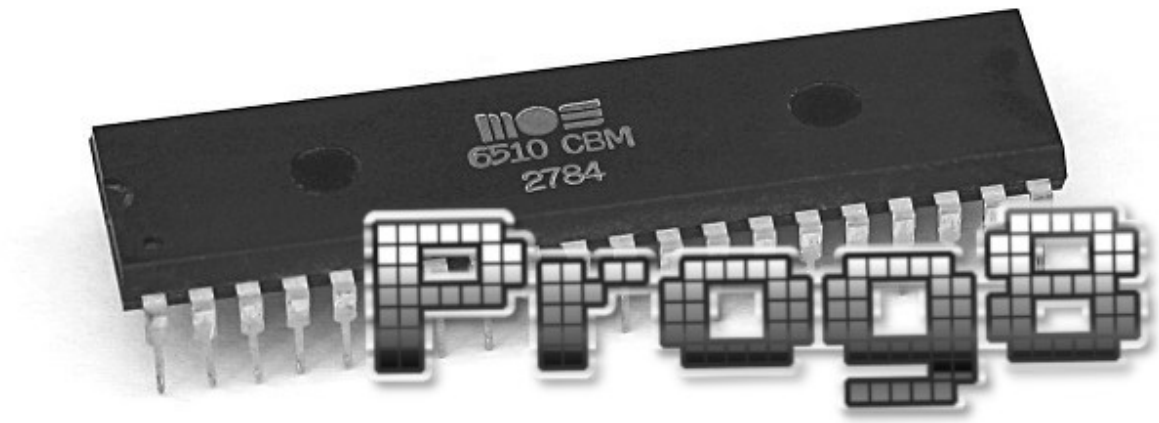
Prog8 Documentation

Release 12.3-SNAPSHOT

Irmen de Jong

Jun 19, 2026

CONTENTS OF THIS MANUAL:



Prog8 is a compiled programming language targeting the 8-bit 6502 CPU family. It 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.

Supported target systems are Commodores (C64, C128, PET), Commander X16, and others.

Get the compiler here *Getting the software*.

Open source Software License

Full source code is on github: https://github.com/irmen/prog8.git Prog8 is copyright © Irmen de Jong (irmen@razorvine.net | http://www.razorvine.net).

This is free software, as defined in the GNU GPL 3.0 (https://www.gnu.org/licenses/gpl.html) *Exception:* All output files generated by the compiler (intermediary files and compiled binary programs) are excluded from this particular license: you can do with those *whatever you want*. This means, for instance, that you can use the Prog8 compiler to create commercial software as long as you only sell *the actual resulting program*.

CHAPTER

ONE

WHAT IS PROG8?

This is a compiled programming language targeting the 8-bit 6502 CPU family. 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.

The targeted CPUs are the 6502 / 6510 / 65c02 microprocessors. They are from the late 1970's and early 1980's and were used in many home computers from that era, such as the Commodore 64.

You can compile programs for various machines that are built into the compiler:

- Commander X16 (with 65c02 cpu, 65816 cpu specifics are currently not supported by prog8 itself)
- Commodore 64
- Commodore 128 (limited support)
- Commodore PET (limited support)
- any other 65(C)02 target machine or setup can be configured to a great extent in a user written configuration file. There are some examples included for the Atari 800 XL, NEO6502, Foenix F256, and such.
- some users have been experimenting with a NES and a C64 OS target as well.

Some language features are mentioned below, and you can also read *Prog8 versus other languages* if you want to quickly read about how Prog8 compares to well-known other languages.

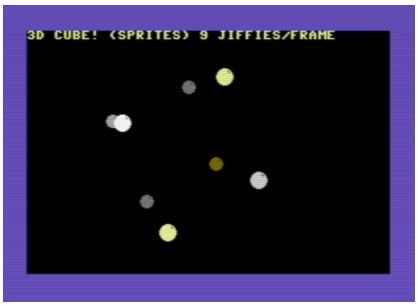
1.1 Community

Most of the development on Prog8 and the use of it is currently centered around the Commander X16 retro computer. However, the other compiler targets are also still worked on, most notably the C64 target where it all started with!

The X16 Discord server contains a small channel dedicated to Prog8. Besides that, you can use the issue tracker on github for questions or problems or contributions.

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 or https://paypal.me/irmendejong .


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.

1.3 Language 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.
- compiled code is very compact; it is much smaller and usually also runs faster than equivalent C code compiled with CC65
- 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 module source files, code blocks, and subroutines. No need for forward declarations.
- Can make vars and routines private, as a simple way to restrict access and hide implementation details
- Provides 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
- Variables are all allocated statically, no memory allocation overhead
- Variable data types include signed and unsigned bytes and words, long integers, floats, arrays, and strings.
- Structs and typed pointers
- 2D arrays (`matrix[row][col]` syntax)
- Tight control over Zeropage usage
- Programs can be restarted after exiting (i.e. run them multiple times without having to reload everything), due to automatic variable (re)initializations.
- Programs can be configured to execute in ROM
- Conditional branches for status flags that map 1:1 to processor branch instructions for optimal efficiency
- `when` statement to avoid if-else chains
- `on .. goto` statement for fast jump tables
- `in` expression for concise and efficient multi-value/containment test
- `defer` statement to help write concise and robust subroutine cleanup logic

- Several specialized built-in functions, such as `lsb`, `msb`, `min`, `max`, `rol`, `ror`
- Various powerful built-in libraries to do I/O, number conversions, graphics and more
- Floating point math is supported on most cbm-compatible compiler targets.
- Provides access to most Kernal ROM routines as external subroutine definitions you can call normally.
- Strings can contain escaped characters but also many symbols directly if they have a PETSCII equivalent, such as "♠♥♣♦π▚●○╳". 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)
- Automatic ROM/RAM bank switching on certain compiler targets when calling routines in other banks
- Identifiers can contain Unicode Letters, so `knäckebröd`, `приблизительно`, □□□□ and `π` are all valid identifiers.
- Subroutines can return more than one result value
- Advanced code optimizations to make the resulting program smaller and faster
- Supports the sixteen 'virtual'16-bit registers R0 to R15 as defined on the Commander X16. You can look at them as general purpose global variables. 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.
- 50 Kb of available program RAM size on the C64 by default; because Basic ROM is banked out altogether
- 41 Kb of available program RAM size on the C128 by default; because Basic ROM is banked out altogether
- 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 by just changing the compilation target flag.

1.3.1 Code example

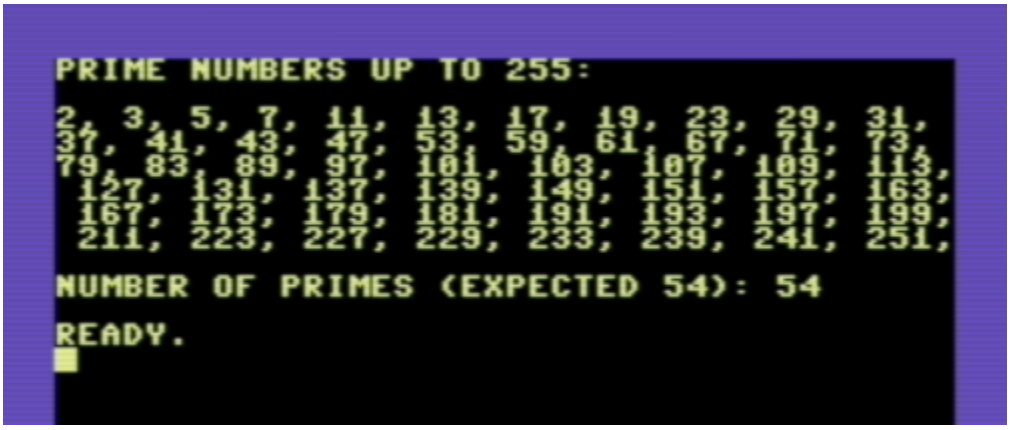
Here is a hello world program:

```
%import textio
%zeropage basicsafe

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

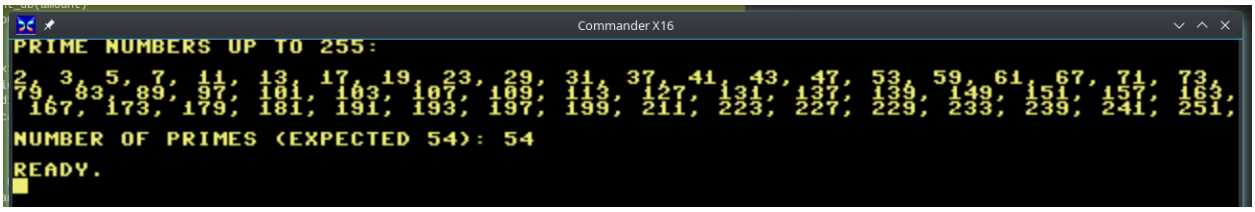
This code calculates prime numbers using the Sieve of Eratosthenes algorithm:

```
%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++
            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 and 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:



1.4 Getting the software

Usually you just download a fat jar of an official released compiler 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*. You can also read there what extra tools you need to get going.

You may look for 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. 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.

Language Server Protocol (LSP): Prog8 has a dedicated language server that provides IDE-like features for editors that support LSP (such as Kate, Vim, VS Code, and many others). The language server currently supports a few limited things such as document symbol overview and go to definition. Other features will be added in the future. The language server is located in the languageServer directory. To use it, configure your editor to launch `prog8-language-server` as the LSP server for `.p8` files.

CTags: for editors that support them, such as Vim, there is an effort by a member of the community to provide "ctags"files for Prog8. The basic purpose of ctags is to index the definitions from your source code and let you find them easily. It is not really to find each use of print for example, but to find where print is defined. Visit the Github repository for the files and detailed usage information.

1.5 AI Assisted Development

If you want to use an AI coding agent to help write Prog8 programs, load the `prog8-coder` skill (located in `.agents/skills/prog8-coder/`). It contains the full language reference – syntax rules, data types, standard library guidance, and common pitfalls.

If you need to write inline assembly (`%asm {{ }}` blocks or `asmsub` routines), also load the `asm6502-coder` skill (`.agents/skills/asm6502-coder/`) for 64tass assembler syntax, register conventions, and symbol prefixing rules.

CHAPTER

TWO

PROG8 VERSUS OTHER LANGUAGES

This chapter is meant for new Prog8 users coming with existing knowledge in another programming language such as C or Python. It discusses some key design aspects of Prog8 and how it differs from what you may know from those languages.

2.1 The language

- Prog8 is a structured imperative programming language. It looks like a mix of Python and C.
- It is meant to sit well above low level assembly code, but still allows that low level access to the system it runs on. Via language features, or even simply by using inline hand-written assembly code.
- **Prog8 is targeting very CPU and memory constrained 8-bit systems, this reflects many design choices to work within those limitations**
  (single digit Megahertz cpu clock speeds, and memory capacity counted in Kilobytes)
- Identifiers and string literals can contain non-ASCII characters so for example `knäckebröd` and □□□□ are valid identifiers.
- There's usually a single statement per line. There is no statement separator.
- Semicolon `;` is used to start a line comment. Multi-line comments are also possible by enclosing it all in `/*` and `*/`.
- Ternary operator `x ? value1 : value2` is available in the form of an *if-expression*: `if x [then] value1 else value2`
- There's a Swift/Zig/Go style `defer` statement for delayed cleanup is available in the subroutine scope.
- Qualified names are searched from within the top level namespace (so you have to provide the full qualified name). Unqualified names are locally scoped.
- A trailing comma is allowed optionally in array literals: [1,2,3,] is a valid array of values 1, 2 and 3.

2.2 No linker

- Even though your programs can consist of many separate module files, the compiler always outputs a single program file. There is no separate linker step. Currently, it's not easily possible to integrate object files created elsewhere. If the object file has a fixed

load location and fixed entrypoints, it can be loaded explicitly and accessed easily using extsub definitions though.

- The prog8 compiler is self-contained in a single jar file. You do need 1 external tool namely 64tass, which performs the assembler step.

2.3 Data types

- There are byte, word (16 bits), long (32 bits) and float datatypes for numbers.
- floats are available as native data type on most cbm-compatible systems.
- **There is no automatic type enlargement:** all calculations remain within the data type of the operands. Any overflow silently wraps or truncates. You'll have to add explicit casts to increase the size of the value if required. For example when adding two byte variables having values 100 and 200, the result won't be 300, because that doesn't fit in a byte. It will be 44. You'll have to cast one or both of the *operands* to a word type first if you want to accommodate the actual result value of 300. Similarly, `long v = w1 * w2` doesn't automatically give you the full 32 bits multiplication result, instead it is still constrained in the word range. If you need the full 32 bits result you'll have to call a specialized routine such as `math.mul32` or `math.mul16_last_upper()`.
- strings and arrays are allocated once, statically, and never resized.
- strings and arrays are mutable: you can change their contents, but always keep the original storage size in mind to avoid overwriting memory outside of the buffer.
- maximum string length is 255 characters + a trailing 0 byte.
- word arrays are split into 2 separate arrays by default (this is configurable): one for the LSBs and one for the MSBs of the words. This enables efficient 6502 instructions to access the words.
- maximum storage size for arrays is 256 bytes (512 for split word arrays) , the maximum number of elements in the array depends on the size of a single element value. you can use larger "arrays"via pointer indexing, see below at Pointers. One way of obtaining a piece of memory to store such an "array"is by using `memory()` builtin function.
- there is limited support for structs and typed pointers, see below at "Pointers and Structs".

2.4 Variables

- There is no dynamic memory management in the language; all variables are statically allocated. (but user written libraries are possible that provide that indirectly).
- Variables can be declared everywhere inside the code but all variable declarations in a subroutine are moved to the top of the subroutine. A for loop, or if/else blocks do not introduce a new scope. A subroutine (also nested ones) *do* introduce a new scope.
- All variables are initialized at the start of the program. There is no random garbage in them: they are zero or any other initialization value you provide.
- This also means you can run a Prog8 program multiple times without having to reload it from disk, unlike programs produced by most other compilers targeting these 8 bit platforms.

2.5 Subroutines

- Subroutines can be nested. Inner subroutines can directly access variables from their parent.
- Subroutine parameters are just local variables in the subroutine. (you can access them directly as such via their scoped name, if you want)
- There is no call stack for subroutine arguments: subroutine parameters are overwritten when called again. Thus recursion is not easily possible, but you can still do it with manual stack handling. There are a couple of example programs that show how to solve this in different ways, among which are fractal-tree.p8, maze.p8 and queens.p8
- There is no function overloading (except for a couple of builtin functions).
- Subroutines can return multiple return values, and you can multi-assign those in a single statement.
- Because every declared variable allocates some memory, it might be beneficial to share the same variables over different subroutines instead of defining the same sort of variables in every subroutine. This reduces the memory needed for variables. A convenient way to do this is by using nested subroutines - these can easily access the variables declared in their parent subroutine(s).
- Everything in prog8 is by default publicly accessible from everywhere else (via fully scoped names). Use the `private` keyword to hide symbols.
- Because there is no callstack for subroutine arguments, it becomes very easy to manipulate the return address that *does* get pushed on the stack by the cpu. With only a little bit of code it is possible to implement a simple cooperative multitasking system that runs multiple tasks simultaneously. See the “multitasking”example, which uses the “coroutines”library. Each task is a subroutine and it simply has its state stored in the statically allocated variables so it can resume after yielding, without doing anything special.

2.6 Pointers and Structs

Legacy ‘untyped’pointers:

- In Prog8 versions **before 12.0** there was no support for typed pointers, only ‘untyped’ ones: Variables of the `uword` datatype can be used as a pointer to one of the possible 65536 memory locations, so the value it points to is always a single byte. This is similar to `uint8_t*` from C. You have to deal with the uword manually if the object it points to is something different.
- Note that there is the `peekw` builtin function that *does* allow you to directly obtain the *word* value at the given memory location. So if you use this, you can use uword pointers as pointers to word values without much hassle.
- “dereferencing”a uword pointer is done via array indexing `ptr[index]` (where index value can be 0-65535!) or via the memory read operator `@(ptr)`, or `peek/peekw(ptr)`.
- Pointers don’t have to be a variable, you can immediately access the value of a given memory location using `@($d020)` for instance. Reading is done by assigning it to a variable, writing is done by just assigning the new value to it.

Typed pointers and structs:

- Since **version 12.0**, prog8 supports struct types and typed pointers.

- Structs are a grouping of one or more fields, that together make up the struct type.
- Typed pointers are just that: a pointer to a specific type (which can be a simple type such as float, or a struct type.)

2.7 Foreign function interface (external/ROM calls)

- You can use the `extsub` keyword to define the call signature of foreign functions (ROM routines or external routines elsewhere in RAM) in a natural way. Calling those generates code that is as efficient or even more efficient as calling regular subroutines. No additional stubs are needed.
- High level support of memory banking: an `extsub` can be defined with the memory bank number (constant or variable) where the routine is located in, and then when you call it as usual, the compiler takes care of the required bank switching.

2.8 Optimizations

- Prog8 contains many compiler optimizations to generate efficient code, but also lacks many optimizations that modern compilers do have. While empirical evidence shows that Prog8 generates more efficient code than some C compilers that also target the same 8 bit systems, the optimizations it makes on your code aren't super sophisticated.
- For time critical code, it may be worth it to inspect the generated assembly code to see if you can write things differently to help the compiler generate more efficient code (or even replace it with hand written inline assembly altogether). For example, if you repeat an expression multiple times it will be evaluated every time, so maybe you should store it in a variable instead and reuse that variable:

```
if board[i+1]==col or board[i+1]-j==col-row or board[i+1]+j==col+row {
    ...do something...
}

; more efficiently written as:

ubyte boardvalue = board[i+1]
if boardvalue==col or boardvalue-j==col-row or boardvalue+j==col+row {
    ...do something...
}
```

CHAPTER

THREE

COMPILING A PROGRAM

3.1 First, getting a working compiler

Before you can compile Prog8 programs, you'll have to download or build the compiler itself. Then 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 "prog8c-all.jar") from the releases on Github
2. run the compiler with "java -jar prog8c.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 (takes care of dependencies for you):

Arch Linux:
Currently, it's available on AUR for Arch Linux and derivative systems. The package is called "prog8". There should be no need to install anything else as it can automatically pull in the required dependencies.

This package, alongside the compiler itself, also globally installs syntax highlighting for `vim` and `nano`. In order to run compiler, you can type `prog8c`. The usage of those commands is exactly 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.

Mac OS (and Linux, and WSL2 on Windows):
Prog8 can be installed via Homebrew using the command `brew install prog8`. It will make the `prog8c` command available and also installs the other required software tools for you. While Homebrew works on Linux, it's probably best to first check your distribution's native package manager.

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 prog8c.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. You will also need at least Java version 17 or higher to build it. 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/prog8c`

`./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/prog8c-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 successful completion, you can start the compiler with:

```
./compiler/build/install/prog8c/bin/prog8c <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 17 or newer). If you do have trouble building or running the compiler on your operating system, please let me know!

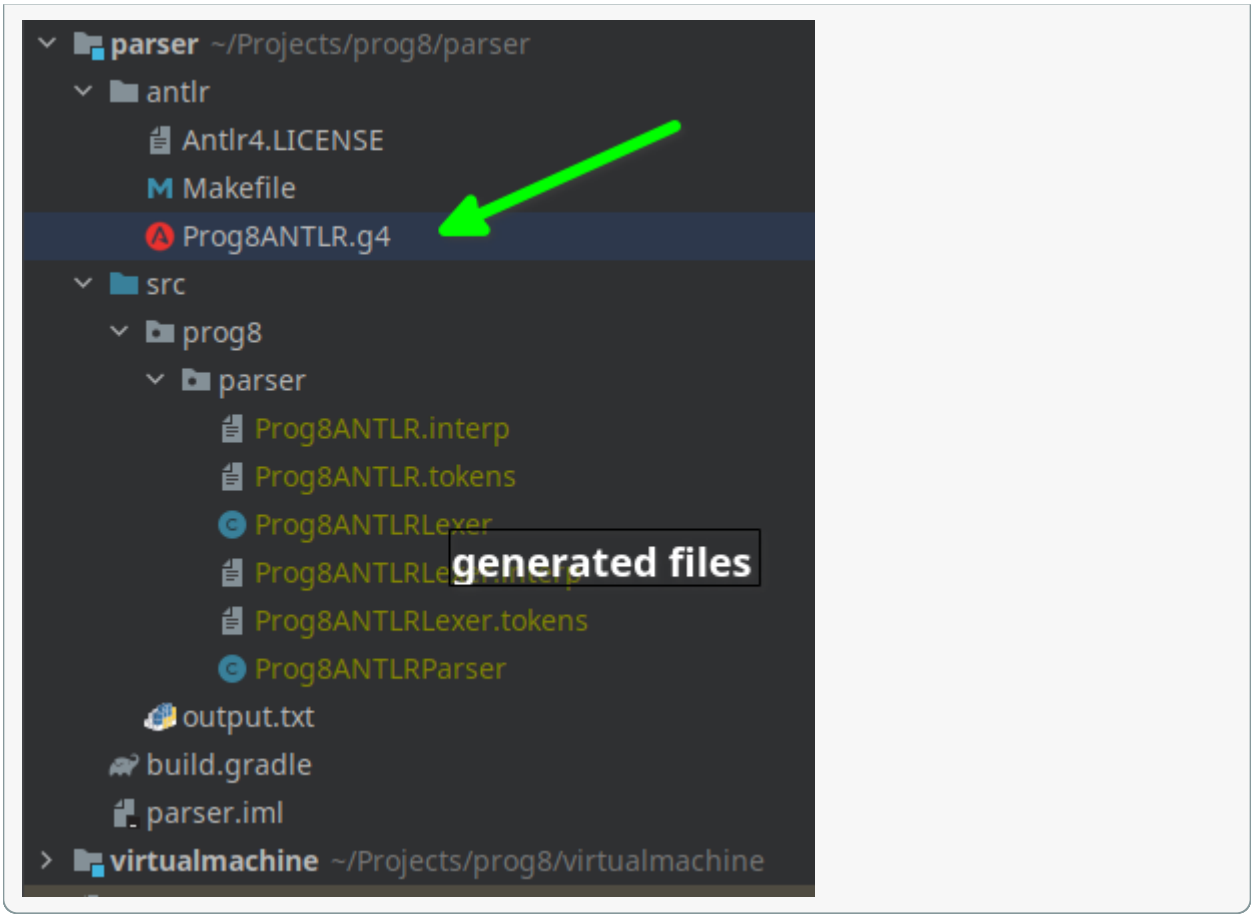
To successfully build and debug in IDEA, you have to do two things manually first:

1. you have to generate the buildversion file, do this with the shell command: `gradle createVersionFile`
2. manually generate the Antlr-parser classes first.

The easiest way to build the parser classes 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.



3.2 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 17 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.

3.3 Running the compiler

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 prog8c.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 always 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.

3.3.1 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.

`-asmlist`
Also generate an assembler listing file <program>.list

`-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 trigger a breakpoint halt in the debugger when this is enabled by running the emulator with -debug.

`-bytes2float <bytes>`
convert a comma separated list of bytes from the specified target system to a float value. Also see -float2bytes

`-check`
Quickly check the program for errors. No actual compilation will be performed.

`-compareir <baseline.p8ir>`
Compare the generated IR file with a baseline IR file. Shows a summary of differences including instruction count, chunk count, register usage, and file size. Also displays the first 10 instruction differences to help identify what changed. Useful for comparing optimized vs non-optimized builds, or tracking changes between compiler versions.

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

`-daemon`
Use the prog8c compilation daemon (auto-starts it if not running). This keeps the compiler running as a background server process, which greatly speeds up subsequent compilations by eliminating JVM startup overhead and allowing reuse of cached data. The daemon listens on a Unix domain socket and automatically shuts down after a period of inactivity. Unlike `-watch`, the daemon does not monitor files for changes: you still invoke `prog8c -daemon` each time you want to compile. The speedup comes from the warm JVM, not from automatic recompilation. Because each invocation is an individual compiler run, you can compile different source files or pass different options on each call —the daemon does not lock you into a single project or set of flags.

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

-dumpvars
print a dump of the variables in the program

-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)

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

-float2bytes <number>
convert floating point number to a list of bytes for the specified target system. Also see -bytes2float

-ignorefootguns
Don't print warnings for 'footgun'issues. Footgun issues are certain things you can do in Prog8 that may make your program blow up unexpectedly, for instance uncareful use of dirty variables, or reusing the R0-R15 registers for subroutine parameters. With this option you're basically saying: "Yes, I know I am treading on mighty thin ice here, but I don't want to be reminded about that".

-libsearch pattern
Search in the embedded library files for occurrences of the given regex pattern. If no exact matches are found, the compiler automatically attempts a "fuzzy"search by inserting wildcards between the characters of the search pattern.

-libdump path
Extract all embedded library files into the given output directory. Note: the library source code belongs to the Prog8 project and is licensed under the GPL 3.0 software license.

-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.

-nostdlib
Disable loading of the standard library. Specifically, this disables searching in the **Target Library Directories** and the **Internal Standard Library** in the *module search path*. The regular filesystem search in the current directory, `-srcdirs`, and the neighboring directory of the importing file still occurs. Note that core library modules (`syslib`, `prog8_math`, `prog8_lib`) are always loaded regardless of this option, so you must provide these in your own source paths if you use this flag.

-out <directory>
Sets directory location for output files instead of current directory. Creates it if it doesn' t exist yet.

-plaintext
Prints output messages in plain text: no colors or fancy symbols.

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

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

-quiet
Don't print compiler and assembler messages.

-quietasm
Don't print assembler messages

-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.

-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.

-nosourcelines
Do not include the original prog8 source code lines as comments in the generated assembly code file, mixed in between the actual generated assembly code. The default behavior is to include the source lines.

-srcdirs <pathlist>
Specify a list of extra paths (separated with the system path separator, ':'on Linux/macOS, ';'on Windows), to search in for imported modules. These directories are prepended to the module search path, meaning they have the highest priority and will be searched before the neighboring directory, the current directory, and the standard library. Useful if you have library modules somewhere that you want to re-use, or to switch implementations of certain routines (performing a "complete overlay") via a command line switch.

-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, `virtual` = builtin virtual machine. You can also specify a file name as target, prog8 will then try to read the target machine's configuration and properties from that configuration file instead of using one of the built-in targets. See *Customizable targets* for details about this.

-timings
Show a more detailed breakdown of the time taken in various compiler phases, for performance analysis of the compiler itself.

-varsgolden
Like `-varshigh`, but places the variables in the "golden ram"area instead (e.g. $0400-$07FF on CX16, $1300-$1BDF on C128). Because this is in normal system memory, there are no bank switching issues. This mode is only available on the Commander X16 and the Commodore 128, and possibly on custom configured targets.

-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 allotted 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.

-version
Just print the compiler version and copyright message, and exit.

-vm
load and run a 'p8ir'intermediate representation file in the internal VirtualMachine instead of compiling a prog8 program file.

-vmtrace
Enable instruction-by-instruction tracing when running the virtual machine (use with `-vm` or `-emu` on the virtual target). Prints each executed IR instruction with its location (chunk name and instruction index). Useful for debugging program behavior and understanding control flow in the IR code.

-traceimports
Prints a detailed trace of every module that the compiler is importing and loading. It shows if the module is loaded from a file or from an internal resource, the exact path, and which import caused the module to be loaded. It also shows the search location type (current directory, neighboring directory, configured in `-srcdirs`, etc).

-warnimplicitcasts
Give warnings for lines where a silent (implicit) type cast is done from a smaller to a larger type. Example: when a byte is assigned to a word variable. This may indicate a potential value evaluation issue because unlike most other programming languages, Prog8 doesn't do automatic type enlargement for expressions. This means that for example if a=50 and b=20, a times b is *not* equal to 1000 if a and b are bytes. Only if one or both of them are explicitly casted to a word type, the calculation will result in the word value 1000. If you have code like this: *uword result = a * b* the compiler silently converted the *byte* result to a *word* variable, but maybe you expected the result to actually be 1000 here (and forgot to add a cast in the expression to make it words)

-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)

-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. *Combined with the Emulator's Host-FS or real*

hardware Calypso's network drive: this enables you to edit prog8 source files with an editor on the X16 itself (such as XVI, XEdit). When saving the source file, the compiler can sit in the background in watch mode, and recompile it for you immediately on save. Then you can directly run the new resulting PRG file in the X16! Unlike `-daemon`, the watch mode keeps the process in the foreground and automatically recompiles on file changes, whereas `-daemon` is a background server that you invoke explicitly each time you want to compile.

3.4 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. They are embedded into the compiler so you don't have to worry about where they are, but their names are generally reserved for the standard library (although you can override them if necessary, see below).

3.4.1 Importing other source files and specifying search location(s)

You can create multiple source files yourself to modularize your programs. You can also create "library"modules with handy routines that can be shared among programs. By importing those module files, you can use them in other modules.

When the compiler encounters an `%import mymodule` directive, it searches for a file named `mymodule.p8` in the following locations (in this exact order):

- **User Source Directories**: Any directories specified with the `-srcdirs` command-line option, in the order they were provided.
- **Neighboring Directory**: The folder containing the source file that contains the `%import` directive.
- **Current Working Directory**: The directory where the compiler was started from (`.`).
- **Target Library Directories**: Target-specific library paths on the filesystem (only used by some targets or custom target configurations).
- **Internal Standard Library**: Built-in modules bundled with the compiler (embedded as internal resources).

> **Note**
>
> The `-nostdlib` command-line option disables searching in both the **Target Library Directories** and the **Internal Standard Library**, allowing for a complete replacement of the standard library while still allowing imports from the filesystem.

This search order allows you to override standard library modules or neighbor modules by placing a version with the same name in a higher priority location. This can also be used as a "lo-fi"way to provide different source files for different compilation targets, which is useful because the compiler currently lacks conditional compilation like `#ifdef`/`#endif` in C.

3.5 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.

3.6 Troubleshooting

3.6.1 Compilation errors or warnings

When there are warnings or errors during compilation of your program, no output files will be produced, and the error messages will be printed on the screen like this:

```
ERROR file:///home/user/code/test.p8:13:9: undefined symbol: zzz
```

The severity is followed by the file and location in that source file where the error occurred (editors and IDEs that show the compiler output usually make these links clickable). The two numbers are the line number and column in the file. There are a couple of message severities:

INFO
: informational messages that can be ignored if you want

WARNING
: things that may be a problem depending on circumstances. If there are no errors, a compiled program is still produced, but it is a good idea to investigate the warnings that are printed.

ERROR
: unrecoverable problem that prevented successful compilation

3.6.2 Compiler doesn't run, complains about "UnsupportedClassVersionError"

You need to install and use JDK version 17 or newer to run the prog8 compiler. Check this with "java -version". See *Required additional tools*.

3.6.3 The computer resets after running my 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.

3.6.4 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*.

3.6.5 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`.

3.6.6 Gradle error when building the compiler yourself

If you get a gradle build error containing the line "No matching toolchains found for requested specification"or "Gradle requires JVM 17 or later to run", it means that the Gradle build tool can't locate the correct version of the JDK to use. You will need a Java JDK version 17 or higher to build and run the compiler.

3.6.7 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.

3.6.8 'shadowing'warnings from 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.

3.6.9 Program loads but crashes immediately on startup

Assuming there are no programming errors in the code and the compiler has no code generation bugs, there can be various factors that cause this:

1. you are using `%option no_sysinit` but the program might depend on proper initialization of the system before it can run. Try removing this directive to see if it helps.
2. the `main` block is too large for default system RAM, causing the vital program startup routines that Prog8 requires to move outside of the default RAM area, making them inaccessible. For example on the C128 this can occur quite quickly because in the default memory configuration, the BASIC ROM already appears at $4000, leaving *very* little space for your program code by default. Prog8 attempts to reconfigure the memory but it can't because the routine doing that is not accessible anymore and the program will jump to random code, crashing the system. Make sure the startup logic appears "soon enough"in your program - if in doubt, check the generated assembly file listing to see how large the `main` block is and where the startup routines are placed. If this is in fact the problem, you need to modularize the code and move stuff out of the `main` block and into their own block(s).

3.7 Examples

A bunch of example programs can be found in the 'examples'directory of the source tree on Github. (Or download the source archive from there to get all the files at once).

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 prog8c.jar -target c64 -emu examples/c64/rasterbars.p8
```

or:

```
$ /path/to/prog8c -target c64 -emu examples/c64/rasterbars.p8
```

CHAPTER

FOUR

PROGRAMMING IN PROG8

This chapter describes a high level overview of the elements that make up a program.

4.1 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. 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.

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. 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. See the syntax reference for all directives. The list of directives is given below at *Directives*.

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'). They are described in the chapter *Variables and Values*.

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. A label is an identifier followed by a colon `:`. It's ok to put the next statement on the same line, immediately after the label. You can jump to it with a goto statement. It is also possible to use a subroutine call to a label (but without parameters and return value), however □□ footgun warning: doing that is tricky because it makes for weird control flow and interferes with defers.

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.

4.2 Variables

How to declare variables in prog8 is explained in a separate chapter *Variables and Values*.

4.3 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
```

(continues on next page)

(continued from previous page)

```
COUNTER
Better_Name_2
something_strange__
knäckebröd
приблизительно
π
```

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
```

Aliases

The `alias` statement makes it easier to refer to symbols from other places, and they can save you from having to type the fully scoped name everytime you need to access that symbol. Aliases can be created in any scope except at the module level. An alias is created with `alias <name> = <target>` and then you can use `<name>` as if it were `<target>`. It is possible to alias variables, labels and subroutines, and even whole blocks and words The name has to be an unscoped identifier name, the target can be any scoped or unscoped symbol. Please consider using aliases sparingly because it may lead to confusing code if you alias well-known block names for example.

4.4 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*:

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

The <blockname> must be a valid identifier, and must be unique in the entire program (there's a directive to merge multiple occurrences). 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.

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". You can use the `private` keyword to hide symbols from other blocks - see *Private symbols*. 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

languages, 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).

:index:`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.

:index:`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
    }
}
```

:index:`Aliases` make it easier to refer to symbols from other places. They save you from having to type the fully scoped name everytime you need to access that symbol. Aliases can be created in any scope except at the module level. You can create and use an alias with the `alias` statement like so:

```
alias  score   = cx16.r7L        ; 'name' the virtual register
alias  prn     = txt.print_ub    ; shorter name for a subroutine elsewhere
...
prn(score)
```

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.

4.5 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` block 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. **It is advised to not put too much code and data into the main block**, because that may cause Prog8's vital program initialization routines to be placed outside of normal accessible program RAM, causing an immediate program crash on startup. If this happens you need to modularize the code and move stuff into their own block(s).

4.6 Directives

`%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.

`%align <interval>`

Level: not at module scope. Tells the assembler to continue assembling on the given alignment interval. For example, `%align $100` will insert an assembler command to align on the next page boundary. Note that this has no impact on variables following this directive! Prog8 reallocates all variables using different rules. If you want to align a specific variable (array or string), you should use one of the alignment tags for variable declarations instead. Valid intervals are from 2 to 65536. **Warning:** if you use this directive in between normal statements, it will disrupt the output of the machine code instructions by making gaps between them, this will probably crash the program!

`%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*.

`%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! or %breakpoint

Level: not at module scope. Defines a debugging breakpoint at this location. See *Debugging (with VICE or Box16)* The version with the exclamation point '!'at the end can be used even if the breakpoint follows an expression. If you don't use the '!'version in this case the compiler may think it is just a term in the expression (modulo operator and breakpoint operand value), instead of a breakpoint directive:

```
a = b
%breakpoint       ; parse error because it thinks it is part of the␣
↪previous line

a = b
%breakpoint!      ; parsed correctly as directive
```

%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 *Strings*.

%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. For more details on where the compiler searches for imported files, see *Importing other source files and specifying search location(s)*.

%jmptable (lib.routine1, lib.routine2, ...)

Level: block. This builds a compact "jump table"meant to be used in libraries. You can put the elements of the table on different lines if you wish. It outputs a sequence of JMP machine code instructions jumping to each of the given subroutines in the jmptable list in order. This way the routines in the library can be accessed using a neat fixed list of offsets at the beginning of the library code, and the actual implementation of those routines can be changed in later versions of the library without existing callers noticing anything.

This is usually put at the top of the main block so that it ends up at the beginning of the library file. *Note:* the compiler will still insert the required bootstrapping code in front of it, which in the case of a library, is the single JMP to the start routine which also does some variable initialization and BSS area clearing. So the first JMP in the jumptable list will actually end up at offset 3 in the resulting binary program. The `jmp start` instruction that prog8 inserts ends up as the implicit first entry of the actual jump table instructions list that is put into the resulting library program:

```
jmp  start              ; first program instruction always generated by␣
↪prog8
jmp  lib.routine1       ; jump table first entry
jmp  lib.routine2       ; jump table second entry
...
```

%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

%memtop <address>

Level: module. Global setting, changes the program's top memory address. This is usually specified internally by the compiler target, but with this you can change it to another value. This can be useful for example to 'reserve'a piece of memory at the end of program space where other data such as external library files can be loaded into. This memtop value is used for a check instruction for the assembler to see if the resulting program size exceeds the given memtop address. This value is exclusive, so $a000 means that $a000 is the first address that program can no longer use. Everything up to and including $9fff is still usable.

%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, setting memory bank configuration 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)
- `merge` (in a block) will merge this block's contents into an already existing block with the same name. Can be used to add or override subroutines to an existing library block, for instance. Overriding (monkeypatching) happens only if the signature of the subroutine exactly matches the original subroutine, including the exact names and types of the parameters. Where blocks with this option are merged into is intricate: it looks for the first other block with the same name that does not have %option merge, if that can't be found, select the first occurrence regardless. If no other blocks are found, no merge is done. Blocks in libraries are considered first to merge into.
- `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. Various standard library modules use it for this purpose.
- `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()`.
- `romable` (module) *WORK-IN-PROGRESS/EXPERIMENTAL* make sure that the generated code is suitable for running in ROM (so no self-modifying code and such, which is normally used to generate smaller/more optimized code) See *ROM-able programs* for more details.

`%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)
- type `xex` : Atari xex program
- type `library` : loadable library file. See *Binary Loadable Libraries*.

`%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 for a few addresses that are used by the system's IRQ handler. Even though that default IRQ handler 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.

`%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).

`%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.

4.7 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.

Optimization

Usually a loop in descending order downto 0 or 1, produces more efficient assembly code than the same loop in ascending order.

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)

4.7.1 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 {
    ; do something with number...
    break       ; break out of the loop early
}
```

See *Ranges* for all of the details.

4.7.2 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
}
```

4.7.3 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
```

(continues on next page)

(continued from previous page)

```
        continue    ; immediately next iteration
} until  <condition>
```

4.7.4 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.

4.7.5 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.

4.8 Conditional Execution

4.8.1 if statement

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
```

(continues on next page)

(continued from previous page)

```
    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.

4.8.2 if expression

Similar to the if statement, but this time selects one of two possible values as the outcome of the expression, depending on the condition. You write it as `if <condition> [then] <value1> else <value2>` (the then keyword is optional) and it can be used anywhere an expression is used to assign or pass a value. The first value will be used if the condition is true, otherwise the second value is used. Sometimes it may be more legible if you surround the condition expression with parentheses so it is better separated visually from the first value following it. You must always provide two alternatives to choose from, and they can only be values (expressions). An example, to select the number of cards to use depending on what game is played:

```
ubyte numcards = if game_is_piquet  32 else 52

; it's more verbose with an if statement:
ubyte numcards
if game_is_piquet
    numcards = 32
else
    numcards = 52
```

The expression form is also available for the conditionals `if_cc`, `if_cs`, `if_z` etc. (These particular variants for checking the value of the Carry status bit actually compile into very efficient branchless assembly code):

```
ubyte carryvalue = if_cs 1 else 0
```

Optional ``then``: For readability reasons it is allowed to put a `then` keyword between the condition expression and the first value expression. In the example above the separation between the two is still kinda obvious, but readability may be less clear with cases like the following where the value and the last part of the condition are both numeric:

```
if value<999 888 else 777              ; slightly hard to read
if value<999 then 888 else 777         ; separation is clearer
```

4.8.3 on .. goto / on .. call statement (jump table)

The `on goto / call` statement is suitable to create a fast call of a subroutine from a list based on an index value. it selects a function to jump to in O(1) whereas a similar when-statement, runs in O(n) because that one checks each index value. The `on goto / call` instead simply gets the correct entry from an array of function pointers and jumps to it directly. The index value that is used is 0-based; 0 will jump to the first entry in the list, 1 to the second, and so on. If the value is too large (i.e. outside the list of functions), no call is performed, and execution continues. In this case you can optionally add an `else` block that is then executed instead. Here's an example:

```
on math.randrange(5) call (
        routines.func1,
        routines.func2,
        routines.func3 )
    else {
        txt.print("no call was done")
    }

on math.randrange(5) goto (routines.func1, routines.func2, routines.func3)
txt.print("no jump was taken")
```

4.8.4 when statement

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")
    50 to 60 step 2 -> txt.print("fifty to sixty, even")
    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. You can explicitly put a list of numbers that all should result in the same case, or even use any *range expression* as long as it denotes a constant list of numbers. Be aware that every number is compared individually so using long lists of numbers and/or many choice cases will result in poor performance. If you need to call a certain function based on some sequential index number, look at the `on .. goto / call` statement instead.

Choices can result in a single statement or a block of multiple statements in which case you have to use { } to enclose them.

The else part is optional.

> **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.

4.9 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
goto  address + idx   ; jump to an address that is the result of an expression
```

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

4.10 Assignments

Assignment statements assign a 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`).

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.

It is also possible to do a multi-value assignment: `x, y, z = 11, 22, 33`. This is just a shorter way to write the three separate assignments x=11 followed by y=22 followed by z=33. For subroutines that return multiple values, you have to write such a multi-value assignment statement as well, to assigns those multiple values. 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.

4.11 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.

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.

4.11.1 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 (unless it's a constant): 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:

```
; for the sake of this example: make sure this is not optimized away as a␣
↪constant
byte @shared 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  (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:

```
; for the sake of this example: make sure this is not optimized away as a␣
↪constant
byte @shared b = 44
w = (b as word)*55
w = b*(55 as word)
w = b * $0037
```

4.12 Operators

The following table lists the operator precedence in Prog8, from highest to lowest. Operators on the same line have the same precedence.

Table 1: Operator Precedence

Precedence	Operator(s)	Description
1 (highest)	`( )`, `sizeof`, `sub(args)`	Parentheses, size of, subroutine call
2	`.`	Member access / scope traversal
3	`+`, `-`, `~` (unary)	Unary plus, unary minus, bitwise NOT
4	`*`, `/`, `%`	Multiplication, division, remainder
5	`+`, `-` (binary)	Addition, subtraction
6	`<<`, `>>`	Bitwise shifts
7	`&`	Bitwise AND
8	`^`	Bitwise XOR
9	`|`	Bitwise OR
10	`<`, `>`, `<=`, `>=`	Comparisons
11	`==`, `!=`	Equality and inequality
12	`to`, `downto`	Range creation
13	`in`, `not in`	Containment check
14	`not`	Logical NOT
15	`and`	Logical AND
16	`or`	Logical OR
17	`xor`	Logical XOR
18	`as`	Type cast
19	`if ... then ... else`	If-expression
20 (lowest)	literals, identifiers, etc.	Primary expressions (literals, variables, indexing, etc)

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) While the operands can be signed integers (the expression will just consider the underlying bit patterns), the result value of a bitwise expression is always unsigned.

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 *Ranges* 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"
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`). This is explained in the *Variables and Values* chapter.

&< and &> are for use on split word arrays, they give you the address of the LSB byte array and MSB byte array separately, respectively. Note that &< is just the same as & in this case. For more details on split word arrays, see *Arrays*.

Typed pointer version: the single & operator still returns an untyped uword address for backward compatibility reasons, so existing programs keep working. The *double ampersand* && operator however returns a *typed* pointer to the value. The semantics are slightly different because adding or subtracting a number from a typed pointer uses *pointer arithmetic* that takes the size of the value that it points to into account.

type cast: `as`
Explicitly convert an expression to another data type. Note that `as` has very low precedence, lower than most other operators. This means that `a + b as long` is parsed as (a

`+ b) as long`. If you want to cast an operand before an operation, use parentheses: `(a as long) * b`.

ternary:
Prog8 doesn't have a ternary operator to choose one of two values (`x? y : z` in many other languages) instead it provides this feature in the form of an *if expression*. See below under "Conditional Execution".

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)`

4.13 Subroutines

4.13.1 Defining a subroutine

You define a subroutine like so:

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

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

The parameters is a (possibly empty) comma separated list of "<datatype> <parameter-name>"pairs specifying the input parameters. The return type has to be specified if the subroutine returns a value.

Subroutines can be defined in a Block, but also nested inside another subroutine. Everything is scoped accordingly. There are three different types of subroutines: regular subroutines (the one above), assembly-only, and external subroutines. These last two are described in detail below.

4.13.2 Private symbols

You can use the `private` keyword to hide a symbol from outside its block. Accessing a private symbol from another block results in a compilation error.

The `private` keyword can be applied to the following declarations:

- **subroutines** (the keyword must come before `inline` if used):

```
private sub helper() {
    ; only callable from within this block
}

private inline sub fast_helper() {
    ; private and inlined - note: private comes BEFORE inline
```

(continues on next page)

(continued from previous page)

```
}

private asmsub asm_helper(ubyte a @A) clobbers(Y) -> ubyte @A {
    ; private asmsub
}
```

- **external subroutines** (`extsub`):

```
private extsub $FFD2 = CHROUT(ubyte char @A)
```

- **structs**:

```
private struct Node {
    byte x
    byte y
}
```

- **enums**:

```
private enum Color { red, green, blue }
```

- **aliases**:

```
private alias MyReg = cx16.r0
```

Reusing *virtual registers* R0-R15 for parameters

Footgun warning

when using this the program can clobber the contents of R0-R15 when doing other operations that also use these registers, or when calling other routines because Prog8 doesn't have a callstack. Be very aware of what you are doing, the compiler can't guarantee correct values by itself anymore.

Normally, every subroutine parameter will get its own local variable in the subroutine where the argument value will be stored when the subroutine is called. In certain situations, this may lead to many variables being allocated. You *can* tell the compiler to not allocate a new variable, but instead to reuse one of the *virtual registers* R0-R15 (accessible in the code as `cx16.r0` - `cx16.r15`) for the parameter. This is done by adding a `@Rx` tag to the parameter. This can only be done for booleans, byte, and word types. Note: the R0-R15 *virtual registers* are described in more detail below for the Assembly subroutines. Here's an example that reuses the R0 and the R1L (lower byte of R1) virtual registers for the parameters:

```
sub  get_indexed_byte(uword pointer @R0, ubyte index @R1) -> ubyte {
    return @(cx16.r0 + cx16.r1L)
}
```

4.13.3 Assembly-Subroutines

These are 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). Such subroutines 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 can only consist of just inline assembly.

The `@ <register>` part is required for rom and assembly-subroutines, as it specifies the cpu registers that take the 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).

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! Inlining may increase code size significantly and can only be used in limited scenarios

Note

The **sixteen '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 other compilation targets!* (however they might not be as efficient there as on the X16, because on most other targets such as the C64, these registers are not placed in zeropage due to lack of space) In both regular **prog8** *and* **assembly** code these 'registers'are directly accessible too via:

- `cx16.r0` - `cx16.r15` (memory-mapped **uword** values, most often these are used)
- `cx16.r0s` - `cx16.r15s` (memory-mapped **word** values, used when you need a signed word)
- `cx16.r0r1sl` - `cx16.r14r15sl` (memory-mapped **long** values in 2 consecutive virtual registers, used when you need a signed long)
- `cx16.r0H`, `cx16.r0L` (for each r0..r15; memory-mapped **ubyte** values, both bytes of the register)

- `cx16.r0sH`, `cx16.r0sL` (for each r0..r15; memory-mapped **byte** values, both bytes of the register)
- `cx16.r0bH`, `cx16.r0bL` (for each r0..r15; memory-mapped **bool** values, both bytes of the register)

You can use them directly but their name isn't very descriptive, so it may be useful to define an alias for them when using them regularly.

Note

Dealing with **long** arguments and return values: A long takes 4 bytes (or 2 words, if you will). *There is no register definition specific to long types*. The way you specify the 'register'for a long argument or return value for an asmsub is by using a *virtual register pair*. For example, you can use R0+R1, R2+R3, R4+R5 and so on to take a long value instead. The syntax to use as a 'register'name for those pairs is `R0R1`, `R2R3`, `R4R5` and so on.

Caution

Virtual register clobbering by compiler operations:

Various compiler operations and builtin routines use the virtual registers R0-R15 as temporary storage or for placing return values. If you are using the virtual registers directly in your program (or via `@R0..@R15` parameter annotations), be aware of the fact that they will not always preserve their value!

Register(s)	When clobbered
`R0-R3`	Many library routines use these as scratch
`R12-R1`	Several long value operations and arithmetic expressions
`R14-R1`	Several long value operations / word '%', '/', divmod
`R0..R15`	if specified: multi-value subroutine return values, parameter reuse. several library routines may clobber one or more of these registers too.

Additionally, on all targets, the virtual registers R0-R15 are **not preserved** across IRQ handler calls. Use `cx16.save_virtual_registers()` / `cx16.restore_virtual_registers()` if your IRQ handler needs to use them.

4.13.4 External subroutines

These define an external subroutine that's implemented outside of the program (for instance, a ROM routine, or a routine in a library loaded elsewhere in RAM). External subroutines are usually defined by compiler library files, with the following syntax:

```
extsub $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 tell what CPU registers are clobbered by the call instead of being unchanged or returning a meaningful result value. This register clobber information currently is only for documentation purposes.

Note that the address ($ffd5 in the example above) can actually be an expression as long as it is a compile time constant. This can make it easier to define jump tables for example, like this:

```
const uword APIBASE = $8000
extsub APIBASE+0 = firstroutine()
extsub APIBASE+3 = secondroutine()
extsub APIBASE+6 = thirdroutine()
```

Banks: it is possible to declare a non-standard ROM or RAM bank that the routine is living in, with `@bank` like this: `extsub @bank 10 $C09F = audio_init()` to define a routine at $C09F in bank 10. You can also specify a variable or even the name of a subroutine (must be parameterless, returning a ubyte) for the bank. See *ROM/RAM bank selection* for more information.

4.13.5 Calling a subroutine

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.

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* (routines calling themselves) without doing some manual work to save and restore the variables that need to retain their value between calls. If you really need a recursive algorithm, there are a few options:

- hand code it in embedded assembly
- rewrite it as an iterative algorithm if possible

- use manual stack handling of the variables that need to retain their values, perhaps using a manual stack or using *push()* and *pop()*

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.

4.13.6 Multiple return values

Subroutines can return more than one value. `asmsub` and `extsub` routines return their multiple values spread across different registers, and can also efficiently use the CPU's status register flags for boolean returnvalues. You have to "multi assign"all return values of the subroutine call to something: 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. Remember that you can use `void` to skip a value. 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 { ... }
```

register usage

Subroutines with multiple return values use cpu registers A, Y, and the R0-R15 "virtual registers"to return those, depending on the number of values returned. A floating point value is passed via the FAC 'register'. Multiple float return values are supported on the virtual target, but limited to a single float on 6502 targets (because the ROM float routines use FAC1/FAC2 as operand registers which would clobber earlier return values).

Using just one of the values: Sometimes it is easier to just have a single return value in a subroutine's signature (even though it actually may return multiple values): this avoids having to put `void` for all other values if you aren't really interested in those. 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.

4.13.7 Deferred ("cleanup") code

Usually when a subroutine exits, it has to clean up things that it worked on. For example, it has to close a file that it opened before to read data from, or it has to free a piece of memory that it allocated via a dynamic memory allocation library, etc. Every spot where the subroutine exits (return statement, jump, or the end of the routine) you have to take care of doing the cleanups required. This can get tedious, and the cleanup code is separated from the place where the resource allocation was done at the start.

The `defer` keyword can be used to schedule a statement (or block of statements) to be executed just before exiting of the current subroutine. That can be via a return statement or a jump to somewhere else, or just the normal ending of the subroutine. This is often useful to "not forget"to clean up stuff, and if the subroutine has multiple ways or places where it can exit, it saves you from repeating the cleanup code at every exit spot. Multiple defers can be scheduled in a single subroutine (up to a maximum of 8). The defers are handled in reversed (LIFO) order. Return values are evaluated before any deferred code is executed. You write defers like so:

```
sub example() -> bool {
    ubyte file = open_file()
    defer close_file(file)              ; "close it when we exit from here"

    uword memory = allocate(1000)
    if memory==0
        return false
    defer deallocate(memory)            ; "deallocate when we exit from here"

    process(file, memory)
    return true
}
```

In this example, the two deferred statements are not immediately executed. Instead, they are executed when the subroutine exits at any point. So for example the `return false` after the memory check will automatically also close the file that was opened earlier because the close_file() call was scheduled there. At the bottom when the `return true` appears, *both* deferred cleanup calls are executed: first the deallocation of the memory, and then the file close. As you can see this saves you from duplicating the cleanup logic, and the logic is declared very close to the spot where the allocation of the resource happens, so it's easier to read and understand.

It's possible to write a defer for a block of statements, but the advice is to keep such cleanup code as simple and short as possible.

Caution

Defers only work for subroutines that are written in regular Prog8 code. If a piece of inlined assembly somehow causes the routine to exit, the compiler cannot detect this, and defers won't be handled in such cases.

Attention

you cannot use anything in a defer statement that depends on values on the CPU stack, for example pop() to use a previously push()-ed value. This is because defers are handled using an internal subroutine call so a return address is pushed on the stack before the defer code is executed.

Attention

Using defer always has a slight code overhead. If you are returning non-constant values in a routine that uses defer, the compiler even has to insert some additional code that uses the cpu stack to save some temporary values.

4.14 Library routines and builtin functions

There are many routines available in the compiler libraries or as builtin functions. The most important ones can be found in the *Library modules and builtin functions* chapter.

CHAPTER

FIVE

VARIABLES AND VALUES

Because this is such a big subject, variables and values have their own chapter. Structs and pointers are in a separate chapter again: *Structs and Pointers*.

5.1 Variables

Variables are named values that can be modified during the execution of the program. 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.

5.1.1 Declaring a variable

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> ]
```

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
long        large   = 998877
ubyte       x,y,z                   ; declare three ubyte variables x y and z
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]*5       ; initialize with five 255 bytes
```

(continues on next page)

(continued from previous page)

```
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
byte  @nozp memvar                  ; variable that is never in zeropage
```

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.
@re-quire	force the variable into Zero page. If ZP is full, compilation will fail.
@noz	force the variable to normal system ram, never place it into zeropage.
@sha	means the variable is shared with some assembly code and that it cannot be optimized away if not used elsewhere.
@nos plit	(only valid on (u)word arrays) Store the array as a single linear array instead of a separate array for lsb and msb values
@alig word	aligns string or array variable on an even memory address
@alig	aligns string or array variable on a 64 byte address interval (example: for C64 sprite data)
@alig page	aligns string or array variable on a 256 byte address interval (example: to avoid page boundaries)
@dir	the variable won't be set to zero when entering the subroutine (note: it will still be set to zero once on program startup, like all other uninitialized variables). You'll usually have to make sure to assign a value yourself before using the variable! This is used to reduce overhead in certain scenarios. □□ Footgun warning.

You can use the `private` keyword (must come first, before the datatype and any tags) to make a variable or constant invisible from outside its block:

```
private ubyte secret = 42
private ubyte @shared hiddenvar = 10
private const ubyte fixed_value = 99
```

This makes the variable only accessible within the block where it's declared. Accessing it from another block will result in a compilation error.

Variables 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
```

(continues on next page)

(continued from previous page)

```
"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. By the way, there is also `@nozp` to keep a variable *out of the zeropage* at all times.

Example:

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

shared variables: 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
```

uninitialized variables: All variables will be initialized by prog8 at startup, they'll get their assigned initialization value, or be cleared to zero. This (re)initialization is also done on each subroutine entry for the variables declared in the subroutine.

There may be certain scenarios where this initialization is redundant and/or where you want to avoid the overhead of it. In some cases, Prog8 itself can detect that a variable doesn't need a separate automatic initialization to zero, if it's trivial that it is not being read between the variable's declaration and the first assignment. For instance, when you declare a variable immediately before a for loop where it is the loop variable. However Prog8 is not yet very smart at detecting these redundant initializations. If you want to be sure, check the generated assembly output.

In any case, you can use the `@dirty` tag on the variable declaration to make the variable *not* being reinitialized when entering the subroutine (it will still be set to 0 once at program startup). This means you usually have to make sure to assign a value yourself, before using the variable. □□ Footgun warning.

memory alignment: A string or array variable can be aligned to a couple of possible interval sizes in memory. The use for this is very situational, but two examples are: sprite data for the C64 that needs to be on a 64 byte aligned memory address, or an array aligned on a full page boundary to avoid any possible extra page boundary clock cycles on certain instructions when accessing the array. You can align on word, 64 bytes, and page boundaries:

```
ubyte[] @alignword array = [1, 2, 3, 4, ...]
ubyte[] @align64 spritedata = [ %00000000, %11111111, ...]
ubyte[] @alignpage lookup = [11, 22, 33, 44, ...]
```

5.1.2 Initializing a variable

You can specify an initialization value in the variable declaration. This will then be used to initialize the variable with at the start of the subroutine, instead of the default value 0. The provided value doesn't have to be a constant; it can be any expression. It is a shorter notation for declaring the variables and then assigning the values to them in separate assignment statement(s).

There are a few special situations:

initializing an array: `ubyte[3] array = [11,22,33]`
: The initialization value has to be a range value or an array literal (remember you can use '[4] * 3'and such). Of course the size of the range or the number of values in the array has to match the declared array size.

initializing a multi variable declaration with different values: `ubyte a,b,c = 11,22,33`
: Here we have separate initialization values for each of the declared variables in that order. This is just a shorter way to write this as writing it as three separate variable declarations.

initializing a multi variable declaration with the same value for all: `ubyte a,b,c = 42`
: The initialization value here is a single constant value which will then be assigned to each of the variables.

initializing from a subroutine returning multiple result values: `ubyte a,b,c = multi()`
: Here the initialization value can also be a subroutine call to a subroutine returning multiple result values, which will then be put into the declared variables in order. Of course the number of values has to match the number of variables.

5.2 Constants

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). Only the simple numeric types (byte, word, long, float) and pointer types 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`, `strings` 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.

5.3 Enums

There is a more convenient way to define a bunch of constants that belong together: a "enum" . That is a grouped list of constants that get autonumbered for you (unless you override the numeric value yourself). It starts numbering from zero by default. Right now the enum declaration does not define a new type, only a list of constants. Here's an example:

```
enum Priority {
    LOW = 1,
    NORMAL,
    HIGH,
    EXTREME=255
}
```

This will define a bunch of constants like so (the “Enum::Field”syntax is specific for enumeration elements):

```
const ubyte Priority::LOW = 1
const ubyte Priority::NORMAL= 2
const ubyte Priority::HIGH = 3
const ubyte Priority::EXTREME = 255
```

5.4 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`
`long`	signed 32 bits integer	4 bytes	`long large = $12345678` there is no unsigned long type at the moment.
`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`. The 5 bytes per float is on CBM targets.
`bool[x]`	boolean array	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"]` note: equivalent to a uword array.
`str`	string (PETSCII)	varies	`str myvar = "hello."` implicitly terminated by a 0-byte
`^^type`	typed pointer	2 bytes	pointer types are explained in their own chapter *Structs and Pointers*

5.4.1 Integers (bytes, words, longs)

Integers are 8, 16 or 32 bit numbers and can be written in normal decimal notation, in hexadecimal and in binary notation. There is no octal notation. Hexadecimal has the '$'prefix, binary has the '%'prefix. 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.

You can use underscores to group digits to make long numbers more readable: any underscores in the number are ignored by the compiler. For instance `3_000_000` is a valid decimal number and so is `%1001_0001` a valid binary number.

A single character in single quotes such as `'a'` is translated into a byte integer, which is the PETSCII value for that character. You can prefix it with the desired encoding, like with strings, see *Strings*.

Endianness: all integers are stored in *little endian* byte order, so the Least significant byte first and the Most significant byte last.

bytes versus words versus longs:

Prog8 tries to determine the data type of integer values according to the table below, and sometimes the context in which they are used.

value	datatype
-128 .. 127	byte
0 .. 255	ubyte
-32768 .. 32767	word
0 .. 65535	uword
-2147483648 .. 2147483647	long (there is no unsigned long right now)

Numeric expressions usually 'stay within their type'unless a cast is used, see *Arithmetic and Logical expressions*. 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`. This is useful in expressions where you want a calculation to be done on word values, and don't want to explicitly have to cast everything all the time. For instance:

```
ubyte  column
uword  offset = column * 64       ; does (column * 64) as uword, wrong result?
uword  offset = column * $0040    ; does (column as uword) * 64 , a word␣
↪calculation
```

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.

Attention

Not all operations on Long integers are supported at the moment, although most common operations should work fine. Notably absent for now are multiplication and division of

longs. There is no unsigned long type at the moment, but you can sometimes simply treat the signed long value as an unsigned 32 bits value just fine. Operations on long integers take a lot of instructions on 8 bit CPUs so code that uses them a lot will be much slower than when you restrict yourself to 8 or 16 bit values. Use long values sparingly.

Danger

longs and cx16.R12,R13,R14,R15: **Some operations on long values require the use of the R12-R15 virtual register as temporary storage** So if you are working with long values, you should assume that the contents of R12-R15 could be destroyed. (Many operations preserve the values, but not all, because of reasons) **Using R12,R13,R14,R15 in expressions that work with longs, will sometimes give a corrupted result, without a warning of the compiler!** It is strongly advised to *not* use R12-R15 at all when dealing with longs.

5.4.2 Booleans

Booleans are a distinct type called `bool` in Prog8 and can have only the values `true` or `false`. In memory, they are stored as a byte containing 0 or 1. You can cast any numeric to a bool, in which case 0 will become `false` and any nonzero value will become `true`.

5.4.3 Floating point numbers

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 cbm-compatible systems, 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, which Prog8 doesn't support yet).

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**)

You can use underscores to group digits in floating point literals to make long numbers more readable: any underscores in the number are ignored by the compiler. For instance `30_000.999_999` is a valid floating point number 30000.999999.

Attention

On the X16, make sure rom bank 4 is still active before doing floating point operations (it's the bank that contains the fp routines). On the C64, you have to make sure the Basic ROM is still banked in (same reason).

5.4.4 Arrays

Arrays can be created from a list of booleans, bytes, words, floats, addresses of other variables (such as explicit address-of expressions, strings, or other array variables), pointers, struct initializers, or `memory()` calls (either directly or via `const` variables holding `memory()` results). The values in an array literal always have to be constants. A trailing comma is allowed, sometimes this is easier when copying values or when adding more stuff to the array later. 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]*99          ; initialize array with 99 times 255 [255,␣
↪255, 255, 255, ...]
byte[] array = 100 to 199           ; initialize array with [100, 101, ..., 198,
↪ 199]
str[] names = ["ally", "pete"]      ; array of string pointers/addresses␣
↪(equivalent to 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)
^^float[3] values                   ; array of three pointers to floats
uword[2] memarr = [memory("m1",10,0), memory("m2",20,0)]  ; array of memory␣
↪block addresses

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

To allow the 6502 CPU to efficiently access values in an array, the array should be small enough to be indexable by a single byte index. This means byte arrays should be <= 256 elements, word arrays <= 256 elements as well (if split, which is the default. When not split, the maximum length is 128. See below for details about this distinction). Float arrays should be <= 51 elements.

Arrays can be initialized with a range expression or an array literal value. You can write out such an initializer value over several lines if you want to improve readability. When an initialization value is given, you are allowed to omit the array size in the declaration, because it can be inferred from the initialization value. You can use '*'to repeat array fragments to build up a larger array.

You can assign a new value to an element in the array, but you can't assign a whole new

array to another array at once. This is usually a costly operation. If you really need this you have to write it out depending on the use case: you can copy the memory using `sys.memcopy(sourcearray, targetarray, sizeof(targetarray))`. Or perhaps use `sys.memset` instead to set it all to the same value, or maybe even simply assign the individual elements.

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.

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

5.4.5 2D Arrays

Prog8 provides a convenient syntactic sugar for working with two-dimensional arrays. Instead of manually calculating offsets into a flat 1D array, you can declare and access arrays using familiar 2D syntax.

Declaration uses two bracket pairs to specify the number of rows and columns:

```
ubyte[3][4] matrix   ; 3 rows, 4 columns (12 elements total)
```

Access uses chained indexing with `[row][column]`:

```
ubyte value = matrix[1][2]   ; element at row 1, column 2
matrix[0][0] = 42            ; set top-left element
```

This is purely **syntactic sugar**. The compiler automatically transforms `matrix[row][col]` into the equivalent 1D index calculation `matrix[row * numCols + col]`. There is no runtime overhead or special 2D tracking —after compilation it behaves exactly like a regular 1D array access.

Initialization:

When initializing a 2D array, provide the values as a **flat list** that matches the total number of elements (rows × columns). Nested list initializers (such as `[[1,2,3],[4,5,6]]`) are **not supported**:

```
ubyte[2][3] matrix = [1, 2, 3, 4, 5, 6]   ; correct: flat list
```

Size limitations:

The combined array size is subject to the same limits as regular 1D arrays:

- For byte arrays (`ubyte`, `byte`, `bool`): maximum **256 elements** total (rows × columns ≤ 256).
- For split word arrays (`uword`, `word`, `str`): maximum **256 elements** total, which occupies 512 bytes of storage (LSB and MSB arrays).
- For long arrays (`long`): maximum **64 elements** total (rows × columns ≤ 64), which occupies 256 bytes.
- For float arrays (`float`): maximum **51 elements** total (rows × columns ≤ 51), which occupies 255 bytes.
- For sequential word arrays (`@nosplit uword` etc): maximum **128 elements** total, which occupies 256 bytes.
- The `@split` tag works normally with 2D syntax.

- The `@nosplit` tag can also be used on 2D word arrays if sequential storage is needed.

Not supported:

- 3D or higher-dimensional arrays are not supported.
- Chained indexing (`arr[y][x]`) can only be used on variables that were explicitly declared with the 2D `[rows][cols]` syntax. Using it on a regular 1D array variable will produce a compile error.

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, negative indexing for pointer variables does *not* mean it will be counting from the end, because the size of the buffer is unknown. Instead, it simply addresses memory that lies *before* the pointer variable. See also *Direct access to memory locations ('peek'and 'poke')* and the chapter about it *Structs and Pointers*.

LSB/MSB split word and str arrays:

As an optimization, (u)word arrays, pointer arrays, and str arrays are split by the compiler in memory as two separate arrays, one with the LSBs and one with the MSBs of the word values. This is more efficient to access by the 6502 cpu. It also allows a maximum length of 256 for word arrays, where normally it would have been 128.

For normal prog8 array indexing, the compiler takes care of the distinction for you under water. *But for assembly code, or code that otherwise accesses the array elements directly, you have to be aware of the distinction from 'normal'arrays.* In the assembly code, the array is generated as two byte arrays namely `name_lsb` and `name_msb`, immediately following each other in memory.

The `@nosplit` tag can be added to the variable declaration to *not* split the array. This is useful for compatibility with code that expects the words to be sequentially in memory (such as the cx16.FB_set_palette routine).

Note

Some obscure array operations may not yet be supported on "split word arrays". If you get a compiler error message hinting that this is the case, simply revert to a regular sequential word array using `@nosplit`, and please report the issue so that the missing function can be added.

Note

Array literals are stored as split arrays if they're initializing a split word array, otherwise, they are stored as sequential words! So if you pass one directly to a subroutine (like `func([1111,2222,3333])`), the array values are sequential in memory. If this is undesirable (i.e. the subroutine expects a split word array), you have to create a normal array variable first and then pass that to the subroutine.

Caution

Be aware that the default is to split word arrays. Normal array access is taken care of by Prog8, so you won't notice this optimization. However if you are accessing the array's values using other ways (for example via a pointer, and then using `peekw` to get the value) you have to be aware of this. In that `peekw` example you have to make sure to use `@nosplit` on the word array so that the words stay sequentially in memory which is what `peekw` needs. Also be careful when passing arrays to library routines (this is via a pointer!): you have to make sure the library routine can deal with the split array otherwise you have to use `@nosplit` as well.

5.4.6 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.

Strings without an encoding prefix are stored in the machine's default character encoding (which is PETSCII on the CBM machines, but can be something else on other targets). There are ways to change the encoding: prefix the string with an encoding name, or use the `%encoding` directive to change it for the whole file at once. Here are examples of the possible 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:"Хозяин и Работник"` 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) See note below!
- `kata:" □□□□□□□□□□□□□□□□□□□□# □ # □"` string in "kata"encoding (Katakana)
- `c64os:"^Hello_World! \\ ~{_}~"` string in "c64os"encoding (C64 OS)

So what follows below 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 capability):

```
iso:"Käse, Straße"
```

str parameters in subroutines

A subroutine parameter declared as type `str` will be changed into a `^^ubyte` type (pointer to ubyte) instead because Prog8 doesn't pass strings by value. This means you *can* assign new values to this parameter variable: you can set it to a new memory address. You're not actually assigning something to a string variable, because that's not what its type actually is.

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.

You cannot assign a new value to a `str` variable. If you really want to set a new string value into the variable, you have to explicitly copy it over the old value with `strings.copy()` or `strings.ncopy()` for instance.

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 "♠♥♣♦π▚●○╳". 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.)

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 `strings.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 `strings.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!)

Note

printing **cp437** encoded strings

To print strings in the **cp437** encoding, you will probably need `txt.print_lit(message)` to properly print them to the screen. This is because this encoding has symbols in place of where normally ASCII control characters such as Line feed would be. A regular `txt.print(message)` will likely get confused by such symbols and print them as control characters, messing up the output.

5.4.7 Structs and Pointers

Struct and Pointer types are explained in their own separate chapter *Structs and Pointers*.

5.4.8 Ranges

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

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]
```

5.4.9 Memory-mapped

When using & (the address-of operator but now applied to the datatype in the variable's declaration), the variable will be placed at a designated position in memory rather than being newly allocated somewhere. The initial value in the declaration should be the valid memory address where the variable should be placed. Reading the variable will then read its value from that address, and setting the variable will directly modify those 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
```

If you need to use the variable's memory address instead of the value placed there, you can still use *&variable* as usual. You can memory map all datatypes except strings.

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

Usually specific memory locations are accessed through a memory-mapped variable, such as `cbm.BGCOL0` that is defined as the background color register at the memory address `$d021` (on the c64 target).

If you want to access any 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
```

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.
```

More information about *typed pointers* can be found in the chapter *Structs and Pointers*.

5.4.11 Converting/Casting 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
```

Note

The `as` operator has very low precedence, lower than almost all other operators. This means that `a + b as long` is parsed as `(a + b) as long`. Use parentheses if you want to cast an operand before an operation: `(a as long) * b`.

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.

5.4.12 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)

CHAPTER

SIX

STRUCTS AND POINTERS

Attention

The 6502 cpu lacks some features (addressing modes, registers) to make pointers work efficiently. Also it requires that pointer variables have to be in zero page, or copied to a temporary zero page variable, before they can even be used as a pointer. This means that pointer operations in prog8 compile to rather large and inefficient assembly code most of the time, when compared to direct array access or regular variables. At least try to place heavily used pointer variables in zero page using `@requirezp` on their declaration, if zero page space allows. Pointer variables that don't have a zeropage tag specified will be treated as having `@zp` so they get priority over other variables to be placed into zeropage.

Note

Due to a few limitations in the language parser, some pointer related syntax is currently unsupported. The compiler tries its best to give a descriptive error message but sometimes there is still a parser limitation that has to be worked around at the moment. For example, this assignment syntax doesn't parse correctly:

```
^^Node  np
np[2].field = 9999          ; cannot use this syntax as assignment target␣
↪right now
ubyte value = np[2].field   ; note that using it as expression value works␣
↪fine
```

To work around this you'll have to explicitly write the pointer dereferencing operator, or break up the expression in multiple steps (which can be beneficial too when you are assigning multiple fields because it will save a pointer calculation for every assignment):

```
^^Node  np
np[2]^^.field = 9999

; alternatively, split up:
^^Node thenode = &&np[2]
thenode.field = 9999
```

6.1 Legacy untyped pointers (uword)

Prior to version 12 of the language, the only pointer type available was a plain `uword` value (the memory address) which could be used as a pointer to an `ubyte` (the byte value at that memory address). Array indexing on an `uword` simply means to point to the `ubyte` at the location of the address + index value.

When the address of a value (explicitly) or a value of a reference type (string, array) was passed as an argument to a subroutine call, it became one of these plain `uword` 'pointers'. The subroutine receiving it always had to interpret the 'pointer'explicitly for what it actually pointed to, if that wasn't a simple byte.

Some implicit conversions were allowed too (such as putting `str` as the type of a subroutine parameter, which would be changed to `uword` by the compiler).

Since Prog8 version 12 there now are *typed pointers* that better express the intent and tell the compiler how to use the pointer; these are explained below.

For backward compatibility reasons, this untyped uword pointer still exists in the language. You can assign any other pointer type to an untyped pointer variable (uword) without the need for an explicit cast. You can assign an untyped pointer (uword) to a typed pointer variable without the need for an explicit cast.

6.2 Typed pointer to simple datatype

Prog8 syntax has the 'double hat'token `^^` that appears either in front of a type ("pointer to this type") or after a pointer variable ("get the value it points to"- a pointer dereference).

So the syntax for declaring typed pointers looks like this:

`^^type`: pointer to a type
: You can declare a pointer to any numeric datatype (bytes, words, longs, floats), and booleans, and struct types. Structs are explained in the next section. So for example; `^^float fptr` declares fptr as a pointer to a float value.

`^^type[size]`: array with size size containing pointers to a type.
: So for example; `^^word[100] values` declares values to be an array of 100 pointers to words. Note that an array of pointers (regardless of the type they point to) is always a split word array. (this is the most efficient way to access the pointers, and they need to be copied to zeropage first to be able to use them anyway. It also allows for arrays of up to 256 pointers instead of 128.)

It is not possible to define pointers to *arrays*; `^^(type[])` is invalid syntax.

Pointers of different types cannot be assigned to one another, unless you use an explicit cast. This rule is not enforced for untyped pointers/regular uword, as described earlier.

The `str` type in subroutine parameters and return values has always been a bit weird in the sense that in these cases, the string is actually passed by reference (it's address pointer is passed) instead of a `str` variable that is accessed by value. In previous Prog8 versions these were untyped uword pointers, but since version 12, these are now translated as `^^ubyte`. Resulting assembly code should be equivalent still.

Note

Pointers to subroutines: While Prog8 allows you to take the address of a subroutine, it has no support yet for typed function pointers. Calling a routine through a pointer with `goto`, `call()` and such, only works with the raw uword address for now.

6.3 Dereferencing a pointer, pointer arithmetic

To get the value the pointer points at, you *dereference* the pointer. The syntax for that is: `pointer^^`. Say the pointer variable is of type `^^float`, then `pointer^^` will return the float value it points at.

You can also use array indexing syntax to get the n-th value. For example: `floatpointer[3]` will return the fourth floating point value in the sequence that the floatpointer points at. Because the pointer is a typed pointer, the compiler knows what the size of the value is that it points at and correctly skips forward the required number of bytes in memory. In this case, say a float takes 5 bytes, then `floatpointer[3]` will return the float value stored at memory address floatpointer+15. Notice that `floatpointer[0]` is equivalent to `floatpointer^^`.

You can add and subtract values from a pointer, this is called **pointer arithmetic**. For example, to advance a pointer to the next value, you can use `pointer++`. To make it point to the preceding value, you can use `pointer--`. **Adding or subtracting X to a pointer will change the pointer by X times the size of the value it points at (the same as the C language does it), instead of simply adding or subtracting the value from the pointer address value.** (that is what Prog8 still does for untyped uword pointers, or pointers to a type that just takes up a single byte of memory).

That special pointer arithmetic is also performed for pointers to struct types: the compiler knows the memory storage size of the whole struct type and advances or rewinds the pointer value (memory address) by the appropriate number of bytes (X times the size of the struct). More info about structs can be found below.

6.4 Structs

A struct is a grouping of multiple variables. Say your game is going to track several enemy sprites on the screen, in which case it may be useful to describe the various properties of an enemy together in a struct type, rather than dealing with all of them separately. You first define the struct type like so:

```
struct Enemy {
    ubyte xpos, ypos
    uword health
    bool elite
}
```

You can use boolean fields, numeric fields (byte, word, long, float), and pointer fields (including str, which is translated into `^^ubyte`). You cannot nest struct types, but inline arrays are supported as fields. 2D arrays are not allowed as struct fields. Fields in a struct are 'packed' (meaning the values are placed back-to-back in memory), and placed in memory in order of declaration. This guarantees exact size and place of the fields. `sizeof()` knows how to calculate the combined size of a struct, and `offsetof()` can be used to get the byte offset of a given field in the struct. The size of a struct cannot exceed 1 memory page (256 bytes).

You can copy the whole contents of a struct to another one by assigning the dereferenced pointers:

```
^^Enemy e1,e2
e1^^ = e2^^     ; copies all fields of e2 into e1
```

The struct type creates a new name scope, so accessing the fields of a struct is done as usual with the dotted notation. Because it implies pointer dereferencing you can usually omit the explicit ^^, prog8 will know what it means:

```
if e1.ypos > 300
    e1.health -= 10

; explicit dereferencing notation:

if e1^^.ypos > 300
    e1^^.health -= 10
```

Note

Structs are currently only supported as a *reference type* (they always have to be accessed through a pointer). It is not yet possible to use them as a value type, or as memory-mapped types. This means you cannot create an array of structs either - only arrays of pointers to structs. There are a couple of simple case where the compiler does allow assignment of struct instances though, and it will automatically copy all the fields for you. You are allowed to write:

```
ptr2^^ = ptr1^^
ptr2^^ = ptr1[2]
ptr2[2] = ptr1^^
```

The compiler replaces this with a memory copy if these are pointers to a struct. In the future more cases may be supported.

Note

Using structs instead of plain arrays usually results in more and less efficient code being generated. This is because the 6502 CPU is not particularly well equipped to dealing with pointers and accessing struct fields via offsets, as compared to direct variable access or array indexing. The prog8 program code may be easier to work with though!

> **Note**
>
> Accessing the first field in a struct is more efficient than subsequent fields, because it is at offset 0 so no additional addition has to be computed on a pointer to reach the first field. Try to put the most often accessed field as the first field to potentially gain a rather substantial boost in code efficiency.

6.4.1 Static initialization of structs

You can 'allocate'and statically initialize a struct. This behave much like initializing arrays does, and it won't reset to the original value when the program is restarted, so beware. *Remember that the struct is statically allocated, and appears just once in the memory:* This means that, for instance, if you do this in a subroutine that gets called multiple times, or inside a loop, the struct *will be the same instance every time*. Read below if you need *dynamic* struct allocation! You write a static struct initialization expression like this:

^^Node : [1,"one", 1000, true, 1.111]
: statically places an instance of struct 'Node'in memory, with its fields set to 1, "one" , 1000 etcetera and returns the address of this struct. The values in the initialization array must correspond exactly with the first to last declared fields in the struct type.

If the struct contains inline arrays, you can initialize them by nesting another list inside the initialization list:

^^Node : [1, [10, 20, 30, 40], 1000]
: Initializes a struct Node where the second field is an inline array (e.g., `ubyte[4] data`). The nested array `[10, 20, 30, 40]` initializes that array field.

^^Node : []
: (without values) Places a 'Node'instance in BSS variable space instead, which gets zeroed out at program startup. Returns the address of this empty struct.

The field values in a struct initializer must be constants. You can use numeric literals, `address-of` expressions, or `memory()` calls (either directly or via `const` variables). It is also possible to put struct initializer inside arrays to make them all statically initialized and accessible via the array:

```
^^Node[] allnodes = [
    ^^Node: [1,"one", 1000, true, 1.111],
    ^^Node: [2,"two", 2000, false, 2.222],
    ^^Node: [],
    ^^Node: [],
]
```

Short form initializers

If the required type can be inferred from the context you can also omit the struct pointer type prefix altogether. The initializer value then is syntactically the same as an array, but Prog8 internally turns it back into a proper struct initializer value based on the the type of the array element or pointer variable it is assigned to. So you can write the above in short form as:

```
^^Node nodepointer = [1,2,3,4]

^^Node[] allnodes = [
```

(continues on next page)

(continued from previous page)

```
    [1,"one", 1000, true, 1.111],
    [2,"two", 2000, false, 2.222],
    [],
    []
]
```

6.4.2 Dynamic allocation of structs

There is no real 'dynamic'memory allocation in Prog8. Everything is statically allocated. This doesn't change with struct types. However, it is possible to write a dynamic memory handling library yourself (it has to track memory blocks manually). If you ask such a library to give you a pointer to a piece of memory with size `sizeof(Enemy)` you can use that as a dynamic pointer to an Enemy struct.

An example of how a super simple dynamic allocator could look like:

```
^^Node newnode = allocator.alloc(sizeof(Node))
...

allocator {
    ; extremely trivial arena allocator
    uword buffer = memory("arena", 2000, 0)
    uword next = buffer

    sub alloc(ubyte size) -> uword {
        defer next += size
        return next
    }

    sub freeall() {
        ; cannot free individual allocations only the whole arena at once
        next = buffer
    }
}
```

6.5 Address-Of: untyped vs typed

& still returns an untyped (uword) pointer, as it did in older Prog8 versions. This is for backward compatibility reasons so existing programs don't break. The new *double ampersand* operator && returns a *typed* pointer to the value. The semantics are slightly different from the old untyped address-of operator, because adding or subtracting a number from a typed pointer uses *pointer arithmetic* that takes the size of the value that it points to into account.

6.6 Accessing struct definitions in Assembly code

Prog8 lets you query the size of a struct type, and the offsets of a field. You can do the same in assembly code if needed: the struct definition gets written as *.struct* into the assembly file:

```
; prog8 struct declaration:
struct Node {
    ubyte type
    uword value
    bool flag
}

; generates this assembly code:
; (the symbol prefixes are explained in the 'Technical details' chapter)
p8b_main.p8t_Node    .struct f0,f1,f2
p8v_type  .byte  \f0
p8v_value  .word  \f1
p8v_flag  .byte  \f2
    .endstruct
```

64tass then lets you query that information:

```
; prog8 code:
ubyte size = sizeof(Node)
ubyte offset1 = offsetof(Node.type)
ubyte offset2 = offsetof(Node.value)
ubyte offset3 = offsetof(Node.flag)

; assembly equivalents (64tass syntax):
lda  #size(p8t_Node)
lda  #p8t_Node.p8v_type
lda  #p8t_Node.p8v_value
lda  #p8t_Node.p8v_flag
```

CHAPTER

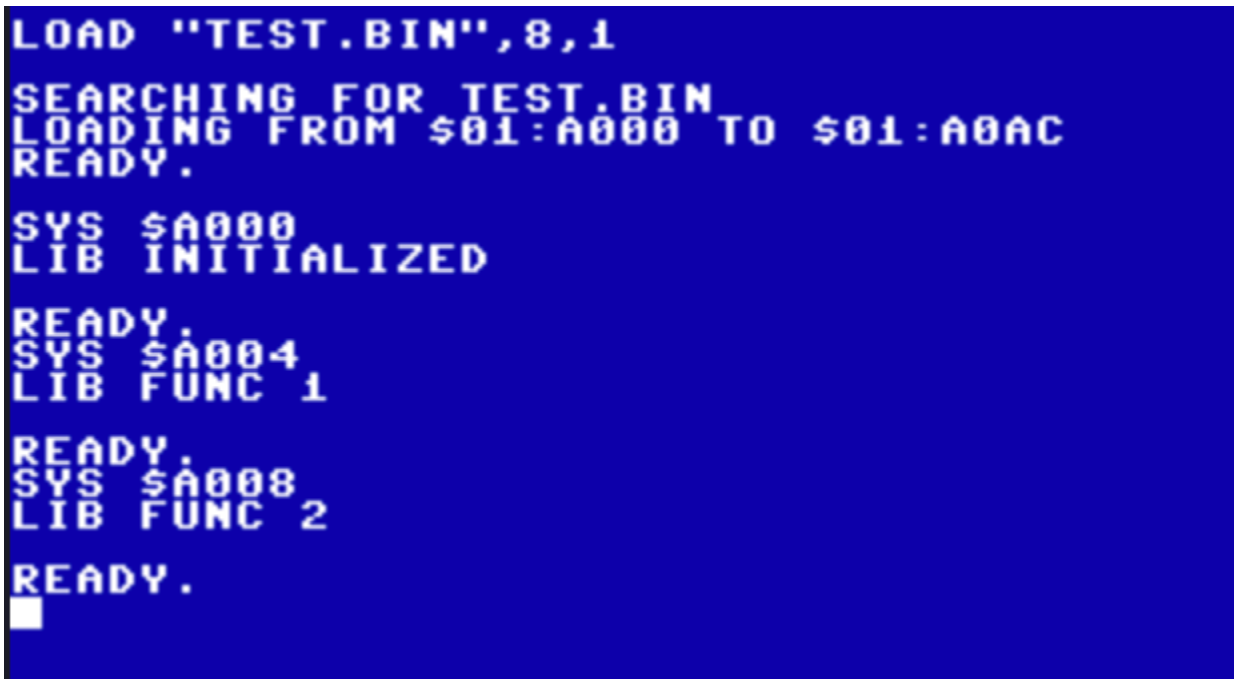
SEVEN

BINARY LOADABLE LIBRARIES

also called 'Library Blobs'.

Prog8 allows you to create binary library files that contain routines callable by other programs. Those programs can be written in Prog8, BASIC, or something else. They just LOAD the binary library file into memory, and call the routines.

An example of a library file loaded in BASIC on the Commander X16:



(On the Commodore-64 and such, it works identically but you have to type the SYS addresses in decimal notation)

7.1 Requirements

Such a loadable library has to adhere to a few rules:

It can't use zero page variables
: Otherwise it might overwrite variables being used by the calling program. For systems that have the 16 'virtual registers'cx16.r0-r15 in zero page: these 16 words are free to use. For other systems, only the internal prog8 zeropage scratch variables can be used. *note: this may be improved upon in a future version*

No system initialization and startup code
: The library cannot perform any regular "system initialization"that normal Prog8 programs usually perform (such as resetting the IO registers, clearing the screen, changing the colors, and other initialization logic). This would disturb the state of the calling program! The library can (must) assume that that the calling program has already done all required initialization.

Variable initialization
: The library still has to initialize any variables it might use and clear uninitialized "BSS" variables! Otherwise the code will not run predictably as prog8 code. So, the library must still have a "start"entrypoint subroutine like any other prog8 program, that must be called before any other library routine can be called.

Binary output and loaded into a fixed memory address
: The library must not have a launcher such as a BASIC SYS command, because it is not ran like a normal program. Also, because it is not possible to create position independent code with prog8, a fixed load address has to be decided on and the library must be compiled with that address as the load address.

7.2 `%output library`

Most of the above requirements can be fulfilled by setting various directives in your source code such as %launcher, %zeropage and so on. But there is a single directive that does it correctly for you in one go (and makes sure there won't be any initialization code left at all): `%output library`

Together with `%address` and possibly `%memtop` -to tell the compiler what the load address of the library should be- it will create a "library.bin"file that fulfills the requirements of a loadable binary library program as listed above.

For older CBM targets (C64, C128 and PET) the library file *will* have a load address header, because these targets require a header to easily load files. For the other targets such as the Commander X16, the library will be a headerless binary file that can then be loaded given the correct load address.

The entrypoint (= the start subroutine) that must be called to initialize the variables, will be the very first thing at the beginning of the library.

7.3 Jump table

For ease of use, libraries should probably have a fixed "jump table"where the offsets of the library routines stay the same across different versions of the library. Without needing new syntax, there's a trick in Prog8 that you can use to build such a jumptable: add a non-split word array at the top of the library main block that contains JMP instructions and the addresses of the individual library subroutines. Do NOT change the order of the subroutines in this table! Also note that the Prog8 compiler will insert a single JMP instruction at the very start of the library, that jumps to the start subroutine (= the entrypoint of the library program). Users of the library need to call this to initialize the variables, so it is a required part of the external interface of the library. Because the compiler will place the global word array jumptable immediately after this JMP instruction, it seems as if the very first entry in the jump table is the jump to the start routine.

Look at the generated assembly code to see exactly what is going on. But the users of the library are none the wiser and it just seems as if it is part of the jump table in a natural way :-)

7.4 Loading and using the library

These examples below assume the target is the Commander X16. Assuming the load address of the library is $A000:

From BASIC:

```
BLOAD "LIBRARY.BIN",8,1,$A000
SYS $A000 : REM TO INITIALIZE VARIABLES, REQUIRED!
SYS $A003 : REM CALL FIRST ROUTINE
SYS $A006 : REM CALL SECOND ROUTINE, ETC.
```

From Prog8:

The `diskio` module actually provides a convenience routine called `loadlib` that loads a Prog8-compiled library blob into memory. It internally automatically uses either regular load() or load_raw(), as required by the compilation target (so you don't have to bother with target machine differences if you want to write portable code):

```
%import diskio

main {

    extsub $A000 = lib_init() clobbers(A)
    extsub $A003 = lib_func1() clobbers(A,X,Y)
    extsub $A006 = lib_func2() clobbers(A,X,Y)

    sub start() {
        if diskio.loadlib("library.bin", $a000) != 0 {
            lib_init()
            lib_func1()
            lib_func2()

            repeat { }
        }
    }
}
```

From C:

```
#include <cbm.h>

int main() {
    void (*lib_init)(void) = (void (*)()) 0xa000;
    void (*lib_func1)(void) = (void (*)()) 0xa003;
    void (*lib_func2)(void) = (void (*)()) 0xa006;

    cbm_k_setlfs(0, 8, 2);
    cbm_k_setnam("library.bin");
    cbm_k_load(0, 0xa000);

    lib_init();
    lib_func1();
    lib_func2();
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

From Assembly:

```
; add error handling as desired.
    ldy  #>libname
    ldx  #<libname
    lda  #11
    jsr  $ffbd      ; SETNAM
    lda  #0
    ldx  #8
    ldy  #2         ; load address override
    jsr  $ffba      ; SETLFS
    lda  #0
    ldx  #<$a000
    ldy  #>$a000
    jsr  $ffd5      ; LOAD
    lda  #13
    jsr  $ffd2      ; CHROUT

    jsr  $A000      ; library init
    jsr  $A003      ; lib func 1
    jsr  $A006      ; lib func 2

    rts

    libname:
        .text  "library.bin"
```

7.5 Example library code

Here is the small example library that was used in the example at the beginning of this chapter:

```
%address  $A000
%memtop   $C000
%output   library

%import textio


main {
    ; Create a jump table as first thing in the library.
    ; NOTE: the compiler has inserted a single JMP instruction at the start
    ; of the 'main' block, that jumps to the start() routine.
    ; This is convenient because the rest of the jump table simply follows it,
    ; making the first jump neatly be the required initialization routine
    ; for the library (initializing variables and BSS region).
    %jmptable (
        library.func1,
```

(continues on next page)

(continued from previous page)

```
        library.func2,
    )

    sub start() {
        ; has to be here for initialization
        txt.print("lib initialized\n")
    }
}


library {
    sub func1() {
        txt.print("lib func 1\n")
    }

    sub func2() {
        txt.print("lib func 2\n")
    }
}
```

CHAPTER

EIGHT

LIBRARY MODULES AND BUILTIN FUNCTIONS

The compiler provides several library modules with useful subroutine and variables. There are also a bunch of builtin functions.

Some of the libraries 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/8c92663824c38ed907abf3e9beb1eb92e70b5118/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.net/

8.1 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.

8.1.1 Array operations

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 `strings.length` routine if you want to dynamically determine the length by counting to the first 0-byte)

8.1.2 Math

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

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.

divmod (dividend, divisor)
Returns quotient and remainder of the division as two ubyte or uword values. Performs the division only once. Using '/'and '%'separately would perform the division twice, so using divmod is much more efficient for this. **Note:** Clobbers `cx16.r15` (the remainder is stored there for the word variant). The `%` operator on its own used to also clobber `cx16.r15` but now preserves it.

gcd (a, b)
Returns the GCD (greatest common divisor) of uwords a and b The routine is efficient and uses bit shifts instead of divisions. **Clobbers:** `cx16.r0` and `cx16.r1`.

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

min (x, y)
Returns the smallest of x and y. Supported for integer types only, for floats use `floats.minf()` 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 (x)
Returns the square root of the number. Accepts unsigned integer (result is ubyte), long (result is uword), and floating point numbers. To do the reverse - squaring a number - just write `x*x`.

8.1.3 CPU Stack

push (value)
pushes a byte value on the CPU hardware stack. Low-level function that is seldomly used in user code.

pushw (value)
pushes a 16-bit word value on the CPU hardware stack. Low-level function that is seldomly used in user code. Don't assume anything about the order in which the bytes are pushed - popw will make sense of them again.

pushl (value)
pushes a 32-bit value on the CPU hardware stack. Low-level function that is seldomly used in user code. Don't assume anything about the order in which the bytes are pushed - popl will make sense of them again.

pushf (value)
pushes a floating point value on the CPU hardware stack. Low-level function that is seldomly used in user code. Don't assume anything about the order in which the bytes are pushed - popf will make sense of them again.

pop ()
pops a byte value off the CPU hardware stack and returns it. Low-level function that is seldomly used in user code.

popw ()
pops a 16-bit word value off the CPU hardware stack that was pushed before by pushw, and returns it. Low-level function that is seldomly used in user code.

popl ()
pops a 32-bit value off the CPU hardware stack that was pushed before by pushl, and returns it. Low-level function that is seldomly used in user code.

popf ()
pops a floating point value off the CPU hardware stack that was pushed before by pushl, and returns it. Low-level function that is seldomly used in user code.

8.1.4 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 (lower) byte of the value x. Equivalent to `x & 255` or even `x as ubyte`.

lsw (x)
Get the least significant (lower) word of the value x. Equivalent to `x & 65535` or even `x as uword`.

msb (x)
Get the most significant (highest) byte of the word or long value (so for a long value, msb($11223344) is $11, not $33. To grab the bank byte of a long variable, you need to do this: `lsb(msw(longvariable))` or the equivalent `@(&longvariable+2)`.) But also look at the `lmh` function.

lmh (x)
Get the three low, mid and high (bank) bytes of the given long value. The upper byte (bits 24-31) of the long value is not considered. So this means lmh($11223344) returns the three bytes $44, $33, $22 in that order. For the Commander X16 a possible use of this function is to set the 24-bits Vera address in one assignment statement: `cx16.VERA_ADDR_L, cx16.VERA_ADDR_M, cx16.VERA_ADDR_H = lmh(address)`

msw (x)
Get the most significant (higher) word of the value x. For all word and byte numbers this will always result in 0. For a long integer though, it returns the upper 16 bits of x as an uword. If x is a constant integer not greater than a 24 bit number ($ffffff), `msw(x)` will actually give you the bank byte of x (bits 16 to 23). You can treat this as an ubyte value,

even if the function is normally returning a uword: `msw($123456)` is $0012, which you can treat as an ubyte. `msw($12345678)` is $1234, an uword. `msw` of a *variable* will always be considered to be an uword, so to grab the bank byte of a long variable, you need to do this: `lsb(msw(longvariable))` or the equivalent `@(&longvariable+2)`.

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 are in 'natural'left to right reading 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)

mklong (msb, b2, b1, lsb)
Efficiently create a long value from four bytes (the msb, second, first and finally the lsb). Avoids multiplication and shifting. So mklong($12, $34, $56, $78) results in $12345678.

> **Note**
>
> The arguments are in 'natural'left to right reading order that is first the msb then the lsb. Don't get confused by how the system actually stores this 32-bit word value in memory (which is in little-endian format, so lsb first then b1, b2 and finally the msb)

mklong2 (msw, lsw)
Efficiently create a long value from two words (the msw, and the lsw). Avoids multiplication and shifting. So mklong2($1234, $abcd) results in $1234abcd.

> **Note**
>
> The arguments are in 'natural'left to right reading order that is first the msw then the lsw. Don't get confused by how the system actually stores this 32-bit word value in memory (which is in little-endian format, so lsw first then the msw)

offsetof (Struct.field)
The offset in bytes of the given field in the struct. The first field will always have offset 0. Usually you just reference the fields directly but in some cases it might be useful to know how many bytes from the start of the structure a field is located at.

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

peekbool (address)
Reads the boolean value (byte 0 or 1) at the given address in memory and returns it. If the memory location contains another value than 0 or 1, results are undefined.

peekw (address)
reads the word value at the given address in memory. Word is read as usual little-endian lsb/msb byte order. Caution: when using peekw to get words out of an array pointer, make sure the array is *not* a split word array (peekw requires the LSB and MSB of the word value to be consecutive in memory).

peekl (address)
reads the signed long value at the given address in memory. Long 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.

pokebool (address, value)
Writes the boolean value at the given address in memory, as byte 0 or 1. Can also be written as pokebowl(addres, value), just for fun.

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

pokel (address, value)
writes the signed long 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))` You can use `if_cc` or `if_cs` after a rol to act on the new carry bit, if required.

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))` You can use `if_cc` or `if_cs` after a ror to act on the new carry bit, if required.

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 or long variable x to a new value.

setmsb (x, value)
Sets the most significant byte of word or long variable x to a new value.

sizeof (name) ; sizeof(datatype) ; sizeof(&name) ; sizeof(&&name) ; sizeof(^^type)
The constant number of bytes that the object 'name', the type 'datatype', or a pointer occupies in memory. For instance, for a variable of type uword, the sizeof is 2. For a 10 element array of floats, it is 50 (on the C64, where a float is 5 bytes). For a string, it returns the size of the string in memory (which includes the 0-byte terminator at the end). For address-of expressions like `&variable` or `&&variable`, it returns the size of a pointer (2 bytes). For pointer types like `^^float` or `^^MyStruct`, it returns the size of a pointer (2 bytes). Note: usually you will be interested in the number of elements in an array, or the number of characters in the string; use `len()` for that.

swap (var1, var2)
Swaps the values in var1 and var2 without the need of a temporary variable. Supports booleans and all other numeric datatypes including pointers. Note that complicated expressions that you want to swap, may not be implemented yet. To avoid such errors you'll have to just swap them in the old fashioned way, until an optimized code path gets implemented in a future Prog8 version.

memory (name, size, alignment) ; memory(name)
Returns the address of the first location of a statically "reserved"block of memory with the given name. The 3-argument version reserves a block of the given size in bytes and optional alignment, and returns the address of it. The 1-argument version acts only as a reference to a block that must be reserved elsewhere in the program. The name must be a string literal, it cannot be empty or be a variable. 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 an uword address, and you can use that like a pointer to the memory buffer. The block is *uninitialized memory*; unlike other variables in Prog8 it is *not* set to zero at the start of the program! (if that is required, you can do so yourself using `memset`). No *dynamic* allocation is done; the block with this name is placed in memory only once! 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 (using the 3-argument version), it is required that the size and alignment arguments are identical, otherwise you'll get a compilation error. To avoid duplicating these details it's easier to use the 1-argument version and just refer to the name only.

call (address) -> uword
Calls a subroutine given by its memory address. You cannot pass arguments directly, although it is of course possible to do this via the global `cx16.r0...` registers for example. It is *not* possible to use cpu registers to pass arguments, because these are clobbered while performing the call! 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
Calls an assembly routine in another bank. 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 that you reuse. Or use `call` if you can.

callfar2 (bank, address, argA, argX, argY, argCarry) -> uword
Identical to `callfar`, except here you can give arguments not only for AY, but for each of the A, X and Y registers (each an ubyte) and the Carry status bit as well (a boolean).

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.

8.2 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.

- c64
- c128
- cx16
- pet32
- virtual

8.3 Library modules

8.3.1 adpcm (cx16 only)

Routines to decode IMA-ADPCM compressed audio sample data. This is a lossy compression that reduces the data size by a factor of 4. Reference info about the compression here and here .

The decoder routines in this module support mono and stereo streams, *but only with 256 byte block size* The standard allows for other block sizes but this decoder only accepts 256! **NOTE:** for speed reasons this implementation doesn't guard against clipping errors. If the sound playback sounds distorted, lower the volume of the source waveform a little bit and try again.

How to create a compatible ADPCM audio file? Use ffmpeg or adpcm-xq like so (example): `ffmpeg -i source.mp3 -ar 16000 -ac 1 -c:a adpcm_ima_wav -block_size 256 -map_metadata -1 -bitexact out.wav` or with adpcm_xq: `adpcm-xq -4 -b8 -n uncompressed.wav output.wav`

`sub decode_block_mono(uword nibblesptr)`
Decodes one 256 byte block of adpcm data, into the Vera's PCM FIFO buffer. Decoded data is 16 bit mono PCM, 505 samples = 1010 bytes.

sub decode_block_stereo(uword nibblesptr)
Decodes one 256 byte block of adpcm data, into the Vera's PCM FIFO buffer. Decoded data is 16 bit stereo PCM, 498 samples = 996 bytes.

If you just want to decode the data in memory you can use a few low-level routines directly to decode single nibbles etc. Look at the adpcm source code to find out what other routines are available.

8.3.2 bcd

Decimal addition and subtraction routines, so for example $0987 + $1111 = $2098 (rather than the usual hex outcome $1a98) Utilizes the BCD mode of the CPU (note: not all 6502 variants support this mode). This mode is useful for example for counting decimal score in a game, to avoid costly conversion to a decimal display string: just print the hexadecimal score representation. (This gets especially noticeable with long integers) Available routines:

- `sub addb(byte a, byte b) -> byte`
- `sub addub(ubyte a, ubyte b) -> ubyte`
- `sub addw(word a, word b) -> word`
- `sub adduw(uword a, uword b) -> uword`
- `sub addl(long a, long b) -> long`
- `sub addtol(^^long a, long b)` (adds b in-place to a, saves copying values)
- `sub subb(byte a, byte b) -> byte`
- `sub subub(ubyte a, ubyte b) -> ubyte`
- `sub subuw(uword a, uword b) -> uword`
- `sub subl(long a, long b) -> long`
- `sub subfroml(^^long a, long b)` (subtracts b in-place from a, saves copying values)

8.3.3 bmx (cx16 only)

Routines to load and save "BMX"files, the CommanderX16 bitmap file format: BMX file format specification Only the *uncompressed* bitmaps variant is supported in this library for now.

The routines are designed to be fast and they 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.

8.3.4 buffers

Provides few data buffer routines. These are available:

- `smallstack` a fast 256 byte stack (LIFO) that is independent of the CPU stack.
- `smallringbuffer` a fast 256 byte ringbuffer (FIFO).
- `stack` a stack (LIFO) with up to 8Kb of data.
- `ringbuffer` a ringbuffer (FIFO) with up to 8Kb of data.

On the Commander X16 the 8Kb stack and ringbuffer implementations use a HIRAM bank instead of regular system memory. You tell it which bank to use by calling `init(bank)` with the bank number as argument.

Read the buffers source code to see what's in there. Note that on the X16, the init() routines have that extra bank parameter.

8.3.5 cbm

Commodore (CBM) common variables, vectors and kernal routines. This 'library'is part of syslib and as such is always available. There's too much in this library to list in the docs here, but you can find things like:

- memory mapped variables such as `TIME_HI`, `TIME_MID` and `TIME_LO` (the 3 bytes making up the jiffy clock)
- cbm kernal routine extsubs such as `CHROUT` and `PLOT`
- helper routines such as `GETIN2` and `RDTIML` (convenience wrappers around existing kernal routines)

It has way too much to include here, you have to study the syslib source code to see what is there. The version linked here is the C64 version itself - on other targets in the CBM family (PET, C128 etc) it will contain different things based on what is available there. But the stuff common to the CBM family of compiler targets is there.

8.3.6 compression

Routines for data compression and decompression. For compression the 'ByteRun1'aka 'PackBits'RLE encoding is available, this is the compression that was used in old MacPaint and Amiga IFF images. Decompressors are available for RLE, TSCrunch and ZX0 (Salvador).

`encode_rle (uword data, uword size, uword target, bool is_last_block) -> uword`
Compress the given data block using ByteRun1 aka PackBits RLE encoding. Returns the size of the compressed RLE data. Worst case result storage size needed = (size + (size+126) / 127) + 1. 'is_last_block'= usually true, but you can set it to false if you want to concatenate multiple compressed blocks (for instance if the source data is >64Kb)

`encode_rle_outfunc (uword data, uword size, uword output_function, bool is_last_block)`
Like `encode_rle` but not with an output buffer, but with an 'output_function'argument. This is the address of a routine that gets a byte arg in A, which is the next RLE byte to write to the compressed output buffer or file. This avoids having to buffer the compressed result first.

`decode_rle (uword compressed, uword target, uword maxsize) -> uword`
Decodes "ByteRun1"(aka PackBits) RLE compressed data. Control byte value 128 ends the decoding. Also stops decompressing if the maxsize has been reached. Returns the size of the decompressed data.

`decode_rle_srcfunc (uword source_function, uword target, uword maxsize) -> uword`
Decodes "ByteRun1"(aka PackBits) RLE compressed data. Control byte value 128 ends the decoding. Also stops decompressing when the maxsize has been reached. Returns the size of the decompressed data. Instead of a source buffer, you provide a callback function that must return the next byte to compress in A. This is useful if the compressed data is read from a disk file for instance as this avoids having to buffer it first.

Note: the callback routine MUST NOT MODIFY the prog8 scratch variables such as P8ZP_SCRATCH_W1 etc!

decode_rle_vram (uword compressed, ubyte vbank, uword vaddr) (cx16 only)
Decodes "ByteRun1"(aka PackBits) RLE compressed data directly into Vera VRAM, without needing an intermediate buffer. Control byte value 128 ends the decoding. While the X16 has pretty fast LZSA decompression in the kernal, RLE is still about 5 times faster to decode. However it also doesn't compress data nearly as well, but that's the usual tradeoff. There is a *compression* routine as well for RLE that you can run on the X16 itself, something that the lzsa compression lacks.

decode_tscrunch (uword compressed, uword target)
Decompress a block of data compressed in the TSCrunch format. It has extremely fast decompression (approaching RLE speeds), better compression as RLE, but slightly worse compression ratio than LZSA. See https://github.com/tonysavon/TSCrunch for the compression format and compressor tool. **NOTE:** for speed reasons this decompressor is *not* bank-aware and *not* I/O register aware; it only outputs to a memory buffer somewhere in the active 64 Kb address range.

decode_tscrunch_inplace (uword compressed)
Decompress a block of data compressed in the TSCrunch format *inplace*. This can save an extra memory buffer if you are reading crunched data from a file into a buffer. It has extremely fast decompression (approaching RLE speeds), better compression than RLE, but slightly worse compression ratio than LZSA. See https://github.com/tonysavon/TSCrunch for the compression format and compressor tool. **NOTE:** for speed reasons this decompressor is *not* bank-aware and *not* I/O register aware; it only outputs to a memory buffer somewhere in the 64 Kb main memory address range.

> **Note**
>
> The TSCrunch in-place format is a bit different than regular memory decompression. It works with PRG files (so with a 2 byte load-address header) for both the *source* and *compressed* data files. So if you want to compress and decompress a block of data from $a000-$c000 your source file has to start with the bytes $00 $0a, then followed by the 8192 data bytes, for a total of 8194 bytes. Then you need to call the compressor program with the '-i'argument to tell it to create an in-place compressed data file. The data file will *not* be loaded at $a000 but have its own load address closer to the end of the memory buffer. If all is well, you can then load and decompress it like so:
>
> ```
> uword tsi_start_addr = diskio.get_loadaddress("data8kb.tsi")
> cx16.rambank(2)     ; or whatever ram bank you want on the X16
> void diskio.load("data8kb.tsi", 0)      ; not load_raw!
> cx16.rambank(2)     ; make sure the ram bank is still the same
> compression.decode_tscrunch_inplace(tsi_start_addr)
> ```

decode_zx0 (uword compressed, uword target)
Decompress a block of data compressed in the ZX0 format. This has faster decompression than LZSA, and a slightly better compression ratio as well. See https://github.com/einar-saukas/ZX0 for the compression format See https://github.com/emmanuel-marty/salvador for the compressor tool. **NOTE:** You have to use it with the "-classic"option to produce a data format that this decoder can handle! **NOTE:** for speed reasons this decompressor is *not* bank-aware and *not* I/O register aware; it only outputs to a memory buffer somewhere in the 64 Kb main memory address range.

8.3.7 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, longs)

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

8.3.8 coroutines

Provides a system to make cooperative multitasking programs via coroutines. A 'coroutine'is a subroutine whose execution can be paused and resumed. This is done in a cooperative way where the coroutine calls `coroutines.yield` to pause and yield control back to the system (which then selects the next coroutine to run). This way it can seem that your program is executing many subroutines at the same time (without the use of an interrupt routine). This library handles the voodoo required to switch between such coroutines.

Read the coroutines source code to see what's in there. And look at the `multitasking` example to see how it can be used. Better docs will be written here in the manual, at some point, but until then: here is a minimal example:

```
%import coroutine

main {
    sub start() {
        coroutines.killall()
        coroutines.add(&some_task, 1111)
        ; ... add more tasks here or later
        coroutines.run(0)
    }

    sub some_task() {
        repeat 100 {
            uword userdata = coroutines.yield()
            ; ... do something...
        }
    }
}
```

8.3.9 cx16

Despite its name, this 'module'is available on *all targets*; it also is always available because it is a part of syslib. On the Commander X16 this module contains a *whole bunch* of things specific to that machine 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.

For the Commander X16 version it has way too much to include here, you have to study the syslib source code to see what is there.

8.3.10 cx16logo

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

`logo ()`
: prints the logo at the current cursor position

`logo_at (column, row)`
: prints the logo at the given position

8.3.11 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

For simplicity sake, this library is designed to work on a *single* open file for reading, and a *single* open file for writing at any time only. If you need to load or save to more than one file at a time, you'll have to write your own I/O routines (or supplement the ones found here)

You can set the active *disk drive number*, so it supports multiple drives, just one at a time. It does not support reading from more than one file or writing to more than one file at a time.

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) Also 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

Opening a file using f_read() or f_read_w() doesn't set the default i/o channels to that file. In fact, after calling routines in diskio, it resets the input and output channels to their defaults (keyboard and screen). If you are going to do kernal I/O calls like CHRIN/CHROUT/(M)ACPTR yourself on the files opened via diskio, you must use reset_read_channel() or reset_write_channel() before doing so. This makes the correct file channel active. The diskio routines themselves do this as well internally.

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.

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. PET support is limited, not all routines are available for this system. Also there are some weird cases in CBM dos when dealing with empty (0-byte) files. It's best to avoid such files altogether.

8.3.12 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.

EOL (end of line) character handling: Writing `iso:'\n'` to the console doesn't produce a proper new line there, because prog8 encodes the newline to character 13 on the X16 (this is what the X16 uses to print a newline on the screen). You have to explicitly output a character 10 on the console to see a newline there. You can do that in several ways:

```
emudbg.console_nl()
emudbg.console_chrout(10)
emudbg.console_write(iso:"hello\x0a")
```

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

8.3.13 floats

Note

Floating point support is available on most cbm-compatible targets (except the C128 for now), and the virtual target. On the X16, make sure rom bank 4 is still active before doing floating point operations (it's the bank that contains the fp routines). On the C64, you have to make sure the Basic ROM is still banked in (same reason).

Provides definitions for the ROM/Kernal subroutines and utility routines dealing with floating point variables.

π 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, 2π]

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

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

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.

interpolate(v, inputMin, inputMax, outputMin, outputMax)
Interpolate a value v in interval [inputMin, inputMax] to output interval [outputMin, outputMax]

lerp(v0, v1, t)
Linear interpolation (LERP). Precise method, which guarantees v = v1 when t = 1. Returns an interpolation between two inputs (v0, v1) for a parameter t in the closed unit interval [0.0, 1.0]

lerp_fast(v0, v1, t)
Linear interpolation (LERP). Imprecise (but faster) method, which does not guarantee v = v1 when t = 1 Returns an interpolation between two inputs (v0, v1) for a parameter t in the closed unit interval [0.0, 1.0]

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.

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.

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)

rad (x)
Degrees to radians.

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.

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

sin (x)
Sine.

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

tan (x)
Tangent.

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

8.3.14 gfx_lores and gfx_hires (cx16 only)

Full-screen multicolor bitmap graphics routines, available on the X16 machine only.

- gfx_lores: optimized routines for 320x240 256 color bitmap graphics mode. Compatible with X16 screen mode 128.
- gfx_hires: optimized routines for 640x480 4 color bitmap graphics mode
- enable bitmap graphics 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 gfx_lores source code or gfx_hires source code to see what's in there.

They share the same routines.

8.3.15 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 X16 there's also various other graphics modules if you want more features and different screen modes. See below for those.

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

8.3.16 lineclip

Line clipping using the Elite-inspired two-stage algorithm (BBC Micro, 6502). Based on the deep dive by Mark Moxon at https://elite.bbcelite.com/deep_dives/line-clipping.html

All coordinates are signed words (`word`). The clipping rectangle is set via `set_cliprect`, then `clip` is called to clip a line segment. The algorithm handles lines that are fully inside, fully outside, or partially crossing the clipping rectangle. With `inside` you can check a single pixel.

sub set_cliprect(word x1, word y1, word x2, word y2)
Set the clipping rectangle coordinates, all inclusive. The rectangle must have x1 < x2 and y1 < y2.

sub inside(word x, word y) -> bool
Returns true if (x,y) is inside the clipping rectangle set by `set_cliprect`.

sub clip(word x1, word y1, word x2, word y2) -> bool, word, word, word, word
Clip the line segment from (x1,y1) to (x2,y2) against the current clipping rectangle set by `set_cliprect`. Returns `visible` (boolean), plus the clipped coordinates (or (0,0)-(0,0) if not visible).

8.3.17 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.

checksumming

crc16 (uword data, uword length, uword initvalue, uword xorout) -> uword
Returns a CRC-16 checksum over the given data buffer, with configurable initialization value and final result xor value. For XMODEM type checksum, use initvalue=0 and xorout=0. For IBM-3740 type checksum, use initvalue=$ffff and xorout=0. (this is then equivalent to the cx16.memory_crc routine). Many other types are possible with different values...

crc16_start(initvalue) / crc16_update(ubyte value) / crc16_end(xorout) -> 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) -> long
Calculates a CRC-32 (ISO-HDLC/PKZIP) checksum over the given data buffer. The 32 bits result is returned as a long value. The routine clobbers R0/R1 and R12 through R15.

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 that returns the result value as a long.

interpolation

lerp(v0, v1, t)
Linear interpolation routine for unsigned byte values. Returns an interpolation between two inputs (v0, v1) for a parameter t in the interval [0, 255] Guarantees v = v1 when t = 255. Also works if v0 > v1.

lerpw(v0, v1, t)
Linear interpolation routine for unsigned word values. Returns an interpolation between two inputs (v0, v1) for a parameter t in the interval [0, 65535] Guarantees v = v1 when t = 65535. Also works if v0 > v1. Clobbers R15.

interpolate(v, inputMin, inputMax, outputMin, outputMax)
Interpolate a value v in interval [inputMin, inputMax] to output interval [outputMin, outputMax] All values are unsigned bytes. Clobbers R15 (there is no version for word values because of lack of precision in the fixed point calculation there).

large multiplications

mul32 (word w1, word w2) -> long
Returns the 32 bits signed long result of w1 * w2

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.

NOTE: the result is only valid if the multiplication was done with uword arguments (or two positive word arguments). As soon as a single negative word value (or both) was used in the multiplication, these upper 16 bits are not valid! Suggestion (if you are on the Commander X16): use `verafx.muls()` to get a hardware accelerated 32 bit signed multiplication.

miscellaneous

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). **Clobbers:** `cx16.r0` through `cx16.r5` (used as temporary variables).

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) **Clobbers:** `cx16.r0`

random numbers

rnd ()
Returns next random byte 0-255 from the pseudo-RNG sequence. Does not work in ROM code; use rnd_rom instead.

rnd_rom ()
Returns next random byte 0-255 from the pseudo-RNG sequence. Works in ROM code, but make sure to initialize the seed values using rndseed_rom.

rndw ()
Returns next random word 0-65535 from the pseudo-RNG sequence. Does not work in ROM code; use rndw_rom instead.

rndw_rom ()
Returns next random word 0-65535 from the pseudo-RNG sequence. Works in ROM code, but make sure to initialize the seed values using rndseed_rom.

randrange (ubyte n) -> ubyte
Returns random byte uniformly distributed from 0 to n-1 (compensates for divisibility bias) Does not work in ROM code; use randrange_rom instead.

randrange_rom (ubyte n) -> ubyte
Returns random byte uniformly distributed from 0 to n-1 (compensates for divisibility bias) Works in ROM code, but make sure to initialize the seed values using rndseed_rom.

randrangew (uword n) -> uword
Returns random word uniformly distributed from 0 to n-1 (compensates for divisibility bias) Does not work in ROM code; use randrangew_rom instead.

randrangew_rom (uword n) -> uword
Returns random word uniformly distributed from 0 to n-1 (compensates for divisibility bias) Works in ROM code, but make sure to initialize the seed values using rndseed_rom.

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 either of the seed values! Does not work in ROM code; use rndseed_rom instead.

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

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

log2w (uword v)
Returns the 2-Log of the word value v. **Clobbers:** `cx16.r0`

trigonometry

Hint

This is a graph showing the various ranges of values mentioned in the integer sine and cosine routines that follow below. (Note that the x input value never corresponds to an exact *degree* around the circle 0..359 as that exceeds a byte value. There's double-degrees

though; 0…179) Only the sine function is shown, but the cosine function follows the same pattern.

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. **Clobbers:** `cx16.r0` through `cx16.r4` (used as temporary variables).

sin8u (x)
Fast 8-bit ubyte sine using a lookup table. x = angle 0…2π scaled as 0…255. Result is unsigned, scaled as 0…255

sin8 (x)
Fast 8-bit byte sine using a lookup table. x = angle 0…2π scaled as 0…255. Result is signed, scaled as -127…127

sinr8u (x)
Fast 8-bit ubyte sine using a lookup table. x = angle 0…2π scaled as 0…179 (so each value increment is a 2° step). Result is unsigned, scaled as 0…255. Input values 180…255 lie outside of the valid input interval and will yield a garbage result!

sinr8 (x)
Fast 8-bit byte sine using a lookup table. x = angle 0…2π scaled as 0…179 (so each value increment is a 2° step). Result is signed, scaled as -127…127. Input values 180…255 lie outside of the valid input interval and will yield a garbage result!

cos8u (x)
Fast 8-bit ubyte cosine using a lookup table. x = angle 0…2π scaled as 0…255. Result is

unsigned, scaled as 0…255

cos8 (x)
Fast 8-bit byte cosine using a lookup table. x = angle 0…2π scaled as 0…255. Result is signed, scaled as -127…127

cosr8u (x)
Fast 8-bit ubyte cosine using a lookup table. x = angle 0…2π scaled as 0…179 (so each value increment is a 2° step). Result is unsigned, scaled as 0…255. Input values 180…255 lie outside of the valid input interval and will yield a garbage result!

cosr8 (x)
Fast 8-bit byte cosine using a lookup table. x = of angle 0…2π scaled as 0…179 (so each value increment is a 2° step). Result is signed, scaled as -127…127. Input values 180…255 lie outside of the valid input interval and will yield a garbage result!

There is no `tan` function in this library (there is a floating point `tan` though in the `floats` library).

8.3.18 monogfx (cx16 and virtual)

Full-screen lores or hires monochrome bitmap graphics routines, available on the X16 machine only.

- two resolutions: lores 320*240 or hires 640*480 bitmap mode
- optimized routines for monochrome (2-color) graphics
- clearing screen, switching screen mode, also back to text mode
- doublebuffering option to avoid flicker
- 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 and the *testmonogfx* example program, to see what's in there.

8.3.19 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.

Some routines may require a colors array as @nosplit (such as fade_step_colors), otherwise wrong colors come out. (this is the same for some kernal routines such as cx16.FB_set_palette)

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

8.3.20 petsnd (PET only)

Make sound on the Pet, without locking up the program in a busy loop: it uses the VIA timer as an oscillator.

sub on()
enable sound

sub off()
disable sound

sub octaves(ubyte octs)
set octave(s), choice from range 1 to 3. (1=octaves 4,5,6, 2=octaves 5,6,7, 3=octaves 6,7,8)

sub note(ubyte note)
play the given note. Use `REST` (value 0) for a silence rest. Note constants are available in the module as well, such as A_4, A_5, F_SHARP_5 etc etc

For non-blocking sequenced playback (driven by an IRQ handler):

sub song(^^ubyte notes, ^^ubyte durations, ubyte length)
Prepare a note/duration array pair for non-blocking sequenced playback. The `durations` array holds the total ticks per note slot (note-on + gap). Call this to set up the sequencer, then call `update()` periodically (e.g. from a vsync IRQ handler) to advance through the notes.

sub set_gap(ubyte ticks)
Set the number of silence ticks between notes (default 1). `0` means no silence gap at all —notes flow into each other seamlessly. The note-on time is automatically adjusted: `note = duration - gap`, minimum 1 tick. This means changing the gap does **not** affect the overall tempo. Call this before `song()`.

sub update() -> bool
Advance the sequencer by one tick. Always returns `true` to facilitate chaining to the system IRQ handler. The sequencer calls `note()` and `off()` internally as needed. See `examples/pet/music.p8` for a complete IRQ-driven example.

sub is_playing() -> bool
Returns true if a song is currently being played back via the sequencer.

Blocking convenience helpers:

sub play_note(ubyte note, ubyte ticks)
Play a single note for a given duration in jiffy ticks (blocking). Requires `on()` to have been called before.

sub play_song(^^ubyte notes, ^^ubyte durations, ubyte length)
Play a sequence of notes with given durations (blocking). Uses `song()` + `update()` internally with vsync timing. Requires `on()` to have been called before.

8.3.21 petgfx (PET, C64, C128)

To draw “graphics”at double the resolution of default text mode while using only Petscii block characters.

sub hline(ubyte x, ubyte y, ubyte length)
Plots a horizontal line starting at x,y and with given length

sub vline(ubyte x, ubyte y, ubyte length)
Plots a vertical line starting at x,y and with given length

sub plot(ubyte x, ubyte y)
Plots a "petscii subpixel"using petscii block characters at position x,y in the text screen where x and y can be double the size of the text screen (so 80 and 50, instead of 40 and 25).

8.3.22 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.

8.3.23 psg (cx16 only)

Available for the Cx16 target. **Note: New code should probably use the psg2 module instead!** Contains a simple abstraction for the Vera's PSG (programmable sound generator) to play simple waveforms. It includes an interrupt handler routine for handling automatic ASR volume envelopes as well.

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

8.3.24 psg2 (cx16 only)

Available for the Cx16 target. Contains an abstraction for the Vera's PSG (programmable sound generator) to play simple waveforms. It has better consistent envelope timings than the older `psg` module, and easier access to all voice parameters if desired. It includes an interrupt handler routine for handling automatic ASR volume envelopes as well.

Envelope timing (one tick = one call to `update()`):

- Attack and Release are linear volume ramps: `duration_ticks = ceil(256 / speed)`
- Higher speed = faster ramp (shorter duration)
- speed=1 -> 256 ticks; speed=4 -> 64 ticks; speed=255 -> 1 tick
- Even speed=255 applies on the *next* call to `update()` —there is always at least ~1 frame of delay if you depend on the normal update cycle.
- For truly immediate changes (0ms), set the voice struct fields and call `update()` directly.
- Sustain holds for `speed` ticks then transitions to Release
- speed=0 stalls at that phase (useful for infinite sustain)
- Actual real-time duration depends on how often `update()` is called. If called every vsync at 60Hz, each tick is ~16.7ms.

Available routines:

init ()
Initialize the module, set all voices to default off state.

off ()
Turn off all voices immediately, then call `update` automatically.

voice (voice_num, channels, volume, waveform, pulsewidth)
Set all parameters for a voice (frequency unchanged). Disables any active envelope.

frequency (voice_num, freq)
Set frequency word for the voice. This is the raw VERA PSG register value (no scaling, written directly to the hardware): `freq = HERZ(Hz) / 0.3725290298461914`.

volume (voice_num, vol)
Set volume directly (0-63). Disables any active envelope. Also sets the max volume for subsequent envelope use.

envelope (voice_num, attack, sustain, release)
Start ASR envelope on the voice (volume starts from 0). `attack` and `release` are speeds 0-255, `sustain` is duration in ticks. Speed 255 means "instant".

getvoice (voice_num) -> ^Voice
Return pointer to a voice's parameter struct (for direct field access).

update () -> bool
Advance all active envelopes by one tick, then write all 16 voices to the VERA PSG registers. This call is IRQ-safe!

Waveform constants: `PULSE`, `SAWTOOTH`, `TRIANGLE`, `NOISE` Channel constants: `LEFT`, `RIGHT`, `BOTH`, `DISABLED`

See the examples/cx16/bdmusic.p8 program for ideas how to use it.

Read the psg2 source code to see everything that's in there.

8.3.25 serial (cx16 only)

Routines for the serial/wifi card of the Commander X16. Supports up to 2 UART chips. The wifi functionality is handled via the ZiModem (ESP32) command set.

Generic UART routines:

sub detect_uarts() -> uword, uword
Scans the I/O address range for UART chips and returns the addresses of up to two discovered UARTs (or 0 if none found).

sub get_baud_string(uword baud) -> str
Returns a readable string representation of the given baud rate (use values from the BAUD enum).

sub initialize_uart(uword uart_addr, uword baud_divisor)
Initializes the given UART with auto flow control and FIFOs enabled. 'baud_divisor'is the divisor calculated based on the desired baud rate (use values from the BAUD enum).

sub write(uword uart_addr, str data)
Writes the string data to the UART (terminated by 0 byte)

sub read_until(uword uart_addr, str match, ^^ubyte buffer, uword max_size) -> uword
Reads bytes from the UART into buffer until the match string is found (the match string itself is also included in the buffer) or max_size bytes have been read. Returns the number of bytes read.

sub discard_until(uword uart_addr, str match)
Reads and discards bytes from the UART until the match string is found (the match string itself is also discarded).

ZiModem wifi routines:

sub zi_initialize(uword uart_addr)
Initializes the ZiModem on the given UART address.

sub zi_reset()
Resets the ZiModem connection (`atz`).

sub zi_write_cmd(str command)
Writes the given command to the ZiModem followed by CR/LF.

sub zi_start_get_file(str filename) -> long
Starts a file download from the given URL and returns the file size. Use `zi_get_file_chunk()` repeatedly to download the data. Note: binary mode transfer can be unreliable; prefer hex mode.

sub zi_start_get_file_hexmode(str filename) -> bool
Starts a file download in hex mode from the given URL. Returns true if the file was found, false otherwise. Use `zi_get_file_chunk_hexmode()` repeatedly to download chunks. Hex mode is more reliable than binary mode for file transfers.

sub zi_get_file_chunk(^^ubyte buffer, uword buffer_size, long remaining_file_size) -> uword
Reads the next chunk of data (up to buffer_size bytes) from an active file download started with `zi_start_get_file()`. Returns the number of bytes read, or 0 when done.

sub zi_get_file_chunk_hexmode(^^ubyte buffer, uword buffer_size) -> uword
Reads the next chunk of hex-encoded data (up to buffer_size bytes) from an active file download started with `zi_start_get_file_hexmode()`. Buffer must be at least 40 bytes. Returns the number of bytes decoded, or 0 when done.

sub zi_end_get_file()
Consumes the trailing OK response after downloading a file with `zi_start_get_file()` / `zi_get_file_chunk()`.

sub zi_get_ip_address() -> str
Returns the IP address of the ZiModem wifi connection.

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

8.3.26 sorting

Various sorting routines (gnome sort and shell sort variants) for byte, word and string arrays. **NOTE:** all word and str arrays have to be @nosplit! Words and pointers need to be consecutive in memory for now. **NOTE:** sorting is done in ascending order. Read the sorting source code to see what's in there. Also check out the *sortingbench* example.

8.3.27 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.

8.3.28 strings

Provides string manipulation routines.

conversion and classification

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

isxdigit (char)
Returns boolean if the character is a hexadecimal digit 0-9, a-f, or A-F.

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)

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

lower_iso (string)
Lowercases the ISO-string in place.

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

upper_iso (string)
Uppercases the ISO-string in place.

lowerchar (char)
Returns lowercased PETSCII character.

lowerchar_iso (char)
Returns lowercased ISO character.

upperchar (char)
Returns uppercased PETSCII character.

upperchar_iso (char)
Returns uppercased ISO character.

miscellaneous

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

compare_nocase (string1, string2) -> ubyte result
Returns -1, 0 or 1 depending on whether string1 sorts before, equal or after string2 (case-insensitively, for petscii strings).

compare_nocase_iso (string1, string2) -> ubyte result
Returns -1, 0 or 1 depending on whether string1 sorts before, equal or after string2 (case-insensitively, for iso strings).

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`!

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.

manipulation

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

copy (from, to) -> ubyte length
Copy a string to another, overwriting that one. Make sure it was large enough to contain the new string. Returns the length of the string that was copied.

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 (which includes a terminating 0 byte). Length must be smaller or equal to the length of the source string. Writes in-place; doesn't return a value (so can't be used in an expression).

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)

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)

nappend (string, suffix, maxlength) -> ubyte length
Appends the suffix string to the other string, up to the given maximum length of the combined string. Returns the length of the combined string.

ncopy (from, to, maxlength) -> ubyte length
Copy a string to another, overwriting that one, but limited to the given length. Returns the length of the string that was copied.

next_token(str source, str delimiters) -> str
Tokenize the source string according to the list of delimiter characters. Like C's `strtok` function. You pass in the string to tokenize and the delimiters (make sure the delimiters end with a trailing 0 as well). The routine returns a pointer to the next token (or first token, if you pass in a new string). To get the next tokens, keep calling the routine but pass 0 for the source string (which tells it to continue processing the previous string). It returns 0 when there are no more tokens.

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 (which includes a terminating 0 byte). Length must be smaller or equal to the length of the source string. Writes in-place; doesn't return a value (so can't be used in an expression).

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

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

slice (source, start, length, target)
Copies a segment from the source string, starting at the given index, and of the given length, to the target string. It is assumed the target string buffer is large enough to contain the result (which includes a terminating 0 byte). Start and length must be within bounds of the source string. Writes in-place; doesn't return a value (so can't be used in an expression).

split (string, parts, max_parts) -> ubyte
Splits string into parts separated by white space (destructive). Pointers to each part are stored in the given parts array (sequential uwords, for instance in a @nosplit uword array), up to the given maximum number of parts. Returns the number of parts stored. Leading and trailing whitespace has been stripped from each part.

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

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

searching

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"`.

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

find (string, char) -> ubyte index, bool found
Locates the first index of the given character in the string, and a boolean (in Carry flag) to say if it was found at all. If the character is not found, index 255 (and false) is returned. You can consider this a safer way of checking if a character occurs in a string than using an *in* containment check - because this find routine properly stops at the first 0-byte string terminator it encounters in case the string was modified.

pattern_match (string, pattern) -> bool (not on Virtual target)
Returns true if the string matches the pattern, false if not. The matching is CASE-SENSITIVE. If you want to match case-insensitively, use pattern_match_nocase(). '?' in the pattern matches any one character. '*'in the pattern matches any substring. An empty pattern matches nothing. If you need everything to match, use a single '*'. Note: this routine does not work when it is located in ROM.

pattern_match_nocase (string, lowercase_pattern, iso_encoding) -> bool (not on Virtual target)
Returns true if the string matches the pattern, false if not. The matching is CASE-

INSENSITIVE (based on ISO encoding if iso_encoding is true, otherwise based on PETSCII encoding). If you want to match a bit faster but case-sensitively, use pattern_match(). '?'in the pattern matches any one character. '*'in the pattern matches any substring. An empty pattern matches nothing. If you need everything to match, use a single '*'. Note: this routine does not work when it is located in ROM.

rfind (string, char) -> ubyte index, bool found
Like `find`, but now looking from the *right* of the string instead.

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

8.3.29 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` blocks depending on the chosen compilation target. Read the sys lib source code for the correct compilation target to see exactly what is there.

8.3.30 sys (part of syslib)

miscellaneous

cpu_is_65816()
Returns true if the CPU in the computer is a 65816, false otherwise (6502 cpu). Note that Prog8 itself has no support yet for this CPU other than detecting its presence.

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

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!

`memcmp (address1, address2, size)`
Compares two blocks of memory of up to 65535 bytes in size. Returns -1 (255), 0 or 1, meaning: block 1 sorts before, equal or after block 2.

`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. On the assembly level: it returns the address of the symbol "`prog8_program_end`".

`progstart ()`
Returns the first address of the program in memory. This usually is $0801 on the C64 and the X16, for example. On the assembly level: it returns the address of the symbol "`prog8_program_start`".

`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)

`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.

`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:

- 7 = Neo6502
- 8 = Atari 8 bits
- 16 = Commander X16
- 25 = Foenix F256 family
- 32 = Commodore PET
- 64 = Commodore 64
- 128 = Commodore 128
- 255 = Virtual machine

`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.

waitirq () (cx16 only)
: efficiently wait until the next interrupt has occurred (any source). It uses the 65C02 'wai' instruction for this.

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.

processor status flags

clear_carry ()
: Clears the CPU status register Carry 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).

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

set_carry ()
: Sets the CPU status register Carry flag.

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

processor stack

Pushing and popping values on the CPU hardware stack can be done via the stack related *Built-in Functions* (pop, push etc). There is a very specialized function in the sys module here as well:

push_returnaddress (address)
: pushes a 16 bit memory address on the CPU hardware stack in the same byte order as a JSR instruction would, which means the next RTS instruction will jump to that address instead. You cannot use pushw() for this because the bytes pushed by JSR are different

8.3.31 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: `print`, `chrout`, `nl`, `print_ub` etc.
- reading text input from the user via the keyboard: `input_chars`
- 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.

There are *a lot of routines* in this module, many more than mentioned above. Read the textio source code to see what's in there. (Note: slight variations for different compiler targets)

8.3.32 verafx (cx16 only)

Available for the Cx16 target. Routines that use the Vera FX logic to accelerate certain operations.

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

`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.

`muls`
: The VeraFX signed word 16*16 to 32 multiplier is accessible via the `muls` routine. It is about 4 to 5 times faster than the default 6502 cpu routine for word multiplication. But it depends 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, but be careful with it because it may interfere with other Vera operations or IRQs. The full 32 bits result value is returned as a long.

`muls16`
: Like `muls` but only returns the lower word of the result, which is sometimes useful if you're just interested in word values.

`mult16`
: VeraFX hardware multiplication of two unsigned words. NOTE: it only returns the lower 16 bits of the full 32 bits result, because the upper 16 bits are not valid for unsigned word multiplications here (the signed word multiplier `muls` does return the full 32 bits result). It is about 4 to 5 times faster than the default 6502 cpu routine for word multiplication. But it depends 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, but be careful with it because it may interfere with other Vera operations or IRQs.

transparency
Set transparent write mode for VeraFX cached writes and also for normal writes to DATA0/DATA. If enabled, pixels with value 0 do not modify VRAM when written (so they are "transparent")

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

8.3.33 wavfile

sub parse_header(^^ubyte wav_data) -> bool
Parses the header of a .wav file. Returns true if it's a valid wav file, false if is invalid.

These header fields can be read out afterwards:

```
uword sample_rate
ubyte bits_per_sample
uword data_offset
ubyte wavefmt           ; WAVE_FORMAT_PCM=1, WAVE_FORMAT_DVI_ADPCM=17,␣
↪etc
ubyte nchannels
uword block_align
long data_size
```

Note

the sample rate in hertz can be converted to a vera rate value via:

```
const float vera_freq_factor = 25e6 / 65536.0
vera_rate = (wavfile.sample_rate as float / vera_freq_factor) + 1.0 as␣
↪ubyte
vera_rate_hz = (vera_rate as float) * vera_freq_factor as uword
```

CHAPTER

NINE

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
- 'pet32': the Commodore PET 4032
- 'virtual': a builtin virtual machine
- custom targets via a separate configuration file (see *Customizable targets*)

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)!

Note

When you specify a file name as target, prog8 will try to read the target machine's configuration and properties from that configuration file instead. See *Customizable targets* for details about this.

9.1 Customizable targets

You can also specify a file name instead of one of the built in target machine names, when using the `-target` option. In this case the compiler will not use one of the builtin machines configurations, but read it from the configuration file. This allows you to define and change

your own target machine configuration, and maybe allow Prog8 to generate programs for new machines or existing ones it doesn't yet know about.

The configuration file should be a "properties"file (Java's ubiquitous configuration file format), which is basically a text file containing "key=value"lines. The contents of the file is pretty extensive and it's easier to just look at some examples that are already included: target configuration examples . The filename base part (the part before any suffix) of the target configuration file will be taken as the name of the compilation target. If it matches one of the existing built-in compilation targets, the internal library files for that target will also be searched, if the user supplied library location doesn't contain a replacement library file for anything that might get imported. The path you need to provide for the `library` variable can be relative (to the current working directory where you launch the compiler from) and you can use a tilde ~ in it like in a shell path, to refer to a user's home directory. Note that library modules not unique to a specific compilation target (for example, *buffers*, *sorting* or *strings*) will be picked up from the internal library files just fine as was always the case. You can still provide custom versions of them in your own library folder of course, like you already could with using the `-srcdirs` compiler flag.

Most of the things discussed in the *Porting Guide* can and must be configured properly in the target configuration file. You also need to create some essential `syslib` library module for the configured target if its name does not match one of the built in targets. This is because the compiler won't have a built in library that can be used this time. The customtarget examples also show how to build the essentials.

The target configuration file also allows you to override the entire launcher assembly fragment that the compiler usually puts at the start of the program (immediately after the setting of the program counter/load address and the program start label). This allows you to have full control over exactly what code or data bytes appear at the program's load address. But with great power comes great responsibility: you'll have to make sure you take care of some very low level Prog8 internals if you want to support stuff such as the `exit` builtin function. Maybe the best thing is to look at the code generated by the compiler for an existing target and adapt that.

9.2 Memory Model

9.2.1 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

9.2.2 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. See *ROM/RAM bank selection* for details about that.

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. Note: these virtual registers are not automatically initialized to zero at program startup.

$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

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

Program RAM / BASIC ROM $A000-$BFFF
On the C64 the Basic ROM normally occupies this memory area. However Prog8 programs that do not use floating point variables, actually bank out the Basic ROM to reclaim the 8 Kb of RAM that is hidden below it. This means that all the memory from $0801 to $D000 (exclusive) is available as program ram to Prog8 programs.

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.

Footnotes for the Commodore 128

Golden Ram $1300 - $1BDF
Application RAM area, free to use if you don't use BASIC variables or strings. Note: $1BE0 - $1BFF are used for the 16 virtual registers cx16.r0 .. cx16.r15. There is no "high ram"defined on the C128.

Program RAM $1C00 - $BFFF
On the C128 the Basic ROM is banked out by default, reclaiming the RAM area from $1C00 to $BFFF. This gives about 41 Kb of contiguous RAM for Prog8 programs.

9.2.3 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!

9.3 CPU

9.3.1 Directly Accessible Registers

The hardware CPU registers (A, X, Y) are not directly accessible from regular Prog8 code. If you need to work with them, you'll have to use some inline assembly with `%asm`. Or, if they are required to have a value as arguments to some external kernal or library assembly routine, just use a normal subroutine call to an `extsub` that correctly specifies what registers go where. The compiler will then take care of loading the arguments into the required registers and returning any response value(s) back to the prog8 code.

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. With the special status branch statements like `if_cc`, `if_cs` etc you can branch directly on the status of the flags.

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 from everywhere.

9.4 IRQ Handling (general)

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).

9.4.1 High-level convenience routines

On the C64, C128, PET32 and CommanderX16 targets there are a few library routines available to make setting up 60hz/vsync IRQs a lot easier (no assembly code required). These routines are:

```
sys.set_irq(uword handler_address)
sys.restore_irq()          ; set everything back to the systems default irq␣
↪handler
sys.set_rasterirq(uword handler_address, uword rasterline)    (not on PET32)
sys.update_rasterirq(uword handler_address, uword rasterline) (not on PET32)
sys.set_rasterline(uword rasterline)                          (not on PET32)
```

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.

Note

The PET32 target only supports `sys.set_irq()` and `sys.restore_irq()`. The raster-related routines are not available on PET32.

9.4.2 Low-level bare IRQ handler

Some targets allow you to install a "bare"IRQ handler directly into the system's interrupt vector, bypassing the high-level `sys.set_irq()` convenience routines. A bare handler is just a subroutine that is called directly by the CPU on each interrupt –there is no register-saving wrapper around it.

On Commodore targets (C64, C128, PET32, Commander X16) you install such a handler by writing to the `cbm.CINV` RAM vector:

```
uword saved_irq = cbm.CINV    ; save old handler address (may be $0000 if␣
↪none)

sys.set_irqd()
cbm.CINV = &my_handler        ; install our handler
sys.clear_irqd()
```

Other targets may use a different vector or may not support bare handlers at all.

Caution

On Commodore targets, the KERNAL saves A, X, Y before dispatching through the CINV vector, so a bare handler does not need to save/restore them. However, the handler **must** save and restore the compiler's internal zero-page scratch registers via `sys.save_prog8_internals()` / `sys.restore_prog8_internals()` —the KERNAL does not know about those, and the handler's own Prog8 code may corrupt them.

The handler must also chain to the previous handler to preserve system services (jiffy clock, keyboard scan, cursor blink). A correct bare handler on Commodore looks like this:

```
uword saved_irq

sub my_handler() {
    sys.save_prog8_internals()

    ; ... your IRQ handler code here ...

    sys.restore_prog8_internals()
    goto saved_irq      ; chain to previous handler
}
```

Make sure to declare `saved_irq` as a `uword` variable in the same scope, and save the current value of the interrupt vector into it before overwriting it.

Caution

Be cautious about calling ROM routines within an interrupt handler. Some kernal routines are fine to call, others can be problematic. The safest approach is just to modify a flag variable in the handler and act on that flag in the regular main loop of the program (i.e. call the required kernal routines there, and then reset the flag).

Caution

It is 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...

Caution

The Commander X16's sixteen 'virtual registers'R0-R15 *are not preserved* in the IRQ handler! (On any system!) 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. Note that Prog8 itself may also use these registers, so be very careful. This is not a X16 specific thing; these registers also exist on the other compiler targets, and the same issue holds there.

There are two utility routines in cx16 that save and restore *all* 16 registers. It's a bit inefficient if only a few are clobbered, but it's easy to put calls to them into your IRQ handler routine at the start and end. These routines are `cx16.save_virtual_registers()` and `cx16.restore_virtual_registers()`.

9.5 Commander X16 specific IRQ handling

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 paragraph below for perhaps a better and easier solution that is tailored to this system.

Caution

When Using VERA in the handler: 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... see caution below
; ... do your work that uses vera here!...
; perhaps also cx16.restore_virtual_registers() here... see caution below
cx16.restore_vera_context()
```

Multi-IRQ support routines

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. *All cautions mentioned above, regarding the VERA and Virtual registers, floating point and kernal routines, still apply!*

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, TIMER1(VIA1), VSYNC, SPRCOL, AFLOW.

`set_vsync_irq_handler (uword address)`
: Sets the vertical 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. Note: the handler must fill the Vera's audio fifo buffer by itself with at least 25% worth of data (1 kb) otherwise the aflow irq keeps triggering.

`set_timer1_irq_handler (uword address)`
: Sets the VIA1 TIMER1 irq handler to use with enable_irq_handlers(). Does not enable or disable VIA1 timer irqs setting.

`disable_irq_handlers ()`
: Hand control back to the system default IRQ handler.

And a utility method to set the VIA1 timer1 to trigger IRQs at regular intervals:

sub set_timer1 (uword delay, bool keeprunning)

Set VIA1 timer1 to trigger after the given delay (cycles). Enables VIA timer1 irqs if delay>0, otherwise disables it. If keeprunning then the timer keeps triggering, otherwise it stops after a single trigger. Note that the speed of the timer depends on the clock speed of the X16 (controlled by a jumper on the motherboard, usually 8 MHz).

CHAPTER

TEN

TECHNICAL DETAILS

10.1 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.

10.2 ROM/RAM bank selection

On certain systems prog8 provides support for managing the ROM or RAM banks that are active.

system	get banks (returns byte)	set banks
c64	`c64.getbanks()`	`c64.banks(x)`
c128	`c128.getbanks()`	`c128.banks(x)`
cx16	`cx16.getrombank()`, `cx16.getrambank()`	`cx16.rombank(x)`, `cx16.rambank(x)`, `cx16.push_rombank(x)`, `cx16.pop_rombank()`, `cx16.push_rambank(x)`, `cx16.pop_rambank()`
other	N/A	N/A

Calling a subroutine in another memory bank can be done by using the `callfar` or `callfar2` builtin functions.

When you are using the routines above, you are doing explicit manual banks control. However, Prog8 also provides something more sophisticated than this, when dealing with banked subroutines:

External subroutines defined with `extsub` can have a non-standard ROM or RAM bank specified as well. The compiler will then transparently change a call to this routine so that the correct bank is activated automatically before the normal jump to the subroutine (and switched back on return). The programmer doesn't have to bother anymore with setting/resetting the banks manually, or having the program crash because the routine is called in the wrong bank! You define such a routine by adding `@bank <bank>` to the extsub subroutine definition. This specifies the bank number where the subroutine is located in. You can use a constant bank number 0-255, a ubyte variable, or even the name of a subroutine (must be parameterless, and returning a ubyte) to make it dynamic:

```
extsub @bank 10  $C09F = audio_init()
extsub @bank banknr  $A000 = first_hiram_routine()
extsub @bank get_bank  $A000 = second_hiram_routine()
```

When a subroutine is used as a banking routine, the compiler will call it just before the actual banked subroutine is invoked. The return value of the banking routine (in register A) is then used as the bank number for the subsequent call. This might be useful for implementing **dynamic overlay loading**, where a banking routine can check if the required code is already loaded in a certain bank, load it from disk if necessary, and then return the bank number to the banked subroutine call.

When you then call this routine in your program as usual, the compiler will no longer generate a simple JSR instruction to the routine. Instead it will generate a piece of code that automatically switches the ROM or RAM bank to the correct value, does the call, and switches the bank back. The exact code will be different for different compilation targets, and not all targets even have banking or support this. As an example, on the Commander X16, prog8 will use the JSRFAR kernal routine for this. On the Commodore 128, a similar call exists (but requires a lot more code to prepare, so beware). On the Commodore 64 some custom code is also emitted that toggle the banks, retains some registers, and does the call. Other compilation targets don't have banking or prog8 doesn't yet support automatic bank selection on them.

There's a "banking"example for the Commander X16 that shows a possible application of the extsub with bank support, check out the bank example code .

Notice that the symbol for this routine in the assembly source code will still be defined as usual. The bank number is not translated into assembly (only as a comment):

```
p8s_audio_init = $c09f ; @bank 10
```

Caution

Calls with automatic bank switching like this are not safe to use from IRQ handlers. Don't use them there. Instead change banks in a controlled manual way (or not at all).

Note

On the C64 and C128, the Basic ROM is *banked out* by default when running a Prog8 program, because it is not needed. This means on the C64 we get access to another 8Kb of RAM at that memory area, which actually gives us a 50 Kb contiguous RAM block from

$0801 to $d000 (exclusive). This means you can create programs of up to **50 Kb** size with prog8 on the C64. On the C128, it means programs can use ~41 Kb of contiguous RAM at $1c00 to $c000 (exclusive). However, if your program uses floats, Prog8 *does* need the Basic ROM for the floating point routines, and it won't be banked out. Such programs are limited to the regular size of about 38 Kb on the C64, and less on the C128. Be aware that the bank setting is only done if you are *not* using `%option no_sysinit` because the program's bootstrap code is what initializes the memory bank configuration.

10.3 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_`
Struct	`p8t_`
Struct Field	`p8v_`
Enum Member	`p8c_EnumName_MemberName`
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
}}
```

10.4 Subroutine Calling Conventions

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, longs, booleans, floats), but array types passed by reference which means as `uword` being a pointer to their address in memory. Strings are passed as a pointer to a byte: `^^ubyte`.
2. calling the subroutine

3. preparing the return value (if any) and returning that from the call.

There is no stack handling involved: Prog8 doesn't have call stack frames.

10.4.1 Regular subroutines

- Each subroutine parameter is represented as a variable scoped to the subroutine. Prog8 doesn't have a call stack.
- The arguments passed in a subroutine call are evaluated by the caller, and then put into those variables by the caller. The order of evaluation of subroutine call arguments *is unspecified* and should not be relied upon.
- The subroutine is invoked.
- The return value is not put into a variable, but the subroutine passes it back to the caller via register(s). See below.

Builtin functions can be different:

some builtin functions are special and won't exactly follow the rules in this paragraph.

Single arguments will often be passed in registers:

For *single* byte, word, and pointer arguments (not long or float), the values are simply loaded in cpu registers by the caller before calling the subroutine. *The subroutine itself will take care of putting the values into the parameter variables.* This saves on code size because otherwise all callers would have to store the values in those variables themselves. Note that this convention is also still used for subroutines that specify parameters to be put into one of the *virtual registers* R0-R15, as those are in the end just variables too (see *Reusing virtual registers R0-R15 for parameters*) The rules are as follows:

Single byte parameter: `sub foo(ubyte bar) { ... }`
gets bar in the accumulator A, *subroutine* stores it into parameter variable

Two byte parameters: `sub foo(ubyte bar, ubyte baz) { ... }`
gets bar in the accumulator A, and baz in Y, *subroutine* stores it into parameter variable

Single word parameter: `sub foo(uword bar) { ... }`
gets bar in the register pair A + Y (lsb in A, msb in Y), *subroutine* stores it into parameter variable

Single pointer parameter: `sub foo(^^ubyte bar) { ... }`
gets bar in the register pair A + Y (lsb in A, msb in Y), *subroutine* stores it into parameter variable

Long or Floating point parameter: `sub foo(long bar) { ... }`, `sub foo(float bar) { ... }`
value for bar gets stored into the parameter variable *by the caller*

Other: `sub foo(ubyte bar, ubyte baz, ubyte zoo) { ... }`
not using registers; all values get stored in the subroutine's parameter variables *by the caller*

Return value

- A byte return value will be put in `A` .
- A boolean return value will be put in `A` too, as 0 or 1.
- A word return or pointer value will be put in `A` + `Y` register pair (lsb in A, msb in Y).

- A long return value will be put into `cx16.r14` and `cx16.r15` (2 word virtual registers, low word and high word, to make up a single 32 bits long)
- A float return value will be put in the `FAC1` float 'register'.

In case of *multiple* return values:

- for an `asmsub` or `extsub` the subroutine's signature specifies the output registers that contain the values explicitly, just as for a single return value.
- for regular subroutines, the compiler will return the first of the return values via the cpu register `A` (or `A + Y` if it's a word value), just like for subroutines that only return a single value. The remainder of the return values are returned via the "virtual registers" cx16.r16-cx16.r0 (using R15 first and counting down to R0). Long values will take a pair of those "virtual registers"that combined make up a single 32 bits value. A floating point value is passed via FAC1. Multiple float return values are supported on the virtual target, but limited to a single float on 6502 targets (because the ROM float routines use FAC1/FAC2 as operand registers which would clobber earlier return values).

10.4.2 `asmsub` and `extsub` routines

These are kernal (ROM) routines or low-level assembly routines, that get their arguments via specific registers. Sometimes even via a processor status flag such as the Carry flag. Note that word values can be put in a "CPU register pair"such as AY (meaning A+Y registers) but also in one of the 16 'virtual'16 bit registers introduced by the Commander X16, R0-R15. Float values can be put in the FAC1 or FAC2 floating point 'registers'. The return values also get returned via designated registers, or via processor status flags again. This means that after calling such a routine you can immediately act on the status via a special branch instruction such as `if_z` or `if_cs` etc. The register/status flag usage is fully specified in the asmsub or extsub signature definition for both the parameters and the return values:

```
extsub $2000 = extfunction(ubyte arg1 @A, uword arg2 @XY, uword arg3 @R0,
                           float frac @FAC1, bool flag @Pc) -> ubyte @Y, bool␣
↪@Pz

asmsub function(ubyte arg1 @A, uword arg2 @XY, uword arg3 @R0,
                float frac @FAC1, bool flag @Pc) -> ubyte @Y, bool @Pz {
    %asm {{
        ...
        ...
    }}
}
```

10.5 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 simplified 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) Note that this IR is still *targeted to one specific compilation target only*; various properties and all library code for the selected target machine is encoded into the IR. It is *not possible* to eventually create a C64 program from an IR file created for the CommanderX16 target.
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.

10.6 ROM-able programs

Normally Prog8 will use some tricks to generate the smallest and most optimized code it can. This includes the following techniques that by default prevent generated program code from running in ROM:

self-modifying code
: This is program code that actually modifies itself during program execution (instructions or operands are modified) When the program is in ROM, such modifications are impossible, so the program will not execute correctly.

inline variables
: These are variables that are located in the same memory region that the program code is in (or even interleaved within the program code). Again, writing to such variables will not work when it is in ROM, so the program will not execute correctly.

(Not all prog8 source code will end up using these techniques but you should not depend on it.)

The directive `%option romable` changes this behavior. It tells the compiler to no longer generate code using these two tricks, and instead revert to slightly slower running code (or needing more instructions) but which *is* able to run from ROM. There are a few things to note:

- string variables and array variables that are initialized with something other than just zeros, *are no longer mutable*. This is because both of these will still end up as part of the same memory region the program code is in (which will be ROM). The compiler will try to detect writes to them and give an error if these occur. However it cannot detect all such writes, so beware.
- arrays without an initialization literal will be placed into the memory region for variables instead which can and should be placed in RAM, so those arrays *are* mutable as usual.
- the same holds for memory blocks allocated using the `memory` function; nothing changes for them.
- the memory region for variables and memory blocks (BSS sections) should be explicitly placed in RAM memory. You can do this with the `-varsgolden` or `-varshigh`, and `-slabsgolden` or `-slabshigh` command line options. Maybe in the future an option will be added to choose a memory address for those manually.

> **Note**
>
> The ROMable code generation is still quite experimental. Problems may still surface, and perhaps stuff will change a bit in a future compiler version.

10.7 Formal ANTLR4 syntax and grammar definition

```
/*
Prog8 combined lexer and parser grammar

NOTES:

- whitespace is ignored. (tabs/spaces)
- every position can be empty, be a comment, or contain ONE statement.

*/

// -> java classes Prog8ANTLRParser and Prog8ANTLRLexer,
// both NOT to be used from Kotlin code, but ONLY through Kotlin class␣
 ↪Prog8Parser
grammar Prog8ANTLR;

@header {
package prog8.parser;
}

EOL :  ('\r'? '\n' | '\r' | '\n')+ ;
LINECOMMENT : EOL [ \t]* COMMENT -> channel(HIDDEN);
COMMENT :  ';' ~[\r\n]* -> channel(HIDDEN) ;
BLOCK_COMMENT : '/*' ( BLOCK_COMMENT | ~'*' | '*' ~'/' )*? '*/' -> skip ;

WS :  [ \t] -> skip ;
// WS2 : '\\' EOL -> skip;
VOID: 'void';
STRUCT: 'struct';
ON: 'on';
GOTO: 'goto';
CALL: 'call';
INLINE: 'inline';
PRIVATE: 'private';
STEP: 'step';
ELSE: 'else';
THEN: 'then';
ENUM: 'enum';

UNICODEDNAME :  [\p{Letter}]([\p{Letter}\p{Mark}\p{Digit}_] | '::')* ;          ␣
 ↪   // match unicode properties
UNDERSCORENAME :  '_' UNICODEDNAME ;           // match unicode properties
DEC_INTEGER :  DEC_DIGIT (DEC_DIGIT | '_')* ;
HEX_INTEGER :  '$' HEX_DIGIT (HEX_DIGIT | '_')* ;
BIN_INTEGER :  '%' BIN_DIGIT (BIN_DIGIT | '_')* ;
```

(continues on next page)

(continued from previous page)

```
ADDRESS_OF: '&' ;
TYPED_ADDRESS_OF: '&&' ;
ADDRESS_OF_MSB: '&>' ;
ADDRESS_OF_LSB: '&<' ;
POINTER: '^^';

fragment HEX_DIGIT: ('a'..'f') | ('A'..'F') | ('0'..'9') ;
fragment BIN_DIGIT: ('0' | '1') ;
fragment DEC_DIGIT: ('0'..'9') ;

FLOAT_NUMBER : DEC_DIGIT (DEC_DIGIT | '_')* ('.' (DEC_DIGIT | '_')*)? (('E'|'e
↪') ('+'|'-')? DEC_INTEGER)?
             | '.' (DEC_DIGIT | '_')+ (('E'|'e') ('+'|'-')? DEC_INTEGER)?
             ;

STRING_ESCAPE_SEQ :  '\\' [\u0021-\u007E] | '\\x' HEX_DIGIT HEX_DIGIT | '\\u'␣
↪HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
STRING :
    '"' ( STRING_ESCAPE_SEQ | ~[\\\r\n\f"] )* '"'
    ;
INLINEASMBLOCK :
    '{{' .+? '}}'
    ;

SINGLECHAR :
    '\'' ( STRING_ESCAPE_SEQ | ~[\\\r\n\f'] ) '\''
    ;

TAG: '@' ('a'..'z' | '0'..'9')+ ;

EMPTYARRAYSIG : '[' [ \t]* ']' ;

NOT_IN: 'not' [ \t]+ 'in' [ \t] ;


// A module (file) consists of zero or more directives or blocks, in any␣
↪order.
// If there are more than one, then they must be separated by EOL (one or␣
↪more newlines).
// However, trailing EOL is NOT required.
// Note: the parser may see *several* consecutive EOLs - this happens when␣
↪EOL and comments are interleaved (see #47)
module: EOL* (module_element (EOL+ module_element)*)? EOL* EOF;

module_element:
    directive | block ;


block: identifier integerliteral? EOL? '{' EOL? (block_statement | EOL)* '}';

// Note: enum and alias appear in both block_statement and statement rules.
// This is intentional - they are declaration statements like␣
```

(continues on next page)

(continued from previous page)

```
↪variabledeclaration,
// and can appear at block level or inside subroutines (local enums/aliases).
block_statement:
    directive
    | variabledeclaration
    | structdeclaration
    | subroutinedeclaration
    | inlineasm
    | labeldef
    | alias
    | enum
    ;


statement :
    directive
    | ongoto
    | variabledeclaration
    | structdeclaration
    | assignment
    | augassignment
    | unconditionaljump
    | postincrdecr
    | functioncall_stmt
    | if_stmt
    | branch_stmt
    | subroutinedeclaration
    | inlineasm
    | returnstmt
    | forloop
    | whileloop
    | untilloop
    | repeatloop
    | unrollloop
    | whenstmt
    | breakstmt
    | continuestmt
    | labeldef
    | defer
    | alias
    | enum
    | swap
    // Error recovery: match tokens that can NEVER start a valid expression
    | 'if' '{'
      { notifyErrorListeners("Missing condition: expected expression after 'if
↪'"); }
    | 'while' '{'
      { notifyErrorListeners("Missing condition: expected expression after
↪'while'"); }
    | 'for' identifier 'in' '}'
      { notifyErrorListeners("Missing range: expected expression after 'in'");
↪ }
```

(continues on next page)

(continued from previous page)

```
    | 'for' identifier 'in' ('else' | 'return' | 'break' | 'continue' | 'defer
↪' | '}')
      { notifyErrorListeners("Missing range: expected expression after 'in'");
↪ }
    | 'return' ('}' | 'else' | 'while' | 'for' | 'if' | 'defer')
      { notifyErrorListeners("Invalid token after 'return'"); }
    | 'defer' ('}' | 'else' | 'while' | 'for' | 'if' | 'return' | 'break' |
↪'continue')
      { notifyErrorListeners("Expected statement after 'defer'"); }
    | 'when' '{'
      { notifyErrorListeners("Missing expression: expected condition after
↪'when'"); }
    | ENUM '{'
      { notifyErrorListeners("Expected enum name after 'enum'"); }
    | STRUCT '{'
      { notifyErrorListeners("Expected struct name after 'struct'"); }
    | ON (GOTO | CALL)
      { notifyErrorListeners("Missing expression: expected index after 'on'");
↪ }
    ;


enum :  PRIVATE? ENUM identifier '{' EOL? enum_member? (',' EOL? enum_
↪member)* ','? EOL? '}' ;       // you can split the values over several␣
↪lines, trailing comma allowed

enum_member :  identifier ('=' integerliteral)?  ;

swap: 'swap' '(' assign_target ',' assign_target ')' ;


variabledeclaration :
    varinitializer
    | vardecl
    | constdecl
    | memoryvardecl
    ;


structdeclaration:
    PRIVATE? STRUCT identifier '{' EOL? (structfielddecl | EOL)+ '}'
    ;

structfielddecl: datatype (arrayindex arrayindex? | EMPTYARRAYSIG)?␣
↪identifierlist;
// grammar allows [] and [][] so the visitor can give user-friendly error␣
↪messages instead of cryptic parse errors


subroutinedeclaration :
    subroutine
    | asmsubroutine
```

(continues on next page)

(continued from previous page)

```
    | extsubroutine
    ;

alias: PRIVATE? 'alias' identifier '=' scoped_identifier ;

defer: 'defer' (statement | statement_block) ;

labeldef :  identifier ':'  ;

unconditionaljump :  GOTO  expression ;

directive : directivename '!'? (directivenamelist | (directivearg? |␣
↪directivearg (',' directivearg)*))   ;

directivename: '%' UNICODEDNAME;

directivenamelist: '(' EOL? scoped_identifier (',' EOL? scoped_identifier)* ',
↪'? EOL?')' ;

directivearg : stringliteral | identifier | integerliteral ;

vardecl: PRIVATE? datatype (arrayindex arrayindex? | EMPTYARRAYSIG)? TAG*␣
↪identifierlist ;
// grammar allows [] and [][] so the visitor can give user-friendly error␣
↪messages for invalid combinations

identifierlist: identifier (',' identifier)* ;

varinitializer :
    vardecl '=' expression
    | vardecl '=' tuple_expression
    ;

constdecl: PRIVATE? 'const' datatype? identifierlist '=' expression ;
// datatype is optional in the grammar so the visitor can give "datatype␣
↪missing" instead of a cryptic parse error

memoryvardecl: ADDRESS_OF varinitializer;

basedatatype:  'ubyte' | 'byte' | 'uword' | 'word' | 'long' | 'float' | 'str'␣
↪| 'bool' ;

datatype: pointertype | basedatatype | structtype=scoped_identifier;

pointertype: POINTER (scoped_identifier | basedatatype);

arrayindex:
    '[' expression ']' #ArrayIndexNormal
    | '[' expression ',' expression ']' #ArrayIndexComma
    ;
// ArrayIndexComma exists in the grammar so the visitor can give a friendly␣
↪error for the wrong 2D array syntax
```

(continues on next page)

(continued from previous page)

```
assignment :
    assign_target '=' expression
    | assign_target '=' assignment
    | multi_assign_target '=' expression
    | multi_assign_target '=' tuple_expression
    ;

augassignment :
    assign_target operator=('+=' | '-=' | '/=' | '*=' | '&=' | '|=' | '^=' | '
↪%=' | '<<=' | '>>=' ) expression
    ;

// Note: VOID can be used in multiple ways but a semantic AST check takes␣
↪care of any mistakes there later.
assign_target:
    scoped_identifier               #IdentifierTarget
    | arrayindexed                  #ArrayindexedTarget
    | directmemory                  #MemoryTarget
    | pointerdereference            #PointerDereferenceTarget
    | VOID                          #VoidTarget
    ;


multi_assign_target:
    assign_target (',' assign_target)+ ;

postincrdecr :  assign_target  operator = ('++' | '--') ;

expression :
    '(' expression ')'
    | sizeof_expression = 'sizeof' '(' sizeof_argument ')'
    | functioncall
    | left = expression EOL? bop = '.' EOL? right = expression          //
↪"scope traversal operator"
    | <assoc=right> prefix = ('+'|'-'|'~') expression
    | left = expression EOL? bop = ('*' | '/' | '%' ) EOL? right = expression
    | left = expression EOL? bop = ('+' | '-' ) EOL? right = expression
    | left = expression EOL? bop = ('<<' | '>>' ) EOL? right = expression
    | left = expression EOL? bop = '&' EOL? right = expression
    | left = expression EOL? bop = '^' EOL? right = expression
    | left = expression EOL? bop = '|' EOL? right = expression
    | left = expression EOL? bop = ('<' | '>' | '<=' | '>=') EOL? right =␣
↪expression
    | left = expression EOL? bop = ('==' | '!=') EOL? right = expression
    | rangefrom = expression rto = ('to'|'downto') rangeto = expression (STEP␣
↪rangestep = expression)?    // can't create separate rule due to mutual␣
↪left-recursion
    | left = expression EOL? bop = 'in' EOL? right = expression
    | left = expression EOL? bop = NOT_IN EOL? right = expression
    | prefix = 'not' expression
    | left = expression EOL? bop = 'and' EOL? right = expression
```

(continues on next page)

(continued from previous page)

```
    | left = expression EOL? bop = 'or' EOL? right = expression
    | left = expression EOL? bop = 'xor' EOL? right = expression
    | literalvalue
    | scoped_identifier
    | arrayindexed
    | directmemory
    | addressof
    | expression typecast
    | if_expression
    | branchcondition_expression
    | pointerdereference
    | staticstructinitializer
    ;

tuple_expression: expression (',' EOL? expression)+  ;

sizeof_argument: basedatatype | scoped_identifier | pointertype | addressof ;


arrayindexed:
    scoped_identifier arrayindex+
    ;


typecast : 'as' datatype;

directmemory : '@' '(' expression ')';

addressof : <assoc=right> (ADDRESS_OF | TYPED_ADDRESS_OF | ADDRESS_OF_LSB |␣
↪ADDRESS_OF_MSB) scoped_identifier arrayindex? ;

functioncall : scoped_identifier '(' EOL? expression_list? EOL? ')'  ;

functioncall_stmt : VOID? scoped_identifier '(' EOL? expression_list? EOL? ')
↪'  ;

expression_list :
    expression (',' EOL? expression)*           // you can split the␣
↪expression list over several lines
    ;

returnstmt : 'return' returnvalues? ;

returnvalues: expression (',' expression)*  ;

breakstmt : 'break';

continuestmt: 'continue';

identifier :  UNICODEDNAME | UNDERSCORENAME | ON | CALL | INLINE | PRIVATE |␣
↪STEP ;              // due to the way antlr creates tokens, need to list␣
↪the tokens here explicitly that we want to allow as identifiers too
```

(continues on next page)

(continued from previous page)

```
scoped_identifier :  identifier ('.' identifier)* ;

integerliteral :  intpart=(DEC_INTEGER | HEX_INTEGER | BIN_INTEGER) ;

booleanliteral :  'true' | 'false' ;

arrayliteral :  EMPTYARRAYSIG | '[' EOL? expression? (',' EOL? expression)* ',
↪'? EOL? ']' ;       // you can split the values over several lines,␣
↪trailing comma allowed

stringliteral : (encoding=UNICODEDNAME ':')? STRING ;

charliteral : (encoding=UNICODEDNAME ':')? SINGLECHAR ;

floatliteral :  FLOAT_NUMBER ;


literalvalue :
    integerliteral
    | booleanliteral
    | arrayliteral
    | stringliteral
    | charliteral
    | floatliteral
    ;

inlineasm :  directivename EOL? INLINEASMBLOCK;         // directive name␣
↪should be '%asm' or '%ir'

subroutine :
    PRIVATE? INLINE? 'sub' identifier '(' sub_params? ')' sub_return_part?␣
↪EOL? (statement_block EOL?)
    ;

sub_return_part : '->' datatype (',' datatype)*  ;

statement_block :
    '{' EOL?
        (statement | EOL) *
    '}'
    ;


sub_params :  sub_param (',' EOL? sub_param)* ;

sub_param: vardecl ('@' register=UNICODEDNAME)? ;

asmsubroutine :
    PRIVATE? INLINE? 'asmsub' asmsub_decl EOL? (statement_block EOL?)
    ;
```

(continues on next page)

(continued from previous page)

```
extsubroutine :
    PRIVATE? 'extsub' (TAG (constbank=integerliteral | varbank=scoped_
↪identifier))? address=expression '=' asmsub_decl
    ;

asmsub_signature : '(' asmsub_params? ')' asmsub_clobbers? asmsub_returns? ;

asmsub_decl : identifier asmsub_signature ;

asmsub_params :  asmsub_param (',' EOL? asmsub_param)* ;

asmsub_param :  vardecl '@' register=UNICODEDNAME ;      // A,X,Y,AX,AY,XY,Pc,
↪Pz,Pn,Pv allowed

asmsub_clobbers : 'clobbers' '(' clobber? ')' ;

clobber :  UNICODEDNAME (',' UNICODEDNAME)* ;       // A,X,Y allowed

asmsub_returns :  '->' asmsub_return (',' EOL? asmsub_return)* ;

asmsub_return :  datatype '@' register=UNICODEDNAME ;     // A,X,Y,AX,AY,XY,
↪Pc,Pz,Pn,Pv allowed


if_stmt :  'if' expression EOL? (statement | statement_block) EOL? else_part?␣
↪ ; // statement is constrained later

else_part :  ELSE EOL? (statement | statement_block) ;   // statement is␣
↪constrained later

if_expression :  'if' expression EOL? THEN? EOL? expression EOL? ELSE EOL?␣
↪expression ;

branchcondition_expression:  branchcondition THEN? expression EOL? ELSE EOL?␣
↪expression ;


// This is a cursed mix of IdentifierReference (scoped identifiers) and␣
↪binary expressions with '.' dereference operators.
// but it is needed for now to not have to rewrite all of Prog8's dependence␣
↪on how the IdentifierReference now works (fully qualified identifier string␣
↪inside)
// in the future this probably has to be reworked completely to split up the␣
↪scoped identifier names and just rely on the '.' operator exclusively,
// but that also requires rewriting al name lookup code.  Ah well, we'll␣
↪cross that bridge when we get there.
pointerdereference:  (prefix = scoped_identifier '.')? derefchain ('.' field␣
↪= identifier)? ;           // TODO  this doesn't look pretty when dealing␣
↪with the difference between a final ^^ at the end or not

derefchain :  singlederef ('.' singlederef)* ;
```

(continues on next page)

(continued from previous page)

```
singlederef : identifier arrayindex? POINTER ;

branch_stmt : branchcondition EOL? (statement | statement_block) EOL? else_
↪part? ;

branchcondition: 'if_cs' | 'if_cc' | 'if_eq' | 'if_z' | 'if_ne' | 'if_nz' |
↪'if_pl' | 'if_pos' | 'if_mi' | 'if_neg' | 'if_vs' | 'if_vc' ;


forloop :  'for' scoped_identifier 'in' expression EOL? (statement |␣
↪statement_block) ;

whileloop:  'while' expression EOL? (statement | statement_block) ;

untilloop:  'do' (statement | statement_block) EOL? 'until' expression ;

repeatloop:  'repeat' expression? EOL? (statement | statement_block) ;

unrollloop:  'unroll' expression EOL? (statement | statement_block) ;      //␣
↪note: expression must evaluate to a constant

whenstmt: 'when' expression EOL? '{' EOL? (when_choice | EOL) * '}' EOL? ;

when_choice:  (expression_list | ELSE ) '->' (statement | statement_block ) ;

// ON...GOTO/ON...CALL with optional else clause - classic BASIC-style multi-
↪way branch.
// This is intentional retro syntax appropriate for 6502/BASIC target␣
↪audience.
// The else_part handles out-of-range indices (useful for menu dispatch with␣
↪error handling).
// Compiles efficiently to 6502 jump tables. NOT a design flaw - fits the␣
↪retro platform.
ongoto: ON expression kind=(GOTO | CALL) directivenamelist EOL? else_part? ;

staticstructinitializer: POINTER? scoped_identifier ':' arrayliteral ;
```

CHAPTER

ELEVEN

PERFORMANCE PROFILING

11.1 Run-time memory profiling with the X16 emulator

The compiler has the `-dumpvars` switch that will print a list of all variables and where they are placed into memory. This can be useful to track which variables end up in zeropage for instance. But it doesn't really show if the choices made are good, i.e. if the variables that are actually the most used in your program, are placed in zeropage.

But there is a way to actually *measure* the behavior of your program as it runs on the X16. See it as a simple way of *profiling* your program to find the hotspots that maybe need optimizing:

The X16 emulator has a `-memorystats` option that enables it to keep track of memory access count statistics, and write the accumulated counts to a file on exit. Prog8 then provides a Python script `profiler.py` (find it in the "scripts"subdirectory of the source code distribution, or online here). This script cross-references the memory stats file with an assembly listing of the program, produced by the Prog8 compiler with the `-asmlist` option. It then prints the top N lines in your (assembly) program source that perform the most reads and writes, which you can use to identify possible hot spots/bottlenecks/variables that should be better placed in zeropage etc. Note that the profiler simply works with the total number of accesses to memory locations. This is *not* the same as the most run-time (cpu instructions cycle times aren't taken into account at all)! Here is an example of the output it generates:

```
$ scripts/profiler.py -n 10 cobramk3-gfx.list memstats.txt
↪                                                          ✓

number of actual lines in the assembly listing: 2134
number of distinct addresses read from  : 22006
number of distinct addresses written to : 8179
total number of reads  : 375106285 (375M)
total number of writes : 63601962 (63M)

top 10 most reads:
$007f (7198687) : $007e 'P8ZP_SCRATCH_W2' (line 13), $007e 'remainder' (line
↪1855)
$007e (6990527) : $007e 'P8ZP_SCRATCH_W2' (line 13), $007e 'remainder' (line
↪1855)
$0265 (5029230) : unknown
$007c (4455140) : $007c 'P8ZP_SCRATCH_W1' (line 12), $007c 'dividend' (line
↪1854), $007c 'result' (line 1856)
$007d (4275195) : $007c 'P8ZP_SCRATCH_W1' (line 12), $007c 'dividend' (line
↪1854), $007c 'result' (line 1856)
$0076 (3374800) : $0076 'label_asm_35_counter' (line 2082)
```

(continues on next page)

(continued from previous page)

```
$15d7 (3374800) : $15d7 '9c 23 9f               stz  cx16.VERA_DATA0' (line␣
↪2022), $15d7 'label_asm_34_repeat' (line 2021)
$15d8 (3374800) : $15d7 '9c 23 9f               stz  cx16.VERA_DATA0' (line␣
↪2022), $15d7 'label_asm_34_repeat' (line 2021)
$15d9 (3374800) : $15da '9c 23 9f               stz  cx16.VERA_DATA0' (line␣
↪2023)
$15da (3374800) : $15da '9c 23 9f               stz  cx16.VERA_DATA0' (line␣
↪2023)

top 10 most writes:
$9f23 (14748104) : $9f23 'VERA_DATA0' (line 1451)
$0265 (5657743) : unknown
$007e (4464393) : $007e 'P8ZP_SCRATCH_W2' (line 13), $007e 'remainder' (line␣
↪1855)
$007f (4464393) : $007e 'P8ZP_SCRATCH_W2' (line 13), $007e 'remainder' (line␣
↪1855)
$007c (4416537) : $007c 'P8ZP_SCRATCH_W1' (line 12), $007c 'dividend' (line␣
↪1854), $007c 'result' (line 1856)
$007d (3820272) : $007c 'P8ZP_SCRATCH_W1' (line 12), $007c 'dividend' (line␣
↪1854), $007c 'result' (line 1856)
$0076 (3375568) : $0076 'label_asm_35_counter' (line 2082)
$01e8 (1310425) : cpu stack
$01e7 (1280140) : cpu stack
$0264 (1258159) : unknown
```

Apparently the most cpu activity while running this program is spent in a division routine which uses the 'remainder'and 'dividend'variables. As you can see, sometimes even actual assembly instructions end up in the tables above if they are in a routine that is executed very often (the 'stz'instructions in this example). The tool isn't powerful enough to see what routine the variables or instructions are part of, but it prints the line number in the assembly listing file so you can investigate that manually.

You can see in the example above that the variables that are among the most used are neatly placed in zeropage already. If you see for instance a variable that is heavily used and that is *not* in zeropage, you could consider adding `@zp` to that variable's declaration to prioritize it to be put into zeropage.

11.2 Subroutine call profiling with the X16 emulator

For the Commander X16, the compiler has a `-profiling` option that instruments your Prog8 program with subroutine call profiling. This allows you to analyze which subroutines take the most time during program execution, helping you identify performance bottlenecks. When you compile with this flag, the instrumented subroutine calls write data to the X16 emulator' s debug output console. You'll have to redirect the emulator's output to a csv file to capture it. The information contains:

- The subroutine name
- A timestamp (emulator cycle count)
- The call depth in the call stack

To use subroutine profiling:

1. Compile your program with the `-profiling` flag:

```
prog8c -target cx16 -profiling your_program.p8
```

2. Run the compiled program in the x16 emulator. When the program exits, the emulator will have generated a `profile.csv` file.
3. The `profile.csv` file contains the profiling data in CSV format with three columns:
   - **Call depth** (hex): The nesting level of the call (lower values = deeper in the call stack, steps down by 2)
   - **Timestamp** (hex): The emulator cycle count when the call was made
   - **Routine name**: The fully qualified name of the subroutine (e.g., `main.start`, `galaxy.init`)

   The file may contain compiler output headers before the actual CSV data starts. The data section begins after a line of dashes.

11.2.1 Using the parse_profile_csv.py script

The `scripts/parse_profile_csv.py` script provides several ways to analyze and visualize the profiling data. It requires the following Python packages:

- `graphviz` (for PDF/SVG call graph generation)
- The `flamegraph.pl` Perl script (for flame graph generation, usually available via your system package manager)

Run the script:

```
python scripts/parse_profile_csv.py
```

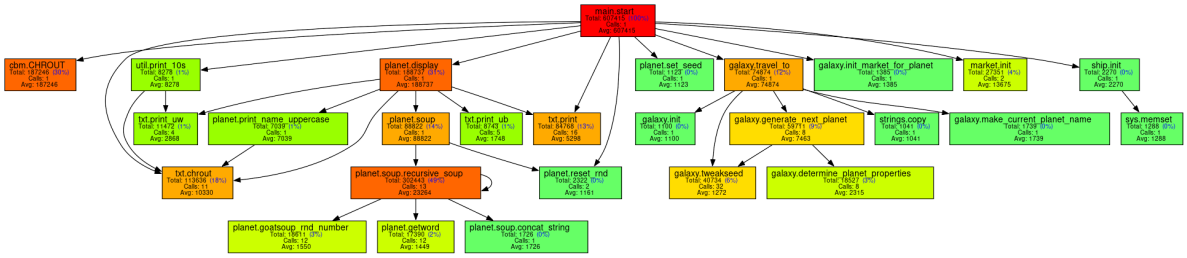
This presents an interactive menu with the following options:

1. Create call graph PDF
Generates a directed graph showing the call relationships between subroutines. Each node displays the routine name, total time, percentage of total runtime, call count, and average time per call. Nodes are color-coded based on their percentage of total runtime:

- **Red** (≥50%): Very hot routines - major bottlenecks
- **Orange-red** (25-49%): Hot routines
- **Orange-yellow** (10-24%): Significant routines
- **Yellow** (5-9%): Moderate routines
- **Green** (<5%): Low impact routines

2. Create call graph SVG
Same as option 1, but outputs an SVG file instead of PDF. The SVG format is interactive and can be viewed in any web browser.

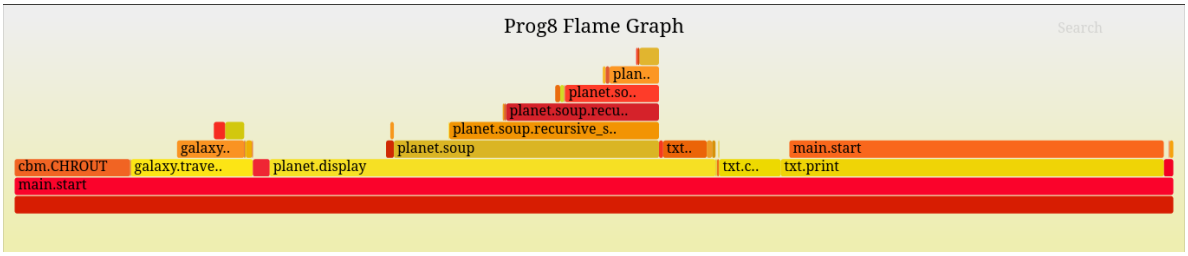
Example call graph:



3. Generate flame graph SVG

Creates an interactive flame graph visualization using the standard `flamegraph.pl` tool. Flame graphs show the call stack horizontally, with the width of each box proportional to the time spent in that routine. This makes it easy to spot performance bottlenecks at a glance. Open the resulting SVG file in a web browser to interact with it (zoom, search, click for details).

Example flame graph:



4. Print routine statistics

Prints three tables to the console:

- **By Time**: Routines sorted by total time (identifies biggest optimization opportunities)
- **By Calls**: Routines sorted by call count (identifies frequently called routines)
- **By Avg Time**: Routines sorted by average time per call (identifies inherently slow routines)

Each table shows the routine name, total time, percentage of total runtime, call count, and average time. An optimization guide explains how to interpret each table.

11.2.2 Example workflow

Here's a typical workflow for profiling a Prog8 program:

```
# Compile with profiling enabled
prog8c -target cx16 -profiling examples/textelite.p8

# Run in emulator, saving output to profile.csv
x16emu -run -prg textelite.prg > profile.csv

# Analyze the results
python scripts/parse_profile_csv.py
# Select option 4 to see statistics, or option 3 for a flame graph
```

The output will show you which routines consume the most time. For example, if a routine shows 49% of total time, optimizing that routine would have the biggest impact on overall performance.

Make sure to eventually get rid of the profiling code and recompile the program normally without the `-profiling` option, because the logic takes up space and does slow down the actual program a bit.

11.2.3 Tips for effective profiling

- Run your program through typical usage scenarios to get representative profiling data
- Look for routines with high total time percentages - these are your optimization priorities
- A routine with many calls but low average time might benefit from small optimizations
- A routine with high average time per call may need algorithmic improvements
- Use the flame graph to quickly identify hot paths in the call stack
- Compare profiles before and after optimizations to measure improvement
- **Recompile the program WITHOUT profiling mode after you're done** –because the profiling logic takes up space and slows down the actual program

11.2.4 Limitations

- Profiling only works on the cx16 target with the x16emu emulator
- Only subroutine call *statements* are instrumented, and *some* function call *expressions* (extsubs)
- The profiler measures cumulative time (time in the routine plus all routines it calls)
- Very short routines may have measurement overhead that skews results
- Recursive routines are supported but may be hard to interpret in the output

CHAPTER

TWELVE

PORTING GUIDE

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

It is not required to change the compiler itself to make it support new compilation targets. A few of the most commonly used ones are built-in (such as c64, cx16), but you can use separate configuration files to create new targets that the compiler can use. See *Customizable targets* for details about this. You still need to provide most of the information asked for in this porting guide and code that into the configuration file.

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.

12.1 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)

12.2 Memory Map

12.2.1 Zeropage

1. *Absolute requirement:* Provide four words (16 bit byte pairs) 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 four 16-bit zero page words are absolutely required to be able to use prog8 on the system. But more known available zeropage locations mean smaller and faster programs.

12.2.2 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?

12.3 Character encodings

1. provide the primary character encoding table that the system uses (i.e. how is text represented in memory. For example, PETSCII)
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.

12.4 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.

12.4.1 Floating point

Prog8 can support floating point math *if* the target system has suitable 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.

12.5 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.

CHAPTER

THIRTEEN

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.

vtsbank: a library for affine transformation and rotation of sprites and tiles
: "VERA Tile Set and Sprite feature"is a module that provides all sorts of routines to perform tile and sprite transformations and rotation, using the VeraFX hardware feature. Includes examples.

C64 REU Banking

A Prog8 library module that provides Commander X16 style RAM banking on a C64 with an REU. This module provides cx16.rambank(), x16jsrfar() and extsub @bank functionality on a C64.

Library blob link example

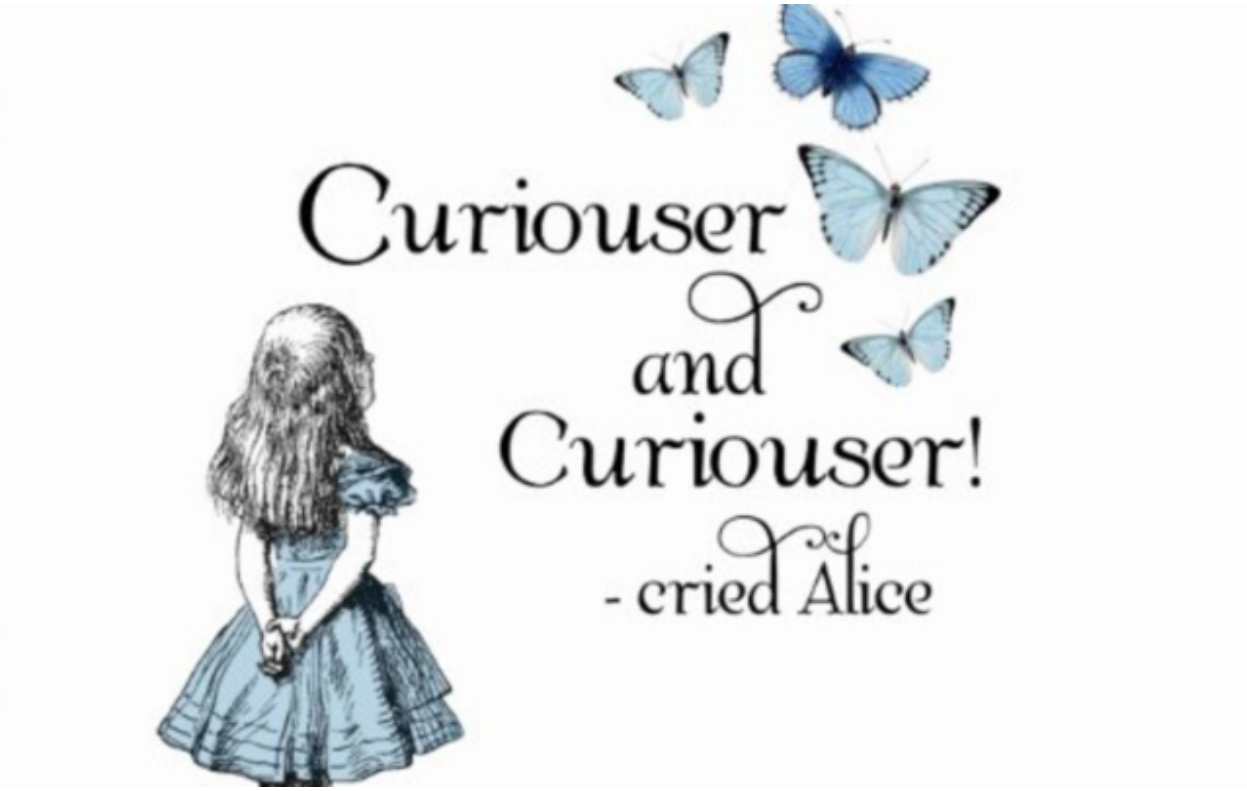
An example of a simple utility that can link symbols in a main Prog8 program so that they are accessible from an externally loaded library blob. It pre-processes the debug symbols list file at compile time, and substitutes references in a template module file.

XLink: an alternative library blob link example

This is another approach to access routines from a banked loaded library, and it does it at run time. In this demo a jump table is not only created in the library, but also in the main program and copied into the library for its use.

Additional custom compilation targets (such as VIC-20)

Various custom targets for Prog8 that are not (yet?) part of the Prog8 examples themselves. These additional compilation targets may be in varying state of completeness. Perhaps most recognisable at the time of adding this link, are the various VIC-20 targets.



CHAPTER

FOURTEEN

TODO

14.1 Future Things and Ideas

- fully remove -nostdlib compiler option? It has become redundant now that the import search path order has been changed?
- make enums strongly typed instead of just syntactic sugar for ints (see ideas/enum-strong-type.md for the plan)
- add %option private_symbols to make access mode private by default; need (new) 'public' keyword to explicitly mark symbols public.
- symboldump: some sort of javadocs generated from the p8 source files (instead of just the function signatures). Use markdown for formatting, not html.
- when implementing unsigned longs: remove the (multiple?) "TODO "hack"to allow unsigned long constants to be used as values for signed longs, without needing a cast
- struct/ptr: implement the remaining TODOs in PointerAssignmentsGen.
- struct/ptr: support pointer to pointer?
- struct/ptr: support for typed function pointers, so that we can call them via functptr^^(args) maybe even without the ^^ operator? (&routine could be typed by default as well then)
- struct/ptr: really fixing the pointer dereferencing issues (cursed hybrid between IdentifierReference, PtrDereferece and PtrIndexedDereference) may require getting rid of scoped identifiers altogether and treat '.'as a "scope or pointer following operator"
- struct/ptr: (later, nasty parser problem:) support chaining pointer dereference on function calls that return a pointer. (type checking now fails on stuff like func().field and func().next.field)
- Make all constants long by default? or not? (remove type name altogether), reduce to target type implicitly if the actual value fits. -> long-consts branch This will break some existing programs that depend on value wraparound, but gives more intuitive constant number handling. Can give descriptive error message for old syntax that still includes the type name?
- add documentation for more library modules instead of just linking to the source code
- sizeof(pointer) is now always 2 (an uword), make this a variable in the ICompilationTarget so that it could be 4 at the time we might ad a 32-bits 68000 target for example. Much code assumes word size addresses though.

- add float support to the configurable compiler targets. Restrictions: just have "cbm-style floats"as an option (to that it can slot into the current float codegen), where all you have to specify is the addresses of AYINT and GIVAYF and FADDT and all their friends.
- Change scoping rules for qualified symbols so that they don't always start from the root but behave like other programming languages (look in local scope first), maybe only when qualified symbol starts with '.'such as: .local.value = 33
- implement the signed remainder byte and word routines on 6502 (virtual target already has them working)
- implement the signed divmod byte and word routines on 6502 (virtual target already has them working)
- 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)
- add new directives `%bssaddress` and `%slabsaddress` to set the memory address for the BSS area and memory slabs (analogous to `%address` for program load address). Note: these should be mutually exclusive with the existing CLI options (`-varsgolden`, `-varshigh`, `-slabsgolden`, `-slabshigh`) because the CLI options are target-aware shorthands (set bank symbols, do bounds checking against predefined ranges) while the directives are raw addresses —they'd conflict if both specified for the same area.
- the c64 sprite multiplexer example may need timing adjustments after compiler changes (not a compiler bug —cycle-exact C64 code is inherently fragile)

14.2 Romable (%option romable)

- ForLoopsAsmGen: fix remaining codegens. Three methods use self-modifying code (patching `cmp #0` immediates) with no romable-safe alternative: - `forOverBytesRangeStepGreaterOne` (byte, abs(step)>=2) - `forOverWordsRangeStepGreaterOne` (word, step>=2) - `forOverWordsRangeStepGreaterOneDescending` (word, step<=-2) Fix pattern (already used by step-1 methods): add `if(romable)` branch that allocates a temp var via `createTempVarReused`, stores the loop end value into it, and compares against it. Existing self-modifying code stays in `else` branch for RAM programs.
- BuiltinFunctionsAsmGen: `callfar` / `callfar2` with non-const bank/addr. Uses self-modifying `sta +0` / `sty +1` to patch JSRFAR operands. Needs a RAM trampoline approach (copy stub with variable args into RAM, JSR to that).
- FunctionCallAsmGen: `extsub` with variable bank. Same JSRFAR operand patching issue. Needs RAM trampoline.
- Add more test coverage for the romable option.

14.3 IR/VM

- maybe change all branch instructions to have 2 exits (label if branch condition true, and label if false) instead of 1, and get rid of the implicit "next code chunk"link between chunks.
- implement more TODOs in AssignmentGen?
- add more optimizations in IRPeepholeOptimizer?

- **Multi-Level IR Design**: Consider introducing a High-Level IR (HLIR) layer before the current low-level IR to preserve semantics like loop bounds, array indexing, and structure field access. The current IR is effectively "assembly with infinite registers."Recommendation when adding non-6502 targets: Implement a custom HLIR using Kotlin sealed classes (inspired by MLIR dialects but lighter weight). Flow: SimpleAst -> HLIR (Loops/Arrays) -> Lowering -> Current IR (Ops/Regs) -> Codegen. Don't adopt LLVM (too low-level) or QBE (too simple). Custom HLIR fits Kotlin best and preserves semantic intent. **Important**: HLIR's value for 6502 is minimal if the backend consumes only the lowered IR. For 6502 to benefit from HLIR, the backend would need to target HLIR directly (bypassing the lowering pass for applicable constructs), adding complexity. HLIR is primarily useful for non-6502 backends (68000) and the VM interpreter. **Split word arrays** are a prime example: currently represented as two separate `_lsb/_msb` ubyte arrays in the IR, so a single `words[i] += 50` expands to 8 byte-level IR instructions (two LOADM, CONCAT, ADD, LSIGB, MSIGB, two STOREM). At the HLIR level this should remain a single word-array augmented assignment; the lowering pass can split it into `_lsb/_msb` ops (for 6502) or keep it as a word op (for 68000).

Missing VM Implementations (VirtualMachine.kt) - `IRInlineBinaryChunk` and `IRInlineAsmChunk` - inline chunks cannot be loaded by the VM (VmProgramLoader.kt). Limitation of the current VM design: program is not loaded into memory as data - VM label address loading - `VmProgramLoader.kt` throws when it cannot resolve a label address as a value (`"vm cannot yet load a label address as a value"`).

14.4 Language Server

- Fix the 2 remaining disabled (xtest) LSP tests in Prog8LanguageServerTest.kt: - find references for subroutines (returns 0 results) - completions: symbol completions (not scope-aware, no user symbols)

14.5 Libraries

- Add split-word array sorting routines to sorting module?
- make a list of all floats.* routines that the compiler expects for full float support?

14.6 Optimizations

- Inline subroutines with 1 (or even more) simple argument, BE VERY CAREFUL though because this is a complicated inlining action that could cause endless loops (Baccarat game suffered from this). Currently inlining is limited to subroutines with 0 params.
- Port more benchmarks from https://thred.github.io/c-bench-64/ to prog8 and see how it stacks up. (see benchmark-c/ directory)
- Compilation speed: try to join multiple modifications in 1 result in the AST processors instead of returning it straight away every time
- various optimizers skip stuff if compTarget.name==VMTarget.NAME. Once 6502-codegen is done from IR code, those 6502 only optimizations should probably be removed

14.7 Dead Code Elimination bug in 64tass, for nested subroutines

- When a subroutine contains a nested `asmsub` (or possibly a nested `sub()`), 64tass cannot properly eliminate the outer subroutine if ANY symbol from within it is referenced elsewhere (even if the outer subroutine itself is never called).
- Workaround: move nested subroutines to be top-level (block-level) subroutines instead.
- Example: in gfx_lores.p8, the nested `plot()` inside `line()` caused unused `line()` to be included in programs that only used other gfx_lores functions (like `circle()`). Fixed by moving it to a separate `internal_line_plot()`.

CHAPTER

FIFTEEN

PROG8 COMPILER RELEASE HISTORY

A condensed timeline of major releases and significant changes in the Prog8 programming language compiler.

Note

Python to Kotlin Transition: The original Prog8 compiler (pre-1.0) was written in **Python**. In early 2022, the compiler was completely rewritten in **Kotlin** starting with v1.0 Beta. The Kotlin rewrite brought significant improvements in performance, IDE support, type safety, and maintainability. This document covers the Kotlin-era releases.

15.1 Summary of Major Language Milestones

Version	Milestone Feature
1.6	Address-of operator &
1.10	`when` statement (multi-way branch)
1.11	Structs (first introduction)
3.0	CPU registers removed, new loop syntax
4.0	CommanderX16 target support
5.0	Calling convention overhaul (registers for returns)
6.0	Virtual registers `cx16.r0-r15`
7.0	Structs removed
8.3	`bool` datatype
9.0	`min`/`max`/`clamp` builtins, `@split` arrays
9.3	Software evaluation stack removed
9.7	Unicode identifiers, `continue` statement
10.0	Short-circuit boolean logic
10.3	`void` keyword for multi-return
11.0	`const long`, split arrays by default
11.1	Multi-value returns from subroutines
11.4	`on .. goto` jump tables
12.0	**``long`` datatype**, **structs reintroduced**, **typed pointers**
12.1	`swap()` builtin, `psg2` module, PET32 floats/gfx/snd, `strings.split()/next_token()`, C128 2MHz mode
12.2	Long loop support, `private`, enum syntax, 2D arrays, new modules (serial, wavfile, adpcm, lineclip)
12.3	Deterministic module search order, -srcdirs priority

15.2 Breaking Changes Summary

Major breaking changes that require code modifications when upgrading:

- **v3.0**: CPU registers (A,X,Y) removed, loop syntax changed
- **v4.0**: Module renames, `mkword()` parameter order flipped
- **v5.0**: Calling convention overhaul
- **v5.3**: Explicit & required for pointer assignments
- **v7.0**: Structs removed, `%target` directive removed
- **v9.0**: `-target` now required
- **v9.3**: Eval stack removed (affects low-level code)
- **v10.0**: `push`/`pop` moved to `sys` module
- **v10.2**: Stricter boolean types
- **v10.4**: Namespace reorganization (`cx16` to `cbm`)
- **v12.0**: Structs reintroduced (different from v1.11), typed pointers
- **v12.1**: Combined virtual register renames (`R0R1_32` → `R0R1`, etc.)
- **v12.2**: `swap()` is now a statement (not a function), `math.crc16()` and `math.crc16_start()` require new parameters, `private` is now a reserved keyword
- **v12.3**: Deterministic module search order, filesystem priority over internal resources

15.3 2019–2022 —Early Development (Python to Kotlin Transition)

v1.0–v1.2 Beta —January–February 2022

- **First public Kotlin release** (v1.0, 26 Jan)
- **MAJOR: Compiler rewritten from Python to Kotlin** —complete implementation language change
- Zeropage variable allocation and block-level variable initialization
- Gradle build system, math optimizations, `floatsafe` zeropage mode

v1.3–v1.6 —March–April 2022

- `asmsub` routines can return values in expressions
- **Address-of operator ``&`` implemented** —replaces `memory` keyword
- For loops can iterate over literal collections
- New builtins: `strlen()`, `sqrt16()`, `pow()`, `powf()`

v1.7–v1.11 —July 2022

- **Array size optional** with initializer
- **``%asmbinary`` directive** for assembly inclusion
- **AST-based Virtual Machine** with optimization passes
- **``when`` statement** —new multi-way control flow

- **Structs added** —composite datatype feature

v1.20 —July 2023

- **Struct literals** —inline struct initialization

15.4 2023 —Language Maturation

v2.0–v2.3 —May–July 2023

- Major compiler speedup (scope lookups, code generation)
- Stricter type checking, optimized `swap()`
- **String value reassignment**, new string functions (`leftstr`, `rightstr`, `substr`)
- Subroutine inlining optimization

v3.0–v3.2 —July–August 2023

- **CPU register variables (A, X, Y) removed** —use inline assembly
- **Loop syntax changes**: `repeat-until` to `do-until`, new `repeat X {}`, `forever {}` removed
- **``continue`` statement removed**
- **``sizeof()`` function added**
- Optimized in-place/augmented assignments —major performance boost

v4.0–v4.1 —August–September 2023

- **CommanderX16 target added** —major platform expansion
- **Floating point support for CX16**, VERA registers, 65c02 features
- **``lsl()``/``lsr()`` removed** —use <<=1/>>=1
- Module renames (`c64scr` to `txt`), `mkword()` parameter order flipped

v4.2–v4.6 —September–October 2023

- **Cross-platform C64/CX16 compatibility**
- **``.w`` postfix removed** —breaking syntax change
- **``%option no_sysinit`` directive** —skip system initialization
- **String escape characters**: `\\xHH`, `\\'`
- **``diskio`` module introduced** —file I/O routines
- **Automatic string comparisons** by value

v5.0–v5.4 —November 2023–January 2024

- **Calling convention overhaul**: args via variables, returns via registers
- **Explicit ``&`` required** for string/array pointer assignments
- **``in`` containment operator**: `if xx in [1,2,3]`
- **Experimental C128 target**
- **Pipe operator ``|>``** and **string encoding syntax** (`iso:"hello"`)
- **``@requirezp`` flag** —force zeropage allocation

- **Rewritten variable allocation** —large program size savings

15.5 2024 —Advanced Features

v6.0–v6.4 —January–March 2024

- **Virtual registers ``cx16.r0..r15``** for CX16 and C64
- **New builtins**: `target()`, `offsetof()`, `memory()`, `cmp()`
- **``gfx2`` module** —enhanced graphics with highres modes
- **``peekw()``, ``pokew()``** —word memory access
- **IRQ handling routines** for CX16
- **Improved RNG** with seeded variants (`rndseed`, `rndseedf`)
- **Assembly codegen completed** —all expression types supported

v7.0–v7.8 —June 2024–February 2025

- **Struct feature removed** —rewrite as separate variables
- **``%target`` directive removed** —use CLI options only
- **Software evaluation stack removed** —frees memory page and X register
- **PET32 target** —Commodore PET 4032 support
- **2x faster multiplication** and square root operations
- **New modules**: `verafx`, `emudbg`, `monogfx`, `sprites`
- **Builtins**: `setlsb()`, `setmsb()`, `math.diff()`, `math.diffw()`

v8.0–v8.13 —April 2024–May 2025

- **R39 CX16 ROM support**
- **``**`` operator removed** —use `floats.pow()`
- **Experimental VM target** with `syscall` builtin
- **API reorganization**: trig/float functions moved to `math`/`floats` modules
- **``bool`` datatype introduced** —optimized true/false (0/1)
- **BSS section** —uninitialized variables, reduced PRG size
- **``divmod()``, ``divmodw()`` builtins**
- **Major codegen optimizations** —significantly smaller and faster code

v9.0–v9.7 —June–December 2024

- **``-target`` now required** (c64 no longer default)
- **New builtins**: `min()`, `max()`, `clamp()`
- **``@split`` storage class** —efficient LSB/MSB array storage
- **``cbm`` module** —all CBM kernal routines
- **Boolean ``when`` conditions**
- **Underscores in numbers**: `320_000`
- **Multiple declarations/assignments**: `ubyte x,y,z` and `x=y=z=calculate()`

- **``continue`` statement** for loops
- **Unicode identifiers**: `knäckebröd`, `π`
- **Negative array indexing** (Python-style)
- **Range containment**: `if x in 10 to 100`

15.6 2024–2026 —Modern Prog8

v10.0–v10.5 —January–November 2024

- **Short-circuit boolean evaluation** (McCarthy logic)
- **Stricter boolean types** —no longer equivalent to bytes, require explicit casting
- **``void`` keyword** —skip unused return values in multi-return assignments
- **Namespace reorganization**: non-X16 variables moved from `cx16` to `cbm`
- **Builtins removed**: `sort`, `reverse`, `any`, `all` to `anyall` module
- **Compiler renamed to ``prog8c``**
- **``defer`` statement** —delayed execution for resource cleanup
- **If-expression**: `result = if x>10 "yes" else "no"`
- **Memory alignment**: `@alignword`, `@alignpage`, `@align64`
- **Array literals with repetition**: `[42] * 99`

v11.0–v11.4 —December 2024–June 2025

- **``const long`` numbers** —32-bit integer literals
- **``goto`` can jump to calculated addresses**
- **Word arrays split by default** (LSB/MSB in separate arrays)
- **Subroutines can return multiple values**
- **Multi-value variable initialization**: `ubyte a,b,c = multi()`
- **``%output library`` and ``%jmptable`` directives** —loadable libraries
- **``%option romable``** —ROM-compatible code warnings
- **ROMABLE programs** —no inline variables or self-modifying code
- **Range choices in ``when`` statements**
- **Foenix256 target**
- **Boolean virtual registers**: `cx16.r0bL`, `cx16.r0bH`
- **``on .. goto`` / ``on .. call``** —efficient jump tables
- **Float to int casts** (truncating)
- **Double buffering** in monogfx module

v12.0–v12.1 —November 2024–February 2026

- **``long`` datatype** —native 32-bit signed integers
- **Structs reintroduced** —grouping multiple fields together
- **Typed pointers** —can point to structs and specific types (not just `uword` addresses)

- **PET32 modules**: `petgfx`, `petsnd`, `diskio`
- **New builtin**: `swap()`
- **C128 enhancements**: `fast()`/`slow()` for 2MHz mode
- **String operations**: case-insensitive comparison, `strings.split()`, `strings.next_token()`
- **Virtual target improvements**: `f_seek()`, `f_tell()`, `diskname()`, `loadlib()`

v12.2 —May 2026

- **Long loop support** —for-loops with long counters on 6502, long math (division, sqrt, min/max/clamp/abs)
- **``private`` keyword** —for structs, enums, aliases, variables, and subroutines
- **Const pointers** —compile-time constant-folding for const pointers
- **``enum`` syntax** —concise constant list declarations
- **2D array support** —`ubyte[3][4] matrix`
- **``swap()`` is now a statement** —more efficient than the previous builtin function
- **``math.crc16()`` signature changed** —now requires `initvalue` and `xorout` parameters for flexibility
- **New compiler options**: `-daemon` (IDE integration), `-nostdlib`
- **New modules**: `serial` (CX16 UART+ZiModem), `lineclip`, `wavfile`, `adpcm`
- **``-libsearch`` fuzzy search fallback** —finds libraries even with partial names

v12.3 —June 2026

- **Deterministic module search order** —replaced alphabetical search with prioritized list.
- **``-srcdirs`` priority** —user-specified source directories now have the highest priority.
- **Neighboring directory priority** —the directory of the importing file is searched before the current directory.
- **Standard library overrides** —the filesystem is now searched before internal resources.
- **Improved error messages** —missing module errors now list all searched filesystem paths and the requester.
- **Fuzzy library search** —`-libsearch` now automatically attempts a fuzzy search if no exact matches are found.
- **Trace imports** —new `-traceimports` option to see exactly how modules are being resolved.
- **Search Path Comparison**:

Step	Old Behavior (Approximate)	New Behavior (Strict)
1	Internal Standard Library	**User Source Directories (-srcdirs)**
2	Target Library Directories	**Neighboring Directory**
3	Neighboring Directory	**Current Working Directory (.)**
4	User Source Directories (alphabetical)	**Target Library Directories**
5	(not applicable)	**Internal Standard Library**

This document summarizes major and minor releases. Bugfix releases (e.g., v12.0.1, v12.1.1) are omitted for brevity.