine has its own scope! This also means that you have to use a fully qualified name to access a variable from a nested subroutine:

```
main {
    sub start() {
        sub nested() {
            ubyte counter
            ...
        }
        ...
        txt.print_ub(counter)                       ; Error: undefined symbol
        txt.print_ub(main.start.nested.counter)     ; OK
    }
}
```

Important: Emphasizing this once more: 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.

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 *Blocks, Scopes, and accessing Symbols*. 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)
false                ; boolean false
-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 zeropage (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. There is no octal notation. You can use underscores to group digits to make long numbers more readable. 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.

Attention: Doing math on signed integers can result in code that is a lot larger and slower than when using unsigned integers. Make sure you really need the signed numbers, otherwise stick to unsigned integers for efficiency.

Booleans

Booleans are a distinct type in Prog8 and can have only the values `true` or `false`. It can be casted to and from other integer types though where a nonzero integer is considered to be true, and zero is false. Logical expressions, comparisons and some other code tends to compile more efficiently if you explicitly use `bool` types instead of 0/1 integers. The in-memory representation of a boolean value is just a byte containing 0 or 1.

If you find that you need a whole bunch of boolean variables or perhaps even an array of them, consider using integer bit mask variable + bitwise operators instead. This saves a lot of memory and may be faster as well.

Floating point numbers

You can use underscores to group digits to make long numbers more readable.

Floats are stored in the 5-byte 'MFLPT' format that is used on CBM machines. Floating point support is available on the c64 and cx16 (and virtual) compiler targets. On the c64 and cx16, the rom routines are used for floating point operations, so on both systems the correct rom banks have to be banked in to make this work. Although the C128 shares the same floating point format, Prog8 currently doesn't support using floating point on that system (because the c128 fp routines require the fp variables to be in another ram bank than the program, something Prog8 doesn't do).

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**)

Arrays

Array types are also supported. They can be formed from a list of booleans, 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
ubyte[99] array = 255           ; initialize array with 99 times 255 [255, 255, 255,␣
↪255, ...]
```

(continues on next page)

(continued from previous page)

```
byte[] array = 100 to 199          ; initialize array with [100, 101, ..., 198, 199]
str[] names = ["ally", "pete"]     ; array of string pointers/addresses (equivalent to␣
↪array of uwords)
uword[] others = [names, array]    ; array of pointers/addresses to other arrays
bool[2] flags = [true, false]      ; array of two boolean values  (take up 1 byte each,␣
↪like a byte array)

value = array[3]            ; the fourth value in the array (index is 0-based)
char = string[4]            ; the fifth character (=byte) in the string
char = string[-2]           ; the second-to-last character in the string (Python-style␣
↪indexing from the end)
```

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 (256 if it's a split array - see below), and float arrays <= 51 elements.

You can write out an array initializer list over several lines if you want to improve readability.

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. In contrast to a real array variable, the index value can be the size of a word. Unlike array variables, you cannot use a negative index to count from the end, because the size of the array is unknown. See also *Direct access to memory locations ('peek' and 'poke')*

LSB/MSB split word arrays: For (u)word arrays, you can make the compiler layout the array in memory as two separate arrays, one with the LSBs and one with the MSBs of the word values. This is more efficient when storing and reading words from the array (the index can be used twice). Add the `@split` tag to the variable declaration to do this. In the assembly code, the array will be generated as two byte arrays namely `name_lsb` and `name_msb`. Note that the maximum length of a split word array is 256! (regular word arrays are limited to 128 elements).

Caution: Not all array operations are supported yet on "split word arrays". The compiler may give an unpleasant error or crash when you hit such a case in your code. If this happens simply revert to a regular word array and please report the issue, so that more support can be added in the future where it is needed.

Strings

Strings are a sequence of characters enclosed in double 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 characters. 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 character 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) **Caution:** This checks *all* elements in the string with the length as it was initially declared. Even when a string was changed and is terminated early with a 0-byte early, the containment check with `in` will still look at all character positions in the initial string. Consider using `string.find` followed by `if_cs` (for instance) to do a "safer" search for a character in such strings (one that stops at the first 0 byte)

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.

Hint: You can declare parameters and return values of subroutines as `str`, but in this case that is equivalent to declaring them as `uword` (because in this case, the address of the string is passed as argument or returned as value).

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' cannot be changed; it has become a compile-time constant value instead. 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 memory is allocated for the constant itself). Onlythe simple numeric types (byte, word, float) can be defined as a constant. If something is defined as a constant, very efficient code can usually be generated from it. Variables on the other hand can't be optimized as much, need memory, and more code to manipulate them. Note that a subset of the library routines in the `math`, `string` and `floats` modules are recognised in compile time expressions. For example, the compiler knows what `math.sin8u(12)` is and replaces it with the computed result.

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 variable 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 address $d020-$d021
```

Direct access to memory locations ('peek' and 'poke')

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

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 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 reinterpretation of the given value as being of the other type, sometimes an actual value conversion is done to convert it into the other type. Try to avoid those 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. If the from value is already outside of the loop range, the whole for loop is skipped.

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). A repeat loop will result in the most efficient code generated so use this if possible.

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, immediately continue into the next cycle of the loop with the `continue` statement. (These are just shorthands for a goto + a label)

The *unroll* loop is not really a loop, but looks like one. It actually duplicates the statements in its block on the spot by the given number of times. It's meant to "unroll loops" - trade memory for speed by avoiding the actual repeat loop counting code. Only simple statements are allowed to be inside an unroll loop (assignments, function calls etc.).

Attention: The value of the loop variable after executing the loop *is undefined* - you cannot rely on it to be the last value in the range for instance! The value of the variable should only be used inside the for loop body. (this is an optimization issue to avoid having to deal with mostly useless post-loop logic to adjust the loop variable's value)

5.2.6 Conditional Execution

if statements

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

```
if xx==5 {
    yy = 99
    zz = 42
} else {
    aa = 3
    bb = 9
}

if xx==5
    yy = 42
else if xx==6
    yy = 43
else
    yy = 44

if aa>4 goto some_label

if xx==3  yy = 4

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

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 function 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).

It is possible to "chain" assignments: `x = y = z = 42`, this is just a shorthand for the three individual assignments with the same value 42.

Only for certain subroutines that return multiple values it is possible to write a "multi assign" statement with comma separated assignment targets, that assigns those multiple values to different targets in one statement. Details can be found here: *Multiple return values*.

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 that can be used in expressions (see *Built-in Functions*). You can also reference identifiers 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). When using variables of the type `bool`, logical expressions will compile more efficiently than when you're using regular integer type operands (because these have to be converted to 0 or 1 every time) Prog8 applies short-circuit aka McCarthy evaluation for `and` and `or` on boolean expressions.

You can use parentheses to group parts of an expression 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.

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)
: Returns the absolute value of a number (integer or floating point).

min (x, y)
: Returns the smallest of x and y. Supported for integer types only, for floats use `floats.minf()` instead.

max (x, y)
: Returns the largest of x and y. Supported for integer types only, for floats use `floats.maxf()` instead.

clamp (value, minimum, maximum)
: Returns the value restricted to the given minimum and maximum. Supported for integer types only, for floats use `floats.clampf()` instead.

sgn (x)
Get the sign of the value (integer or floating point). The result is a byte: -1, 0 or 1 (negative, zero, positive).

sqrt (w)
Returns the square root of the number. Supports unsigned integer (result is ubyte) and floating point numbers. To do the reverse - squaring a number - just write `x*x`.

divmod (dividend, divisor, quotient, remainder)
Performs division only once and returns both quotient and remainder in a single call, where using '/' and '%' separately would perform the division operation twice. All values are ubytes or all are uwords. The last two arguments must be variables to receive the quotient and remainder results, respectively.

Array operations

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

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

peekf (address)
reads the float value at the given address in memory. On CBM machines, this reads 5 bytes.

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.

pokef (address, value)
writes the float value at the given address in memory. On CBM machines, this writes 5 bytes.

pokemon (address, value)
Like poke(), but also returns the previous value in the given address. Also doesn't have anything to do with a certain video game.

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))`

setlsb (x, value)
Sets the least significant byte of word variable x to a new value. Leaves the MSB untouched.

setmsb (x, value)
Sets the most significant byte of word variable x to a new value. Leaves the LSB untouched.

sizeof (name) ; sizeof (number)
Number of bytes that the object 'name', or the number 'number' 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 C64, where a float is 5 bytes). Note: usually you will be interested in the number of elements in an array, use len() for that.

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. The block is uninitialized memory, it is *not* set to zero! 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.

call (address) -> uword
Calls a subroutine given by its memory address. You cannot pass arguments directly, although it is ofcourse possible to do this via the global `cx16.r0...` registers for example. It is assumed the subroutine returns a word value (in AY), if it does not, just add void to the call to ignore the result value. This function effectively creates an "indirect JSR" if you use it on a `uword` pointer variable. But because it doesn't handle bank switching etcetera by itself, it is a lot faster than `callfar`. And it works on other systems than just the Commander X16.

callfar (bank, address, argumentword) -> uword ; NOTE: specific to cx16 target for now
Calls an assembly routine in another bank on the Commander X16 (using its `JSRFAR` routine) Be aware that ram OR rom bank may be changed depending on the address it jumps to! The argumentword will be loaded into the A+Y registers before calling the routine. The uword value that the routine returns in the A+Y registers, will be returned. NOTE: this routine is very inefficient, so don't use it to call often. Set the bank yourself or even write a custom tailored trampoline routine if you need to. Or use `call` if you can.

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 subroutine call 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
Saves all registers including status (or only X) on the stack Note: the 16 bit 'virtual' registers of the Commander X16 are *not* saved, but you can use `cx16.save_virtual_registers()` for that.

rrestore
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, but you can use `cx16.restore_virtual_registers()` for that.

5.2.11 Library routines

There are many routines available in the compiler libraries. Some are used internally by the compiler as well.

The most important ones can be found in the *Library modules* chapter.

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, saved in UTF-8 encoding, which 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 on a line 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. Everything surrounded with `/*` and `*/`, this can span multiple lines, is a block-comment and is ignored. This block comment is experimental for now: it may change or even be removed again in a future compiler version. Examples:

```
counter = 42    ; set the initial value to 42
; next is the code that...
/* this
is
all
ignored */
```

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, with 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.

Note: `kernalsafe` and `full` on the C64 leave enough room in the zeropage to reallocate the 16 virtual registers cx16.r0…cx16.r15 from the Commander X16 into the zeropage as well (but not on the same locations). They are relocated automatically by the compiler. The other options need those locations for other things so those virtual registers have to be put into memory elsewhere (outside of the zeropage). Trying to use them as zeropage variables or pointers etc. will be a lot slower in those cases! On the Commander X16 the registers are always in zeropage. On other targets, for now, they are always outside of the zeropage.

%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.

%zpallowed <fromaddress>,<toaddress>

Level: module. Global setting, can occur multiple times. It allows you to designate a part of the zeropage that the compiler is allowed to use (if other options don't prevent usage).

%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 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). Warning: if you use this to align array variables in the block, these have to be initialized with a value to make them stay in the block and get aligned properly. Otherwise they'll end up at a random spot in the BSS section and the alignment doesn't apply there.
- `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). Warning: if you use this to align array variables in the block, these have to be initialized with a value to make them stay in the block and get aligned properly. Otherwise they'll end up at a random spot in the BSS section and the alignment doesn't apply there.
- `merge` (in a block) will merge this block's contents into an already existing block with the same name. Useful in library scenarios. Can result in a bunch of unused symbol warnings, this depends on the import order.
- `splitarrays` (block or module) makes all word-arrays in this scope lsb/msb split arrays (as if they all have the @split tag). See Arrays.
- `no_symbol_prefixing` (block or module) makes the compiler *not* use symbol-prefixing when translating prog8 code into assembly. Only use this if you know what you're doing because it could result in invalid assembly code being generated. This option can be useful when writing library modules that you don't want to be exposing prefixed assembly symbols.
- `ignore_unused` (block or module) suppress warnings about unused variables and subroutines. Instead, these will be silently stripped. This option is useful in library modules that contain many more routines beside the ones that you actually use.
- `verafxmuls` (block, cx16 target only) uses Vera FX hardware word multiplication on the CommanderX16 for all word multiplications in this block. Warning: this may interfere with IRQs and other Vera operations, so use this only when you know what you're doing. It's safer to explicitly use `verafx.muls()`.

%encoding <encodingname>

Overrides, in the module file it occurs in, the default text encoding to use for strings and characters that have no explicit encoding prefix. You can use one of the recognised encoding names, see *String*.

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

Level: not at module scope. This directive can only be used inside a block. The assembler itself will include the file as binary bytes at this point, prog8 will not process this at all. This means that the filename must be spelled exactly as it appears on your computer's file system. Note that this filename may differ in case compared to when you chose to load the file from disk from within the program code itself (for example on the C64 and X16 there's the PETSCII encoding difference). The file is located relative to the current working directory! The optional offset and length can be used to select a particular piece of the file. To reference the contents of the included binary data, you can put a label in your prog8 code just before the %asmbinary. To find out where the included binary data ends, add another label directly after it. 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 of course 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.

> **Caution:** Avoid using single-letter symbols in included assembly code, as they could be confused with CPU registers. Also, note that all prog8 symbols are prefixed in assembly code, see *Symbol prefixing in generated Assembly code*.

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"
end_of_included_bin:

    }
}
```

%breakpoint

Level: not at module scope. Defines a debugging breakpoint at this location. See *Debugging (with VICE or Box16)*

%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. 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.

If you use the correct scoping rules you can access symbols from the prog8 program from inside the assembly code. Sometimes you'll have to declare a variable in prog8 with *@shared* if it is only used in such assembly code.

Note: 64tass syntax is required for the assembly code. As such, mnemonics need to be written in lowercase.

Caution: Avoid using single-letter symbols in included assembly code, as they could be confused with CPU registers. Also, note that all prog8 symbols are prefixed in assembly code, see *Symbol prefixing in generated Assembly code*.

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. Note that any Unicode Letter symbol is accepted as a letter! Examples of valid identifiers:

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

Scoped names

Sometimes called "qualified names" or "dotted names", a scoped name is a sequence of identifiers separated by a dot. They are used to reference symbols in other scopes. Note that unlike many other programming languages, scoped names always need to be fully scoped (because they always start in the global scope). Also see *Blocks, Scopes, and accessing Symbols*:

```
main.start              ; the entrypoint subroutine
main.start.variable     ; a variable in the entrypoint subroutine
```

5.3.4 Code blocks

A named block of actual program code. It defines 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 initial value yourself, zero will be used. The syntax for variable declarations is:

```
<datatype>  [ @tag ]  <variable name>   [ = <initial value> ]
```

Here are the tags you can add to a variable:

Tag	Effect
@zp	prioritize the variable for putting it into Zero page. No guarantees; if ZP is full the variable will be placed in another memory location.
@requirezp	force the variable into Zero page. If ZP is full, compilation will fail.
@shared	means the variable is shared with some assembly code and that it cannot be optimized away if not used elsewhere.
@split	(only valid on (u)word arrays) Makes the array to be placed in memory as 2 separate byte arrays; one with the LSBs one with the MSBs of the word values. May improve performance.

For boolean and numeric variables, you can actually declare them in one go by listing the names in a comma separated list. Type tags, and the optional initialization value, are applied equally to all variables in such a list.

Various examples:

```
word        thing   = 0
byte        counter = len([1, 2, 3]) * 20
byte        age     = 2018 - 1974
float       wallet  = 55.25
ubyte       x,y,z                   ; declare three ubyte variables x y and z
```

(continues on next page)

(continued from previous page)

```
str         name    = "my name is Alice"
uword       address = &counter
bool        flag    = true
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'`
`bool`	boolean	1 byte = 8 bits	`bool myvar = true` or `bool myvar == false`
`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`
`bool[x]`	boolean array	5*x bytes	`bool[4] myvar` note: consider using bit flags in a byte or word instead to save space
`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]`
`bool[]`	boolean array	depends on value	`bool[] myvar = [true, false, true]` note: consider using bit flags in a byte or word instead to save space
`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.

numbers: unless prefixed for hex or binary as described below, all numbers are decimal numbers. There is no octal notation.

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. Otherwise the parser thinks you've typed an invalid binary number.

digit grouping: for any number you can use underscores to group the digits to make the number more readable. Any underscores in the number are ignored by the compiler. For instance `%1001_0001` is a valid binary number and `3_000_000.99` is a valid floating point number.

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 `word ww = bytevalue as word` will convert the byte value to a signed word.

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 ('peek' and 'poke')

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, and the index value can be larger than a byte in this case:

```
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).

Only the simple numeric types (byte, word, float) can be defined as a constant:

```
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). Unlike the start and end values, the step value must be a constant. 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:

```
0 to 7                  ; range of values 0, 1, 2, 3, 4, 5, 6, 7
20 downto 10 step -3    ; range of values 20, 17, 14, 11

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

byte[] array = 10 to 13  ; sets the array to [10, 11, 12, 13]

for  i  in  0 to 127  {
    ; i loops 0, 1, 2, ... 127
}
```

Range expressions are most often used in for loops, but can be also be used to create array initialization 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. Negative index means counted from the end of the array rather than the beginning, where -1 means the last element in the array, -2 the second-to-last, etc. (Python uses this same scheme) Use brackets to index into an array: `arrayvar[x]`

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

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 (and allows index values of word size). See *Direct access to memory locations ('peek' and 'poke')*

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 examples of the various encodings:

- `"hello"` a string translated into the default character encoding (PETSCII on the CBM machines)
- `petscii:"hello"` string in CBM PETSCII encoding
- `sc:"my name is Alice"` string in CBM screencode encoding
- `iso:"Ich heiße François"` string in iso-8859-15 encoding (Latin)
- `iso5:"  "` string in iso-8859-5 encoding (Cyrillic)
- `iso16:"zażółć gęślą jaźń"` string in iso-8859-16 encoding (Eastern Europe)
- `atascii:"I am Atari!"` string in "atascii" encoding (Atari 8-bit)
- `cp437:" IBM Pc  ♪¶"` string in "cp437" encoding (IBM PC codepage 437)

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 after the %, it will be parsed as a binary number. So `25 %10` will be parsed as the number 25 followed by the binary number 2, which is a syntax error. Note that 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. It's possible to chain assignments like `x = y = z = 42` as a shorthand for the three assignments with the same value.

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 += 1`, and `aa--` is equivalent to `aa -= 1`. Because these operations are so common, and often used in other languages, we have these short forms. *Notes:* unlike some other languages, they are *not* expressions in prog8, but statements. You cannot increment or decrement something inside an expression like, for example, `x = value[aa++]` is invalid. Also because of this, there is no *prefix* increment and decrement.

comparison: == != < > <= >=

Equality, Inequality, Less-than, Greater-than, Less-or-Equal-than, Greater-or-Equal-than comparisons. The result is a boolean, true or false.

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, true or false. Prog8 applies short-circuit aka McCarthy evaluation for `and` and `or`.

range creation: `to`, `downto`

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. See *Range expression* for details.

containment check: `in`

Tests if a value is present in a list of values, which can be a string, or an array, or a range expression. 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. Checking N in a range from x to y, is identical to x<=N and N<=y; the actual range of values is never created. Examples:

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

if cc in 10 to 20 {
    txt.print("10 <= cc and cc <=20")
}

str email_address = "name@test.com"
```

(continues on next page)

(continued from previous page)

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

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 an external routine in ROM or elsewhere in memory) 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 and some more values in R0 and R1. In all of these cases, you have to "multi assign" all return values of the subroutine call to something. You simply write the assignment targets as a comma separated list, where the element's order corresponds to the order of the return values declared in the subroutine's signature. So for instance:

```
bool   flag
ubyte  bytevar
uword  wordvar

wordvar, flag, bytevar = multisub()        ; call and assign the three result values

asmsub multisub() -> uword @AY, bool @Pc, ubyte @X { ... }
```

Using just one of the values

Sometimes it is easier to just have a single return value in the subroutine's signagure (even though it actually may return multiple values): this avoids having to put `void` for all other values. It also allows it to be called in expressions such as if-statements again. Examples of these second 'convenience' definition are library routines such as `cbm.STOP2` and `cbm.GETIN2`, that only return a single value where the "official" versions `STOP` and `GETIN` always return multiple values.

Skipping values: Instead of using `void` to ignore the result of a subroutine call altogether, you can also use it as a placeholder name in a multi-assignment. This skips assignment of the return value in that place. One of the cases where this is useful, is with boolean values returned in status flags such as the carry flag. Storing that flag as a boolean in a variable first, and then possibly adding an `if flag...` statement afterwards, is a lot less efficient than just keeping the flag as-is and using a conditional branch such as `if_cs` to do something with it. So in the case above that could be:

```
wordvar, void, bytevar = multisub()
if_cs
    something()
```

Notice that a call to a subroutine that returns multiple values cannot be used inside an expression, because expression terms always need to be a single value. You'll have to use a separate multi-assignment first and then use the result of that in the expression. However, also read the sidebar about a possible alternative.

5.3.9 Subroutine definitions

The syntax is:

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

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

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

External subroutines implemented in ROM (or elsewhere in memory) are usually defined by compiler library files, with the following syntax:

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

This defines the `LOAD` subroutine at 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-written subroutines in the program source code itself, 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  cbm.Screen,y
        sta  cbm.Screen+$0100,y
        sta  cbm.Screen+$0200,y
        sta  cbm.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), special 16-bit register pairs to take word values (AX, AY and XY) and even a processor status flag such as Carry (Pc).

It is not possible to use floating point arguments or return values in an asmsub.

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). Remember that a step value in a range must be a constant value.

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
    continue    ; immediately next iteration
}
```

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
}
```

To loop over a decreasing or descending range, use the `downto` keyword:

```
ubyte i

...

for i in 155 downto 20 {        ; 155, 154, 153, ..., 20
    ; do something
}
```

Similarly, a descending range may be specified by using `to` in combination with a `step` that is `< 0`:

```
ubyte i

...

for i in 155 to 20 step -1 {    ; 155, 154, 153, ..., 20
    ; do something
}
```

The following example 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 {
```

(continues on next page)

(continued from previous page)

```
    ; do something with number...
    break       ; break out of the loop early
}
```

See *Range expression* for all of the details.

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
        continue    ; immediately next iteration
}
```

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
        continue    ; immediately next iteration
} 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
    continue    ; immediately next iteration
}
```

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

unroll loop

Like a repeat loop, but trades memory for speed by not generating the code for the counter. Instead it duplicates the code inside the loop on the spot for the given number of iterations. This means that only a constant number of iterations can be specified. Also, only simple statements such as assignments and function calls can be inside the loop:

```
unroll 80 {
    cx16.VERA_DATA0 = 255
}
```

A *break* or *continue* statement cannot occur in an unroll loop, as there is no actual loop to break out of.

5.3.12 Conditional Execution and Jumps

Unconditional jump: goto

To jump to another part of the program, you use a `goto` statement with an address or the name of a label or subroutine. Referencing labels or subroutines outside of their defined scope requires using qualified "dotted names":

```
goto  $c000           ; address
goto  name            ; label or subroutine
goto  main.mysub.name ; qualified dotted name; see, "Blocks, Scopes, and accessing␣
↪Symbols"

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.

if statements

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

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

If <statements> is just a single statement, for instance just a `goto` or a single assignment, it's possible to just write the statement without any curly braces. However if <statements> is a block of multiple statements, you'll have to enclose it in curly braces:

```
if  <expression> {
        <statements>
} else if <expression> {
        <statements>
} else {
        <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 conditional 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 or a block again:

```
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 function 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 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.

Note: Several algorithms and math routines in Prog8's assembly library files are adapted from code publicly available on https://www.codebase64.org/

5.4.1 Low-fi variable and subroutine definitions in all available library modules

These are auto generated and contain no documentation, but provide a view into what's available. Grouped per compilation target.

- atari
- c64
- c128
- cx16
- pet32
- virtual

5.4.2 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.

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.

This module is usually imported automatically and can provide definitions in the `sys`, `cbm`, `c64`, `cx16`, `c128`, `atari` blocks depending on the chosen compilation target. Read the sys lib source code for the correct compilation target to see exactly what is there.

5.4.3 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:

- 8 = Atari 8 bits
- 16 = Commander X16
- 64 = Commodore 64
- 128 = Commodore 128
- 255 = Virtual machine

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!

exit2 (resultA, resultX, resultY)
Immediately stops the program and exits it, with the result values in the A, X and Y registers. Note: custom interrupt handlers remain active unless manually cleared first!

exit3 (resultA, resultX, resultY, carry)
Immediately stops the program and exits it, with the result values in the A, X and Y registers, and the carry flag in the status 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.

irqsafe_set_irqd ()
Sets the CPU status register Interrupt Disable flag, in a way that is safe to be used inside a IRQ handler. Pair with `irqsafe_clear_irqd()`.

irqsafe_clear_irqd ()
Clears the CPU status register Interrupt Disable flag, in a way that is safe to be used inside a IRQ handler. Pair with `irqsafe_set_irqd()`. Inside an IRQ handler this makes sure it doesn't inadvertently clear the irqd status bit, and it can still be used inside normal code as well (where it *does* clear the irqd status bit if it was cleared before entering).

progend ()
Returns the last address of the program in memory + 1. This means: the memory address directly after all the program code and variables, including the uninitialized ones ("BSS" variables) and the uninitialized memory blocks reserved by the *memory()* function. 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 regular system irq handler has run for this to work as it depends on the system jiffy clock. If this is is not possible (for instance because your program is running its own irq handler logic *and* no longer calls the kernal's handler routine), you'll have to write your own wait routine instead.

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 regular system irq handler has to run for this to work (this is not required on C64 and C128)

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)

disable_caseswitch() and enable_caseswitch()
Disable or enable the ability to switch character set case using a keyboard combination.

save_prog8_internals() and restore_prog8_internals()
Normally not used in user code, the compiler utilizes these for the internal interrupt logic. It stores and restores the values of the internal prog8 variables. This allows other code to run that might clobber these values temporarily.

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

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

pop ()
pops a byte value off the CPU hardware stack and returns it. Low-level function that should normally not be used.

popw ()
pops a 16-bit word value off the CPU hardware stack and returns it. Low-level function that should normally not be used.

5.4.4 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)

Read the conv source code to see what's in there.

5.4.5 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, numbers and booleans
- 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
- convert petscii to screencode characters

All routines work with Screencode character encoding, except *print*, *chrout* and *input_chars*, these work with PETSCII encoding instead.

Read the textio source code to see what's in there. (Note: slight variations for different compiler targets)

5.4.6 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
- send arbitrary CbmDos command to disk drive

Commander X16 additions: Headerless load and save routines are available (load_raw, save_raw). On the Commander X16 it tries to use that machine's fast Kernal loading routines if possible. Routines to directly load data into video ram are also present (vload and vload_raw). Also contains a helper function to calculate the file size of a loaded file (although that is truncated to 16 bits, 64Kb) Als contains routines for operating on subdirectories (chdir, mkdir, rmdir), to relabel the disk, and to seek in open files.

Read the diskio source code to see what's in there. (Note: slight variations for different compiler targets)

Note: If you are using the X16 emulator with HostFS, and are experiencing weird behavior with these routines, please first try again with an SD-card image instead of HostFs. It is possible that there are still small differences between HostFS and actual CBM DOS in the X16 emulator.

Note: You can set the active disk drive number so it supports multiple drives, but it does not support multiple open files at the same time.

Attention: Error handling is peculiar on CBM dos systems (C64, C128, cx16, PET). Read the descriptions for the various methods in this library for details and tips.

5.4.7 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 0+carry bit clear if the character was not found. You can consider this a safer way of checking if a character occurs in a string than using an *in* containment check - because the find routine properly stops at the first 0-byte string terminator it encounters. Simply call this and only act on the carry status with `if_cc` for example. Much like the difference between len(str) and length(str).

`contains (string, char) -> bool`
: Just returns true if the character is in the given string, or false if it's not. For string literals, you can use a containment check expression instead: `char in "hello world"`.

`compare (string1, string2) -> ubyte result`
: Returns -1, 0 or 1 depending on whether string1 sorts before, equal or after string2. Note that you can also directly compare strings and string values with each other using ==, < etcetera (it will use string.compare for you under water automatically). This even works when dealing with uword (pointer) variables when comparing them to a string type.

`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.

append (string, suffix) -> ubyte length
Appends the suffix string to the other string (make sure the memory buffer is large enough!) Returns the length of the combined string.

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

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

lowerchar (char)
Returns lowercased PETSCII character.

upperchar (char)
Returns uppercased PETSCII character.

strip (string)
Gets rid of whitespace and other non-visible characters at the edges of the string. (destructive)

rstrip (string)
Gets rid of whitespace and other non-visible characters at the end of the string. (destructive)

lstrip (string)
Gets rid of whitespace and other non-visible characters at the start of the string. (destructive)

lstripped (string) -> str
Returns pointer to first non-whitespace and non-visible character at the start of the string (non-destructive lstrip)

trim (string)
Gets rid of whitespace characters at the edges of the string. (destructive)

rtrim (string)
Gets rid of whitespace characters at the end of the string. (destructive)

ltrim (string)
Gets rid of whitespace characters at the start of the string. (destructive)

ltrimmed (string) -> str
Returns pointer to first non-whitespace character at the start of the string (non-destructive ltrim)

isdigit (char)
Returns boolean if the character is a numerical digit 0-9

islower (char), isupper (char), isletter (char)
Returns true if the character is a shifted-PETSCII lowercase letter, uppercase letter, or any letter, respectively.

isspace (char)
Returns true if the PETSCII character is a whitespace (tab, space, return, and shifted versions)

isprint (char)
Returns true if the PETSCII character is a "printable" character (space or any visible symbol)

startswith (string, prefix) -> bool
Returns true if string starts with prefix, otherwise false

endswith (string, suffix) -> bool
Returns true if string ends with suffix, otherwise false

pattern_match (string, pattern) -> bool (not on Virtual target)
Returns true if the string matches the pattern, false if not. '?' in the pattern matches any one character. '*' in the pattern matches any substring.

hash (string) -> ubyte
Returns a simple 8 bit hash value for the given string. The formula is: hash(-1)=179; clear carry; hash(i) = ROL

hash(i-1) XOR string[i] (where ROL is the cpu ROL instruction) On the English word list in /usr/share/dict/words it seems to have a pretty even distribution.

5.4.8 floats

Note: Floating point support is only available on c64, cx16 and virtual targets for now.

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.

and PI
float const for the number Pi, 3.141592653589793…

TWOPI
float const for the number 2 times Pi

atan (x)
Arctangent.

atan2 (y, x)
Two-argument arctangent that returns an angle in the correct quadrant for the signs of x and y, normalized to the range [0, 2]

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

cos (x)
Cosine.

cot (x)
Cotangent: 1/tan(x)

csc (x)
Cosecant: 1/sin(x)

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.

minf (x, y)
returns the smallest of x and y.

maxf (x, y)
returns the largest of x and y.

clampf (value, minimum, maximum)
returns the value restricted to the given minimum and maximum.

print (x)
Prints the floating point number x as a string. There's no leading whitespace (unlike cbm BASIC when printing a floating point number)

tostr (x)
Converts the floating point number x to a string (returns address of the string buffer) There's no leading whitespace.

rad (x)
Degrees to radians.

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

sin (x)
Sine.

secant (x)
Secant: 1/cos(x)

tan (x)
Tangent.

rnd ()
returns the next random float between 0.0 and 1.0 from the Pseudo RNG sequence.

rndseed (seed)
Sets a new seed for the float pseudo-RNG sequence. Use a negative non-zero number as seed value.

parse (stringvalue)
Parses the string value as floating point number. Warning: this routine may stop working on the Commander X16 when a new ROM version is released, because it uses an internal BASIC routine. Then it will require a fix.

5.4.9 graphics

Bitmap graphics routines:

- 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. On the cx16 there's also the `gfx2` library if you want more features and different screen modes. See below for that one.

Read the graphics source code to see what's in there. (Note: slight variations for different compiler targets)

5.4.10 math

Low-level integer math routines (which you usually don't have to bother with directly, but they are used by the compiler internally). Pseudo-Random number generators (byte and word). Various 8-bit integer trig functions that use lookup tables to quickly calculate sine and cosines. Usually a custom lookup table is the way to go if your application needs these, but perhaps the provided ones can be of service too.

log2 (ubyte v)
Returns the 2-Log of the byte value v.

log2w (uword v)
Returns the 2-Log of the word value v.

rnd ()
Returns next random byte 0-255 from the pseudo-RNG sequence.

rndw ()
Returns next random word 0-65535 from the pseudo-RNG sequence.

randrange (ubyte n) -> ubyte
Returns random byte uniformly distributed from 0 to n-1 (compensates for divisibility bias)

randrangew (uword n) -> uword
Returns random word uniformly distributed from 0 to n-1 (compensates for divisibility bias)

rndseed (uword seed1, uword seed2)
Sets a new seed for the pseudo-RNG sequence (both rnd and rndw). The seed consists of two words. Do not use zeros for the seed!

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

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!

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

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!

atan2 (ubyte x1, ubyte y1, ubyte x2, ubyte y2)
Fast arctan routine that uses more memory because of large lookup tables. Calculate the angle, in a 256-degree circle, between two points in the positive quadrant.

direction (ubyte x1, ubyte y1, ubyte x2, ubyte y2)
From a pair of positive coordinates, calculate discrete direction between 0 and 23. This is a heavily optimized routine (small and fast).

direction_sc (byte x1, byte y1, byte x2, byte y2)
From a pair of signed coordinates around the origin, calculate discrete direction between 0 and 23. This is a heavily optimized routine (small and fast).

direction_qd (ubyte quadrant, ubyte xdelta, ubyte ydelta)
If you already know the quadrant and x/y deltas, calculate discrete direction between 0 and 23. This is a heavily optimized routine (small and fast).

diff (ubyte b1, ubyte b2) -> ubyte
Returns the absolute difference, or distance, between the two byte values. (This routine is more efficient than doing a compare and a subtract separately, or using abs)

diffw (uword w1, uword w2) -> uword
Returns the absolute difference, or distance, between the two word values. (This routine is more efficient than doing a compare and a subtract separately, or using abs)

mul16_last_upper () -> uword
Fetches the upper 16 bits of the previous 16*16 bit multiplication. To avoid corrupting the result, it is best performed immediately after the multiplication. Note: It is only for the regular 6502 cpu multiplication routine. It does not work for the verafx multiplication routines on the Commander X16! These have a different way to obtain the upper 16 bits of the result: just read cx16.r0.

crc16 (uword data, uword length) -> uword
Returns a CRC-16 (XMODEM) checksum over the given data buffer. Note: on the Commander X16, there is a CRC-16 routine in the kernal: cx16.memory_crc(). That one is faster, but yields different results. It is unclear to me what flavour of crc it is calculating.

crc16_start() / crc16_update(ubyte value) / crc16_end() -> uword
"streaming" crc16 calculation routines, when the data doesn't fit in a single buffer. Tracks the crc16 checksum in cx16.r15! If your code uses that, it must save/restore it before calling this routine! Call the start() routine first, feed it bytes with the update() routine, finalize with calling the end() routine which returns the crc16 value.

crc32 (uword data, uword length)
Calculates a CRC-32 (POSIX) checksum over the given data buffer. The 32 bits result is stored in cx16.r14 (low word) and cx16.r15 (high word).

crc32_start() / crc32_update(ubyte value) / crc32_end()
"streaming" crc32 calculation routines, when the data doesn't fit in a single buffer. Tracks the crc32 checksum in cx16.r14 and cx16.r15! If your code uses these, it must save/restore them before calling this routine! Call the start() routine first, feed it bytes with the update() routine, finalize with calling the end() routine. The 32 bits result is stored in cx16.r14 (low word) and cx16.r15 (high word).

5.4.11 cx16logo

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

logo ()
prints the logo at the current cursor position

logo_at (column, row)
printss the logo at the given position

5.4.12 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.

5.4.13 cx16

This is available on *all targets*, it is always imported as part of syslib. On the Commander X16 this module contains a *whole bunch* of things specific to that machine. It's way too much to include here, you have to study the syslib source code to see what is there.

On the other targets, it only contains the definition of the 16 memory mapped virtual registers (cx16.r0 - cx16.r15) and the following utility routines:

`save_virtual_registers()`
: save the values of all 16 virtual registers r0 - r15 in a buffer. Might be useful in an IRQ handler to avoid clobbering them.

`restore_virtual_registers()`
: restore the values of all 16 virtual registers r0 - r15 from the buffer. Might be useful in an IRQ handler to avoid clobbering them.

`cpu_is_65816()`
: Returns true if the CPU in the computer is a 65816, false otherwise (6502 cpu).

`reset_system ()`
: Soft-reset the system back to initial power-on BASIC prompt. (same as the routine in sys)

`poweroff_system ()`
: Powers down the computer.

`set_led_brightness (ubyte brightness)`
: Sets the brightness of the activity led on the computer.

5.4.14 bmx (cx16 only)

Routines to load and save "BMX" files, the CommanderX16 bitmap file format. Specification available here: https://cx16forum.com/forum/viewtopic.php?t=6945 Only *uncompressed* bitmaps are supported in this library for now.

The routines are designed to be fast and bulk load/save the data directly into or from vram, without the need to buffer something in main memory.

For details about what routines are available, have a look at the bmx source code . There's also the "showbmx" example to look at.

5.4.15 emudbg (cx16 only)

X16Emu Emulator debug routines, for Cx16 only. Allows you to interface with the emulator's debug routines/registers. There's stuff like `is_emulator` to detect if running in the emulator, and `console_write` to write a (iso) string to the emulator's console (stdout) etc.

Read the emudbg source code to see what's in there. Information about the exposed debug registers is in the emulator's documentation.

5.4.16 monogfx (cx16 and virtual)

Full-screen lores or hires monochrome bitmap graphics routines, available on the Cx16 machine only. Same interface as gfx2, but is optimized for monochrome (1 bpp) screens.

- lores 320*240 or hires 640*480 bitmap mode, monochrome
- clearing screen, switching screen mode, also back to text mode
- drawing and reading individual pixels
- drawing lines, rectangles, filled rectangles, circles, discs
- flood fill
- drawing text inside the bitmap
- can draw using a stipple pattern (alternate black/white pixels) and in invert mode (toggle pixels)

Read the monogfx source code to see what's in there.

5.4.17 gfx2 (cx16 only)

Full-screen multicolor bitmap graphics routines, available on the Cx16 machine only. Same interface as monogfx, but for color screens. For 1 bpp monochrome screens, use monogfx.

- multiple full-screen bitmap color resolutions
- clearing screen, switching screen mode, also back to text mode
- drawing and reading individual pixels
- drawing lines, rectangles, filled rectangles, circles, discs
- flood fill
- drawing text inside the bitmap

Read the gfx2 source code to see what's in there.

5.4.18 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 C64.

Read the palette source code to see what's in there.

5.4.19 psg (cx16 only)

Available for the Cx16 target. Contains a simple abstraction for the Vera's PSG (programmable sound generator) to play simple waveforms. It includes an interrupt routine to handle simple Attack/Release envelopes as well. See the examples/cx16/bdmusic.p8 program for ideas how to use it.

Read the psg source code to see what's in there.

5.4.20 sprites (cx16 only)

Available for the Cx16 target. Simple routines to manipulate sprites. They're not written for high performance, but for simplicity. That's why they control one sprite at a time. The exception is the `pos_batch` routine, which is quite efficient to update sprite positions of multiple sprites in one go. See the examples/cx16/sprites/dragon.p8 and dragons.p8 programs for ideas how to use it.

Read the sprites source code to see what's in there.

5.4.21 verafx (cx16 only)

Available for the Cx16 target. Experimental routines that use the new Vera FX logic (hopefully coming in the Vera in new X16 boards, the emulators already support it).

`available`
: Returns true if Vera FX is available, false if not (that would be an older Vera chip)

`mult`, `muls`
: The hardware 16*16 multiplier is exposed via `mult` and `muls` routines (unsigned and signed respectively). They are about 4 to 5 times faster as the default 6502 cpu routine for word multiplication. But they depend on some Vera manipulation and 4 bytes in vram just below the PSG registers for storage. Note: there is a block level %option "verafxmuls" that automatically replaces all word multiplications in that block by calls to verafx.muls/mult, but be careful with it because it may interfere with other Vera operations or IRQs.

  Note: the lower 16 bits of the 32 bits result is returned as the normal subroutine's returnvalue, but the upper 16 bits is returned in cx16.r0 so you can still access those separately.

`clear`
: Very quickly clear a piece of vram to a given byte value (it writes 4 bytes at a time). The routine is around 3 times faster as a regular unrolled loop to clear vram.

`copy`
: Very quickly copy a portion of the video memory to somewhere else in vram (4 bytes at a time) Sometimes this is also called "blitting". This routine is about 50% faster as a regular byte-by-byte copy.

`transparency`
: Enable or disable transparent writes (color 0 will be transparent if enabled).

Read the verafx source code to see what's in there.

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*)

- 'pet32': the Commodore PET 4032 (*limited support*)
- 'atari': the Atari 800 XL (*experimental support*)
- 'virtual': a builtin virtual machine

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 often possible to compile the *exact same program* for different machines (just change the compilation target flag)!

5.5.1 Memory Model

Generic 6502 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 support for RAM expansions or bank switching built natively into the language.

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 depending on the specific computer system

Memory map for the C64 and the X16

This is the default memory map of the 64 Kb addressable memory for those two systems. Both systems have ways to alter the memory map and/or to switch memory banks, but that is not shown here.

Footnotes for the Commander X16

Golden Ram $0400 - $07FF
free to use.

Zero Page $0000 - $00FF
$00 and $01 are hardwired as Rom and Ram banking registers.

$02 - $21 are the 16 virtual cx16 registers R0-R15.

$22 - $7F are used by Prog8 to put variables in.

The top half of the ZP ($80-$FF) is reserved for use by the Kernal and Basic in normal operation. Zero page use by Prog8 can be manipulated with the `%zeropage` directive, various options may free up more locations for use by Prog8 or to reserve them for other things.

Footnotes for the Commodore 64

RAM $C000-$CFFF
free to use: $C000 - $CFDF *reserved:* $CFE0 - $CFFF for the 16 virtual cx16 registers R0-R15

Zero Page $0000 - $00FF
Consider the full zero page to be reserved for use by the Kernal and Basic in normal operation. Zero page use by Prog8 can be manipulated with the `%zeropage` directive, various options may free up more locations for use by Prog8 or to reserve them for other things.

Zero page usage by the Prog8 compiler

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 restrictive 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 C64'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.

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 60hz/vsync IRQs and raster/line IRQs a lot easier (no assembly code required).

These routines are:

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

The IRQ handler routine must return a boolean value (0 or 1) in the A register: 0 means do *not* run the system IRQ handler routine afterwards, 1 means run the system IRQ handler routine afterwards.

CommanderX16 specific notes

Note that for the CommanderX16 the set_rasterirq() will disable VSYNC irqs and never call the system IRQ handler regardless of the return value of the user handler routine. This also means the default sys.wait() routine won't work anymore, when using this handler.

These two helper routines are not particularly suited to handle multiple IRQ sources on the Commander X16. It's possible but it requires correct fiddling with IRQ enable bits, acknowledging the IRQs, and properly calling or not calling the system IRQ handler routine. See the section below for perhaps a better and easier solution that is tailored to this system.

The Commander X16 syslib provides some additional routines that should be used *in your IRQ handler routine* if it uses the Vera registers. They take care of saving and restoring the Vera state of the interrupted main program, otherwise the IRQ handler's manipulation will corrupt any Vera operations that were going on in the main program. The routines are:

```
cx16.save_vera_context()
; perhaps also cx16.save_virtual_registers() here...
; ... do your work that uses vera here!...
; perhaps also cx16.restore_virtual_registers() here...
cx16.restore_vera_context()
```

Caution: The Commander X16's 16 'virtual registers' R0-R15 are located in zeropage and *are not preserved* in the IRQ handler! So you should make sure that the handler routine does NOT use these registers, or do some sort of saving/restoring yourself of the ones that you do need in the IRQ handler. There are two utility routines in cx16 that save and restore *all* 16 registers so it's a bit inefficient but safe. (these are `save_virtual_registers()` and `restore_virtual_registers()`)

It is also advised to not use floating point calculations inside IRQ handler routines. Beside them being very slow, there are intricate requirements such as having the correct ROM bank enabled to be able to successfully call them (and making sure the correct ROM bank is reset at the end of the handler), and the possibility of corrupting variables and floating point calculations that are being executed in the interrupted main program. These memory locations should be backed up and restored at the end of the handler, further increasing its execution time…

5.5.4 Commander X16 specific IRQ handling

Instead of using the routines in `sys` as mentioned above (that are more or less portable across the C64,C128 and cx16), you can also use the special routines made for the Commander X16, in `cx16`. The idea is to let Prog8 do the irq dispatching and housekeeping for you, and that your program only has to register the specific handlers for the specific IRQ sources that you want to handle.

Look at the examples/cx16/multi-irq-new.p8 example to see how these routines can be used. Here they are, all available in `cx16`:

`disable_irqs ()`
: Disables all Vera IRQ sources. Note that the CPU irq disable flag is not changed by this routine. you can manipulate that via `sys.set_irqd()` and `sys.clear_irqd()` as usual.

`enable_irq_handlers (bool disable_all_irq_sources)`
: Install the "master IRQ handler" that will dispatch IRQs to the registered handler for each type. Only Vera IRQs supported for now. Pass true to initially disable all Vera interrupt sources (they will be enabled individually again by setting the various handlers), or pass false to not touch this. The handlers don't need to clear its ISR bit, but have to return 0 or 1 in A, where 1 means: continue with the system IRQ handler, 0 means: don't call that. The order in which the handlers are invoked if multiple interrupts occur simultaneously is: LINE, SPRCOL, AFLOW, VSYNC.

`set_vsync_irq_handler (uword address)`
: Sets the verical sync interrupt handler routine. Also enables VSYNC interrupts.

`set_line_irq_handler (uword rasterline, uword address)`
: Sets the rasterline interrupt handler routine to trigger on the specified raster line. Also enables LINE interrupts. You can use `sys.set_rasterline()` later to adjust the rasterline on which to trigger.

`set_sprcol_irq_handler (uword address)`
: Sets the sprite collision interrupt handler routine. Also enables SPRCOL interrupts.

`set_aflow_irq_handler (uword address)`
: Sets the audio buffer underrun interrupt handler routine. Also enables AFLOW interrupts.

`disable_irq_handlers ()`
: Hand control back to the system 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 of course 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.

Variables that are not put into zeropage, will be put into a special 'BSS' section for the assembler. This section is usually placed at the end of the resulting program but because it only contains empty space it won't actually increase the size of the resulting program binary. Prog8 takes care of properly filling this memory area with zeros at program startup and then reinitializes the subset of variables that have a nonzero initialization value.

Arrays with initialization values are not put into BSS but just occupy a sequence of bytes in the program memory: their values are not reinitialized at program start.

It is possible to relocate the BSS section using a compiler option so that more system ram is available for the program code itself.

5.6.2 Symbol prefixing in generated Assembly code

All symbols in the prog8 program will be prefixed in the generated assembly code:

Element type	prefix
Block	`p8b_`
Subroutine	`p8s_`
Variable	`p8v_`
Constant	`p8c_`
Label	`p8l_`
other	`p8_`

This is to avoid naming conflicts with CPU registers, assembly instructions, etc. So if you're referencing symbols from the prog8 program in inlined assembly code, you have to take this into account. Stick the proper prefix in front of every symbol name component that you want to reference that is coming from a prog8 source file. All elements in scoped names such as `main.routine.var1` are prefixed so this becomes `p8b_main.p8s_routine.p8v_var1`.

Attention: Symbols from library modules are *not* prefixed and can be used in assembly code as-is. So you can write:

```
%asm {{
    lda  #'a'
    jsr  cbm.CHROUT
}}
```

5.6.3 Subroutine Calling Convention

Calling a subroutine requires three steps:

1. preparing the arguments (if any) and passing them to the routine. Numeric types are passed by value (bytes, words, booleans, floats), but array types and strings are passed by reference which means as `uword` being a pointer to their address in memory.
2. calling the subroutine
3. preparing the return value (if any) and returning that from the call.

`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 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 purpose 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 Compiler Internals

Here is a diagram of how the compiler translates your program source code into a binary program:

Some notes and references into the compiler's source code modules:

1. The `compileProgram()` function (in the `compiler` module) does all the coordination and basically drives all of the flow shown in the diagram.
2. ANTLR is a Java parser generator and is used for initial parsing of the source code. (`parser` module)
3. Most of the compiler and the optimizer operate on the *Compiler AST*. These are complicated syntax nodes closely representing the Prog8 program structure. (`compilerAst` module)
4. For code generation, a much simpler AST has been defined that replaces the *Compiler AST*. Most notably, node type information is now baked in. (`codeCore` module, Pt- classes)
5. An *Intermediate Representation* has been defined that is generated from the intermediate AST. This IR is more or less a machine code language for a virtual machine - and indeed this is what the built-in prog8 VM will execute if you use the 'virtual' compilation target and use `-emu` to launch the VM. (`intermediate` and `codeGenIntermediate` modules, and `virtualmachine` module for the VM related stuff)
6. The code generator backends all implement a common interface `ICodeGeneratorBackend` defined in the `codeCore` module. Currently they get handed the program Ast, Symboltable and several other things. If the code generator wants it can use the `IRCodeGen` class from the `codeGenIntermediate` module to convert the Ast into IR first. The VM target uses this, but the 6502 codegen doesn't right now.

5.6.5 Upgrading from version 8

Version 9 introduced several large, incompatible changes. If you still have programs written for Prog8 version 8 or earlier, it is likely that you'll have to modify them to be able to compile with version 9 or newer.

Information about this can be found in older Prog8 documentation .

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.

Note: The assembly code that prog8 generates is not suitable to be put into ROM. (It contains embedded variables, and self-modifying code). If the target system is designed to run programs from ROM, and has just a little bit of RAM intended for variables, prog8 is likely not a feasible language for such a system right now.

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

Zeropage

1. *Absolute requirement:* Provide three times 2 consecutive bytes (i.e. three 16-bit pointers) in the zeropage that are free to use at all times.
2. Provide list of any additional free zeropage locations for a normal running system (BASIC + Kernal enabled)
3. Provide list of any additional free zeropage locations when BASIC is off, but floating point routines should still work
4. Provide list of any additional free zeropage 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 zeropage 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 part(s) of the address space is ROM?
4. what part(s) of the address space is memory mapped I/O registers?
5. is there a block of "high ram" available (ram that is not the main ram used to load programs in) that could be used for variables?

6. is there a banking system? How does it work (how do you select Ram/Rom banks)? How is the default bank configuration set? Note that prog8 itself has no notion of banking, but this knowledge may be required for proper system initialization.

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 can support floating point math *if* the target system has floating point math routines in ROM. If that is the case:

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. But not required.

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

Also of course 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 Software written in Prog8

Apart from the many examples available in the source code repository, there are also larger pieces of software written using Prog8. Here's a list.

Assembler
File-based assembler for the Commander X16.

Chess
Chess game for the Commander X16, with 2-player or computer opponent game modes.

Image viewer
Multi-format image viewer for the Commander X16. Can display cx16 BMX, C64 Koala, C64 Doodle, BMP, PCX and Amiga IFF images, including color cycling.

Paint program
Bitmap image paint program for the Commander X16, work in progress.

Petaxian
Galaga type shoot em up game using only petscii graphics. Runs on C64 and Commander X16.

Rock Runner
Faithful Boulderdash clone, a well known arcade puzzle game from the 80's. where you must collect all diamonds in a level while avoiding the hazards to reach the exit. Can load the thousands of available fan made level files. This game is for the Commander X16.

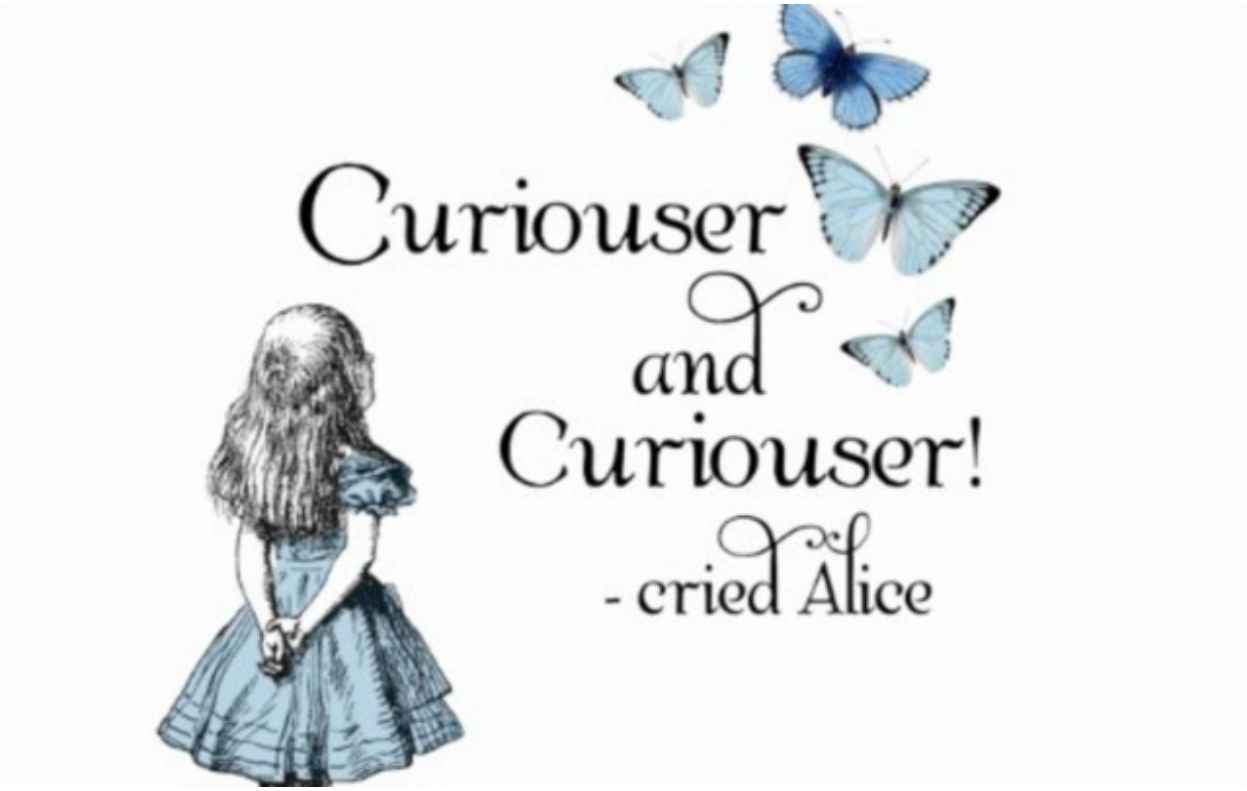
Shell
Unix like command shell for the Commander X16.

Streaming Music Demo
Demoscene like "music demos" for the Commander X16. They display graphics, animated song lyrics, and play a high quality sampled song streamed from disk.

Various things:

GalaX16 and other programs
Beginnings of a Galaga game for the Commander X16.

Prog8 code for ZSMkit
ZSMkit is an advanced music and sound effects engine for the Commander X16.



5.9 TODO

…

5.9.1 Future Things and Ideas

Compiler:

- IR: implement missing operators in AssignmentGen (array shifts etc)
- can we support signed % (remainder) somehow?
- instead of copy-pasting inline asmsubs, make them into a 64tass macro and use that instead. that will allow them to be reused from custom user written assembly code as well.
- Multidimensional arrays and chained indexing, purely as syntactic sugar over regular arrays.
- make a form of "manual generics" possible like: varsub routine(T arg)->T where T is expanded to a specific type (this is already done hardcoded for several of the builtin functions)
- **[much work:] more support for (64tass) SEGMENTS ?**
  - (What, how, isn't current BSS support enough?)
  - Add a mechanism to allocate variables into golden ram (or segments really) (see GoldenRam class)
  - maybe treat block "golden" in a special way: can only contain vars, every var will be allocated in the Golden ram area?
  - maybe or may not needed: the variables can NOT have initialization values, they will all be set to zero on startup (simple memset) just initialize them yourself in start() if you need a non-zero value .

  - OR.... do all this automatically if 'golden' is enabled as a compiler option? So compiler allocates in ZP first, then Golden Ram, then regular ram
  - OR.... make all this more generic and use some %segment option to create real segments for 64tass?
  - (need separate step in codegen and IR to write the "golden" variables)
- do we need (array)variable alignment tag instead of block alignment tag? You want to align the data, not the code in the block?
- ir: related to the one above: block alignment doesn't translate well to variables in the block (the actual stuff that needs to be aligned in memory) but: need variable alignment tag instead of block alignment tag, really
- ir: fix call() return value handling
- ir: proper code gen for the CALLI instruction and that it (optionally) returns a word value that needs to be assigned to a reg
- ir: idea: (but LLVM IR simply keeps the variables, so not a good idea then?...): replace all scalar variables by an allocated register. Keep a table of the variable to register mapping (including the datatype) global initialization values are simply a list of LOAD instructions. Variables replaced include all subroutine parameters! So the only variables that remain as variables are arrays and strings.
- ir: add more optimizations in IRPeepholeOptimizer
- ir: the @split arrays are currently also split in _lsb/_msb arrays in the IR, and operations take multiple (byte) instructions that may lead to verbose and slow operation and machine code generation down the line. maybe another representation is needed once actual codegeneration is done from the IR...?
- ir: getting it in shape for code generation...
- [problematic due to using 64tass:] better support for building library programs, where unused .proc are NOT deleted from the assembly. Perhaps replace all uses of .proc/.pend/.endproc by .block/.bend will fix that with a compiler flag? But all library code written in asm uses .proc already..... (textual search/replace when writing the actual asm?) Once new codegen is written that is based on the IR, this point is mostly moot anyway as that will have its own dead code removal on the IR level.
- 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) But the V flag is also set on certain normal instructions
- Zig-like defer to execute a statement/anonymousscope at subroutine exit?
- generate WASM to eventually run prog8 on a browser canvas? Use binaryen toolkit and/or my binaryen kotlin library?
- implement split words arrays all()
- implement split words arrays sort()

Libraries:

- gfx2: add EOR mode support like in monogfx and see PAINT for inspiration. Self modifying code to keep it optimized?
- fix the problems in atari target, and flesh out its libraries.
- c128 target: make syslib more complete (missing kernal routines)?
- pet32 target: make syslib more complete (missing kernal routines)?
- VM: implement more diskio support

Optimizations:

- VariableAllocator: can we think of a smarter strategy for allocating variables into zeropage, rather than first-come-first-served? for instance, vars used inside loops first, then loopvars, then uwords used as pointers, then the rest
- various optimizers skip stuff if compTarget.name==VMTarget.NAME. Once 6502-codegen is done from IR code, those checks should probably be removed, or be made permanent
- optimizeCommonSubExpressions: currently only looks in expressions on a single line, could search across multiple expressions

STRUCTS again?

What if we were to re-introduce Structs in prog8? Some thoughts:

- can contain only numeric types (byte,word,float) - no nested structs, no reference types (strings, arrays) inside structs
- only as a reference type (uword pointer). This removes a lot of the problems related to introducing a variable length value type.
- arrays of struct is just an array of uword pointers. Can even be @split?
- need to introduce typed pointer datatype in prog8
- str is then syntactic sugar for pointer to character/byte?
- arrays are then syntactic sugar for pointer to byte/word/float?

Other language/syntax features to think about

- add (rom/ram)bank support to romsub. A call will then automatically switch banks, use callfar and something else when in banked ram. challenges: how to not make this too X16 specific? How does the compiler know what bank to switch (ram/rom)? How to make it performant when we want to (i.e. NOT have it use callfar/auto bank switching) ?

CHAPTER

SIX

INDEX

- genindex

INDEX

W