

```
# File: Emuchron/script/line2.txt
# This script is used for testing glcdLine()

# Erase LCD display
le

# Set horizontal and vertical display size
vs hor=127
vs ver=63

# Paint in total 9x4 edge-to-edge lines
rf factor=0.1 factor<=0.9 factor=factor+0.1
# From left to top and left to bottom
pl f 0 ver*factor hor-(hor*factor) 0
pl f 0 ver*factor hor*factor ver
# From right to top and right to bottom
pl f hor ver-(ver*factor) hor-(hor*factor) 0
pl f hor ver-(ver*factor) hor*factor ver
rn

# Paint the glcdline function name in a rectangle box
pr f 48 27 31 9
pa f 50 29 5x5p h 1 1 glcdline
```

- EMUCHRON - A Monochron emulator for Debian Linux



Author:	Toine Ceulemans
Version:	v2.0
Date:	15 December 2015

This page is intentionally left blank

Disclaimer

The Emuchron project and its contents is provided as-is and is distributed under the GNU Public License which can be found at <http://www.gnu.org/licenses/gpl.txt>.

QuintusVisuals® is a registered trademark of Quintus consultants b.v.
TIBCO Spotfire® is a registered trademark of TIBCO.

Intended audience

This document is intended for:

- Monochron clock programmers

Prerequisites

The reader of this document is familiar with Linux in general and Debian Linux in particular.

Acknowledgements

- CaitSith2 and ladyada
The Emuchron project started with the original Monochron pong clock firmware.
<https://github.com/adafruit/monochron>
- Balza3
The Mario alarm in Emuchron is based on notes, beats and play logic provided in an Arduino project.
<http://www.youtube.com/watch?v=VqeYvJpibLY>
- Tz / HarleyHacking
The core functionality to encode a QR uses code from project qduino.
<https://github.com/tz1/qduino>

Version history

Version (date) Author	Description
v2.0 (2015-12-15) T. Ceulemans	<p>Emuchron emulator code base:</p> <ul style="list-style-type: none"> – Support for Debian 8 (Jessie) 64-bit. – Restructured mchron interpreter code. – Interpreter is now fully based on a command dictionary and the use of type <code>double</code>. – A variable name is now unlimited in length and consists of any combination of uppercase/lowercase characters a..z. – Extended functionality and use of expression evaluator. – Added if-then-else logic commands. – Added commands to show the command dictionary, the result of an expression and paint a formatted number on the stubbed LCD display. – Command 'rw' (repeat-while) is replaced by 'rf' (repeat-for) with an improved syntax structure. – The wait command 'w' now has a wait granularity of 0.001 sec instead of 0.01 sec. <p>Monochron firmware code base:</p> <ul style="list-style-type: none"> – Alarming/snoozing will stop upon pressing the 'M' button. – Improved single-press detection of '+' button. – Press-hold increment steps are doubled after 10 regular increments for min/sec/year elements. – Improved pong clock game play. – Improved object code execution efficiency over entire code base. – Optimizations in glcd module resulting in improved draw performance in almost all high-level glcd functions. – Code optimizations aimed at reducing firmware size. <p>Relevant bug fixes:</p> <ul style="list-style-type: none"> – An mchron crash occurs when printing mchron statistics ('sp') under certain circumstances. – In <code>glcdPutStr3v()</code> for orientation <code>ORI_VERTICAL_TD</code>, text for horizontal location <code>x</code> is incorrectly painted at location <code>x-1</code>. <p>Generic:</p> <ul style="list-style-type: none"> – Minor bug fixing in both code bases and documentation.

Summary

[Emuchron](#) is a lightweight Monochron emulator for Debian Linux 6, 7 and 8. It features a test and debugging platform for Monochron clocks and high level glcd graphics functions, and a software framework for clock plugins.

Included in the software are enhancements to the high level glcd graphics library, modified clock configuration pages, several example clocks, a graphics performance test module, a build option to switch between a two-tone and Mario melody alarm, and demo and test scripts.

Preface

Even before I bought [Adafruit's Monochron clock](#) in mid-2012 I thought about the clocks I wanted to code.

While waiting for the clock to be delivered at my doorstep and for a friend with the right tools to put it together, by using the pong firmware as a base I started coding some basic clocks. However, without an actual Monochron clock to upload the firmware to it is rather difficult to verify the correctness of the code. Being too impatient I wrote a very simple tool in a Debian Linux environment that was able to dump the (perceived) results of a glcd graphics function in a plain text file, thus allowing me to analyze the output of functional clock code. Over time that tool was enhanced and parts were rewritten several times, up to the moment that I got myself a basic Monochron emulator fitting my needs very well. This emulator then served as a base to develop, debug and optimize both new and existing code.

Since then parts of Emuchron were, again, rewritten while enhancing its features and making it more robust. In late 2013 documentation was written in preparation for a first publication on github.

Document conventions


Throughout the document examples are provided of Emuchron command line interface sessions.

Relevant end-user input is printed in black/bold. See example below.

```
mchron> # A command prompt is no end-user input and comment lines are usually not
mchron> # relevant end-user input. They are therefor not in bold. Actual mchron
mchron> # commands are relevant and as such are printed in bold.
mchron> # See the bold 'pl' (paint line) mchron command example below.
mchron> pl f 100 10 126 62
mchron>
```

Relevant end-user actions and tool feedback is printed in red/bold. See example below.

```
mchron> # Press '<ctrl>d' on an empty line to exit mchron
mchron>
<ctrl>d - exit
$
```



This page is intentionally left blank

Table of contents

1	Introduction	1
1.1	About this manual	1
1.2	Problem description and solution	1
1.3	Emuchron features and limitations	1
1.3.1	The Emuchron emulator	1
1.3.2	The Emuchron clock plugin framework	2
1.3.3	The Emuchron command line tool mchron	3
1.4	Debian Linux and AVR	3
1.5	Migrating from Emuchron v1.x to v2.x	4
2	The Emuchron project	6
2.1	The project folder structure	6
2.2	Monochron firmware high-level runtime environment	6
2.3	Emuchron emulator high-level runtime environment	8
2.4	Monochron main loop, buttons and clocks	10
2.5	Monochron variables for clock plugins	12
2.6	The glcd graphics library enhancements	13
2.6.1	Overview of high-level glcd functions	14
2.6.2	The <code>lcdLine[]</code> buffer	14
2.6.3	Text fonts and font scaling	15
2.6.4	Text orientation	15
2.6.5	Fill patterns	16
2.6.6	Fill alignment	17
2.6.7	Circle draw patterns	17
2.7	Monochron configuration screens	18
2.8	Monochron two-tone and Mario alarm melodies	19
2.9	Performance tests for high-level glcd functions	20
2.10	Demo and test mchron command scripts	20
2.11	The pre-built monochron.hex firmware	21
2.12	Quick guide into the <code>clockDriver_t</code> structure	21
2.13	Quick guide into adding a new clock plugin	22
3	Setting up the software environment	23
3.1	Introduction	23
3.2	Configuring Debian	23
3.2.1	General Debian requirements	23
3.2.2	Configuring a Debian VM in VirtualBox	23
3.2.3	Configuring a Debian VM in VMware Fusion	24
3.3	Unpacking the project software	24
3.4	Installing required Linux packages	24
3.5	Copying configuration file for minicom	25
3.6	Setting up and using an ncurses Monochron terminal	25
3.6.1	Creating a Monochron terminal profile	25
3.6.2	Starting a Monochron ncurses terminal	25
3.6.3	Changing the size of a Monochron ncurses terminal	26
3.7	Debian 8 issues and regression in functionality	27
3.7.1	ALSA audio is getting ever less responsive	27
3.7.2	There may not be audio at all	27
3.7.3	A terminal profile can no longer set a terminal header	28
3.7.4	The gdb debugger cannot find file "syscall-template.S"	28
4	Building firmware and the emulator	30

4.1	Building Monochron firmware	30
4.2	Building Emuchron and mchron command line tool	31
4.3	Uploading Monochron firmware to Monochron clock	31
5	The mchron command line tool	34
5.1	Introduction.....	34
5.2	Starting mchron	34
5.3	Interrupting and stopping mchron	35
5.4	Pre-emptive coredump of mchron.....	36
5.5	The mchron stack trace.....	37
5.6	Recovering from command syntax and parse errors.....	37
5.7	The mchron command line history log	37
5.8	The mchron command groups.....	38
5.8.1	'#' – Comments	39
5.8.2	'a' – Alarm	40
5.8.3	'b' – Beep	41
5.8.4	'c' – Clock	42
5.8.5	'd' – Date.....	43
5.8.6	'e' – Execute	44
5.8.7	'h' – Help	45
5.8.8	'i' – If	46
5.8.9	'l' – LCD.....	47
5.8.10	'm' – Monochron	48
5.8.11	'p' – Paint	49
5.8.12	'r' – Repeat	51
5.8.13	's' – Statistics	52
5.8.14	't' – Time	56
5.8.15	'v' – Variable	57
5.8.16	'w' – Wait.....	58
5.8.17	'x' – Exit	59
5.9	Processing an mchron 'hello world!' command.....	60
5.10	Building and executing an mchron command list	62
6	Debugging clock and graphics code	65
6.1	Debugging using the FTDI debug strings method.....	65
6.1.1	Requirements and limitations	65
6.1.2	Monochron debug strings via FTDI port on Debian Linux	65
6.2	Debugging using Emuchron stubbed FTDI debug strings.....	67
6.3	Debugging Emuchron using gdb	68
6.3.1	Requirements for Debian 8 when using gdb.....	69
6.3.2	Limitations on using ncurses.....	69
6.3.3	Debugging Emuchron with ncurses device using Nemiver	69
6.3.4	Debugging Emuchron with ncurses device using DDD	72
6.3.5	Debugging an mchron coredump file	74
7	Frequently asked questions	75
7.1	Differences between Monochron and Emuchron	75
7.2	Linux mathlib accuracy vs. AVR mathlib accuracy	75
7.3	Accuracy and reliability of the expression evaluator	76
7.4	Monochron real time clock (RTC) scanning.....	76
7.5	The ncurses output appears somewhere else	77
7.6	VirtualBox: mchron OpenGL warnings/failure/coredump	77
7.7	No command history when using ncurses LCD device	78
7.8	Performance of the mchron interpreter.....	79

7.9	After an mchron coredump there is no coredump file	79
7.10	There is a delay in starting a stubbed Mario alarm	79
7.11	Firmware size penalty for new Emuchron functionality.....	79
7.12	Is it required to build firmware on Debian Linux	80
7.13	My debugger cannot find file "syscall-template.S"	80
8	Known bugs	81
8.1	The mchron terminal no longer echoes characters	81
8.2	Pending characters in the mchron terminal input buffer	81
A	Screendumps of example clocks	82
A.1	Analog clocks	83
A.2	Big Digit clocks.....	84
A.3	Digital clocks	84
A.4	Mosquito clock	85
A.5	Nerd clock	85
A.6	Pong clock	86
A.7	Puzzle clock	86
A.8	QR clocks	87
A.9	Slider clock	87
A.10	QuintusVisuals clocks	88
B	High-level glcd performance tests	89
B.1	Test results Emuchron v1.3 vs v1.2	89
B.2	Test results Emuchron v2.0 vs v1.3	93
C	Setting up a Monochron terminal profile.....	98
C.1	Setting up a terminal profile in Debian 6 and 7.....	98
C.2	Setting up a terminal profile in Debian 8	102

This page is intentionally left blank

1 Introduction

1.1 About this manual

The purpose of this manual is to provide background information on Monochron and the Emuchron emulator.

With respect to Monochron and Emuchron, this document in combination with actual code and test and demo scripts should provide enough information to get started.

1.2 Problem description and solution

Coding clocks for the Monochron open source clock is (debatable) fun, but has its drawbacks. The main drawback is not being able to properly test clock and graphics code on a functional level. Clocks sometimes seem to hang, the graphics turn out not to be fluid or are simply incorrect.

Up to now the only way to debug a functional clock and graphics functionality is to generate debug output strings in the Monochron clock and send them via the FTDI bus to a terminal application on the connected computer. Although this debug method is useful, it is considered cumbersome and inflexible.

Enter Emuchron, a lightweight Monochron emulator for Debian Linux.

The main feature of Emuchron is to emulate the Monochron hardware and keep the emulator stubs as far away as possible from functional clock code and high-level graphics functions. This allows a programmer to code, debug and test clocks and graphics functions in a controlled Debian Linux environment ahead of uploading firmware to Monochron. Emuchron is controlled via a command line tool dedicated to supporting these development and test features.

Next, effort is put into creating a Monochron clock plugin environment with the aim to reduce efforts for developing new clocks and building Monochron firmware. This is demonstrated by the list of clocks built from scratch and a migrated pong clock, all included in the firmware node.

And finally, to enhance the graphic capabilities of Monochron clocks, the high-level glcd graphics library now includes a 5x5 proportional font and new text, area fill and support routines.

1.3 Emuchron features and limitations

1.3.1 The Emuchron emulator

The main reason for creating Emuchron is to acquire a means to develop, test and debug clock and graphics functions ahead of uploading it to the Monochron clock. This is achieved by emulating the underlying Monochron hardware using data and function stubs.

These stubs do not implement hardware specific elements such as timing on ports and hardware interrupts. In other words, Emuchron is not meant to be used to develop and debug low level firmware functionality that interacts with hardware.

Instead, Emuchron relies on the fact that this low level firmware functionality is stable. By providing a hardware emulation layer for the low level firmware, Emuchron is then able to provide an environment upon which high level functionality, being software clocks and high-level graphics functions, can be built.

So, Emuchron depends on the stability of the low level firmware functionality. This requirement is fulfilled by taking the original Monochron pong clock firmware, that has been stable for a long time, and use that as a strong foundation. In Emuchron, the core of this firmware has been left unchanged, but most of the other routines have been modified, replaced or enhanced to fit Emuchron requirements.

An example of the Emuchron emulator approach is a function that writes a data byte, containing 8 bit pixels, to the LCD display. The actual firmware does this by setting up a data connection to the LCD display with built-in delays to compensate for hardware response times. In our emulator case, Emuchron has a stub that replaces this firmware functionality with a function that stores the data byte in a data structure representing the LCD display memory. When stored, the data is then passed on to an LCD emulator device. Eventually, the data byte will show up as individual pixels in the window driven by the LCD emulator device.

Like the stub for the LCD display there are others that emulate all other hardware elements, being the real time clock, the clock buttons, the alarm on/off switch and the piezo speaker. Some of these stubs re-use Monochron code while others require fully dedicated stub code.

1.3.2 The Emuchron clock plugin framework

From a software development point of view, Emuchron requires that functional clock code should never access the hardware directly but instead use a (stubbed) interface to low level functionality. This is seen as a software architecture requirement.

This is fulfilled by creating a software layer in which a software clock is regarded as a plugin that only needs to implement functional clock code. Of course, the clock code will access graphics functions that eventually write to the LCD, but the hardware aspect of this access will be hidden from clock plugin level. Even better, some aspects do not need to be dealt with in a clock plugin at all. Sounding the alarm, snoozing, and scanning the buttons and the alarm on/off switch are handled outside the scope of a clock plugin, thus greatly simplifying the efforts needed to create new clocks.

The software framework is implemented by creating a list of global variables that represent the hardware state of the clock that is accessible at clock plugin level. It is the task of the software layers underneath the clock plugins to make these global variables truly represent the hardware state. And in addition to that, have it guaranteed that these variables are stable during the execution of functional clock code.

Clock plugins need to expose only two public functions with a defined interface for clock initialization and clock update. An optional third public function can be defined for clock button handling.

An example of the representation of the clock state in data is a variable that indicates that the time has changed. In addition to this variable there are others that hold the previous timestamp and the new timestamp. This allows a clock plugin to find out what needs to be changed in the layout of a clock, to be achieved by calling the appropriate graphics functions. The main point here is that a clock plugin never needs to interact with the real time clock itself.

1.3.3 The Emuchron command line tool mchron

Emulating hardware and providing software layers to simplify the creation of new clocks and graphics functions is however incomplete as the end user of the emulator must be given proper testing tools as well.

For this, Emuchron provides a command line tool named mchron that allows accessing clock plugins at will, feed clocks with a continuous stream of time and keyboard events, change the time/date/alarm, access the graphics library to draw on the stubbed LCD display, and run a stubbed Monochron application ahead of building the actual firmware. In combination with the standard gdb debugger and a gdb GUI frontend this is a powerful means to test specific functionality and find and solve bugs.

The mchron interpreter supports named variables representing numeric values, repeat and if-then-else logic constructs, and basic mathematical expression evaluation for numeric command arguments. Commands for mchron can be prepared using a standard text editor and saved as a script file. This script file can then be loaded and executed in mchron, which comes in handy for creating demos and standard test suites for clocks and graphics functions.

An example on how to use the mchron command line tool is the following scenario, using only five mchron commands:

- mchron> **cs 1**
Select the first built-in clock plugin, being an analog clock.
The clock will initialize and paint itself on the stubbed LCD device, yet remains static.
- mchron> **ap 0**
Set the stubbed alarm switch position to off.
In case it was switched on, the clock will now display the mchron date instead of the alarm time.
- mchron> **e s ../script/minutes.txt**
Execute the mchron commands from a text file to feed the clock with 60 minute timestamps between 16.00pm and 16.59pm.
Each timestamp will differ a minute from the previous one and will be displayed on the stubbed LCD device for 0.2 seconds.
We use this script to see how the clock reacts to changes in minutes.
- mchron> **ts 23 59 15**
Set the mchron time to nearly midnight.
The clock will update itself to the new time but remains static.
- mchron> **cf n**
Feed the clock with a continuous stream of time and keyboard events.
The clock is now started in a test environment that is rather similar to the actual Monochron application, so it will update itself every second.
We will now be able to see on the stubbed LCD device whether the clock correctly processes a day change in its date area.

1.4 Debian Linux and AVR

Emuchron is developed in Debian 6 and has been verified to work in Debian 7 and 8. The table below provides the details of the several environments in which Emuchron is verified to work at the time of writing this document. This list is not actively maintained.

Debian version and host	Version info
Version: Debian 6 32-bit Host: Windows-7 Professional VM Memory: 512MB	VirtualBox: 5.0.8 Linux kernel: 2.6.32-5-686 gcc/avr-gcc: 4.4.5/4.3.5
Version: Debian 7 32-bit Host: Windows-7 Professional VM Memory: 512MB	VirtualBox: 5.0.8 Linux kernel: 3.2.0-4-486 gcc/avr-gcc: 4.7.2/4.7.2
Version: Debian 8 64-bit Host: Windows-7 Professional VM Memory: 1GB	VirtualBox: 5.0.8 Linux kernel: 3.16.0-4-amd64 gcc/avr-gcc: 4.9.2/4.8.1
Version: Debian 6 32-bit Host: OS-X 10.11 VM Memory: 512MB	VMware Fusion: 7.1.3 Linux kernel: 2.6.32-5-686 gcc/avr-gcc: 4.4.5/4.3.5
Version: Debian 7 64-bit Host: OS-X 10.11 VM Memory: 512MB	VMware Fusion: 7.1.3 Linux kernel: 3.2.0-4-amd64 gcc/avr-gcc: 4.7.2/4.7.2
Version: Debian 8 64-bit Host: OS-X 10.11 VM Memory: 1GB	VMware Fusion: 7.1.3 Linux kernel: 3.16.0-4-amd64 gcc/avr-gcc: 4.9.2/4.8.1

Table 1: The Emuchron runtime environments for Debian and AVR

Note: 64-bit is supported on Debian 7 and 8 only. 32-bit is supported on Debian 6 and 7 only.

Note: The information above shows up-to-date version info at the time of writing. In the development stage of Emuchron older versions of VM tools, Linux kernels and hosts were used as well.

1.5 Migrating from Emuchron v1.x to v2.x

Compared to v1.x, both the core of the Monochron firmware and the clock plugin framework are left unchanged. This means that clocks plugins created in v1.x are expected to function properly in v2.x without any code changes.

However, in v2.x the v1.x modules ratt.c/ratt.h [firmware] are renamed to monomain.c/monomain.h [firmware]. This means that clock plugins must replace an include reference in order to build properly in v2.x. See below an example for clock plugin nerd.c [firmware/clock].

```

//*****
// Filename : 'nerd.c'
// Title    : Animation code for MONOCHRON nerd clock
//*****

#ifdef EMULIN
#include "../emulator/stub.h"
#endif
#ifdef EMULIN
#include "../util.h"
#endif
#include "../ks0108.h"
-- #include "../ratt.h"
++ #include "../monomain.h"
#include "../glcd.h"
#include "../anim.h"
#include "nerd.h"

```

From an emulator perspective, specific functionality of the mchcron interpreter is modified in v2.x. This requires changes in command scripts that are created in v1.x. Find below an overview.

In Emuchron v2.x, variables are assigned a value using an expression based on the assignment operator.

```
# Emuchron v1.x: assign value to variable using two command arguments
vs x 15

# Emuchron v2.x: assign value to variable using assignment operator
vs x=15
```

In Emuchron v2.x, the 'rw' (repeat-while) command is replaced by 'rf' (repeat-for). The syntax structure of the new repeat command is improved and more or less conform a 'C'-style `for()` construct.

```
# Emuchron v1.x: repeat while
rw x < 128 0 1
  # Do something
rn

# Emuchron v2.x: repeat for
rf x=0 x<128 x=x+1
  # Do something
rn
```

In Emuchron v2.x, the operator to check for inequality of argument values is changed from '<>' into 'C'-style operator '!= '.

```
# Emuchron v1.x: repeat while with '<>' comparison
rw y <> 64 0 1
  # Do something
rn

# Emuchron v2.x: repeat for with '!=' comparison
rf y=0 y!=64 y=y+1
  # Do something
rn
```

In Emuchron v2.x, the wait command uses a granularity of 0.001 sec.

```
# Emuchron v1.x: wait 0.25 sec (granularity = 0.01 sec)
w 25

# Emuchron v2.x: wait 0.25 sec (granularity = 0.001 sec)
w 250
```

2 The Emuchron project

2.1 The project folder structure

The Emuchron project uses the following folder structure.

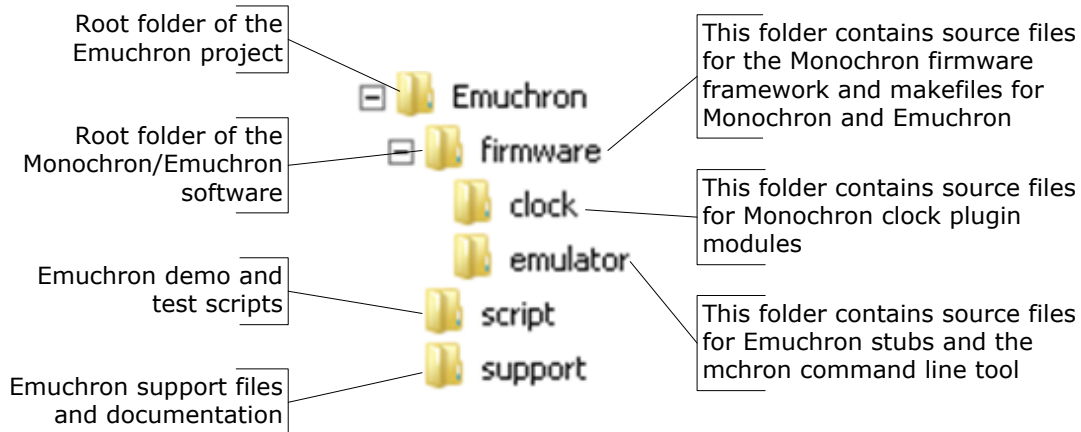


Figure 1: The Emuchron project folder structure

2.2 Monochron firmware high-level runtime environment

The following graph depicts the Monochron runtime environment, including references to source files being used to build the firmware.

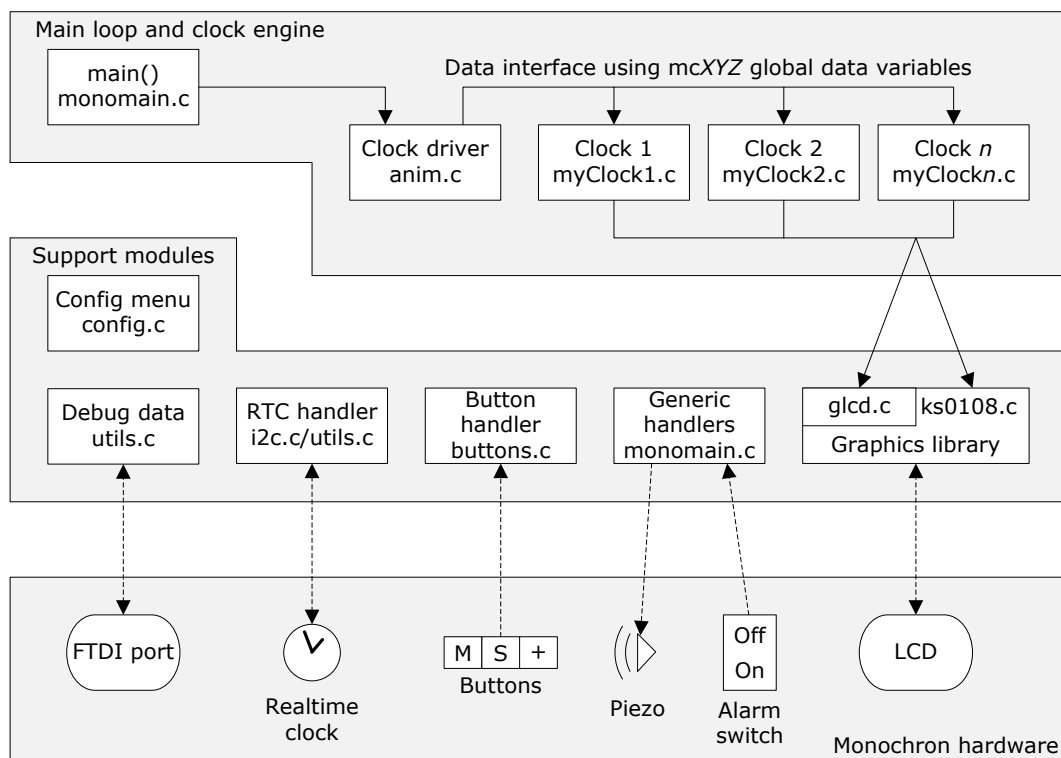


Figure 2: The Monochron runtime environment

Note that this high-level view only shows how the several modules are linked to another from a clock plugin perspective, and is not fully correct. For example,

the graph does not show that upon startup, `main()` in `monomain.c` [firmware] will take care of initializing the LCD hardware via the graphics library.

The following modules apply:

Module	Description
<code>anim.c</code> [firmware]	In module <code>anim.c</code> we find the handler for all plugin clocks. It will take care of initializing and updating clocks and switching between clocks. It prepares the software interface to the plugin clocks. It is responsible for most of the mcXYZ data interface to the clock plugins.
<code>buttons.c</code> [firmware]	The button support handler module takes care of button press and button hold events and mapping these into a software state. Its functionality is used in <code>monomain.c</code> [firmware].
<code>config.c</code> [firmware]	This support module contains the main entry for the configuration menu as used in the Monochron application. It is activated in <code>main()</code> by pressing the 'M' button.
<code>glcd.c</code> [firmware]	The high-level graphics library. It contains functions to draw text, lines, dots, (filled) circles and (filled) rectangles. This module does not contain hardware agnostic code and uses <code>ks0108.c</code> [firmware] for the actual interface to the LCD.
<code>i2c.c</code> [firmware]	In module <code>i2c.c</code> we find the interface to the real time clock (RTC).
<code>ks0108.c</code> [firmware]	The low-level graphics library. It contains functions to initialize the LCD, clear it, write data to and read data from the LCD. This module interacts with the LCD hardware.
<code>monomain.c</code> [firmware]	In module <code>monomain.c</code> we find the <code>main()</code> function. Next to <code>main()</code> , <code>monomain.c</code> contains much additional functionality related to interrupt handlers, handling the real time clock, the alarm and snooze, the piezo speaker and handling the state of the alarm switch. The <code>main()</code> function contains an infinite loop and will interact with the clock driver in <code>anim.c</code> [firmware] and the clock configuration menu in <code>config.c</code> [firmware] when appropriate. It is responsible for a subset of the mcXYZ data interface to the clock plugins.
<code>myClockx.c</code> [firmware/clock]	A Monochron plugin clock. Based on the mcXYZ data interface the module is responsible for drawing and updating itself on the LCD. This is where functional clock code resides.
<code>utils.c</code> [firmware]	This support module contains formatting utility routines used by the RTC interface. It also provides a means to format and send debug strings over the FDTI port at runtime. Reading and logging the FTDI debug strings requires a terminal application on the connected computer. This method used to be the only method of debugging a Monochron application.

Table 2: The Monochron runtime environment

2.3 Emuchron emulator high-level runtime environment

The following graph depicts the Emuchron emulator environment, including references to source files being used to build the software.

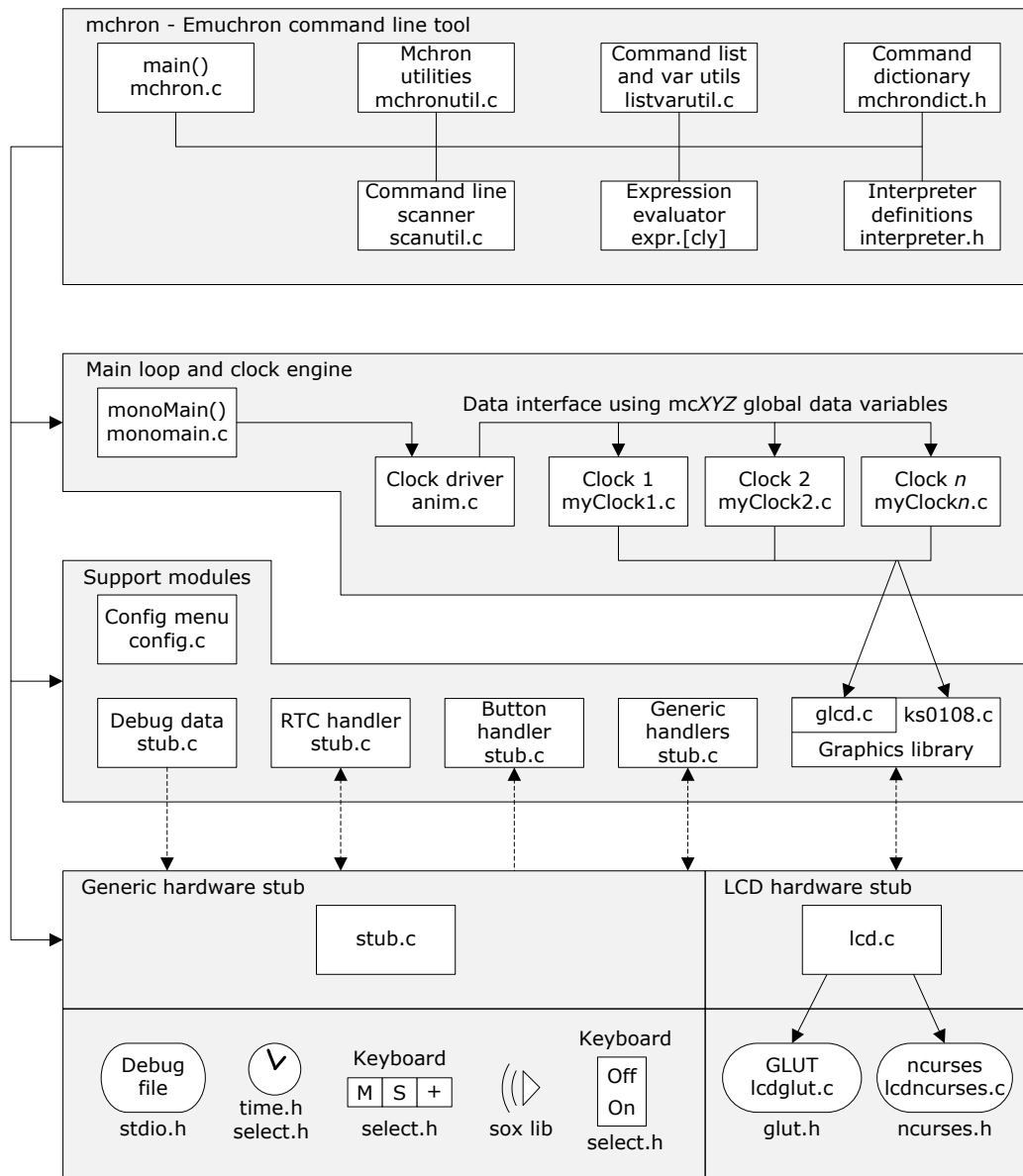


Figure 3: The Emuchron runtime environment

Again, note that this is a high-level view only showing how the several modules are linked to another from a clock plugin perspective.

Compared to figure 2 notice the following:

- On top of the environment we find the mchron.c [firmware/emulator] module with the `main()`. It controls the entire emulator environment using the mchron command line interface.
- The 'Main loop and clock engine' block (monomain.c [firmware] – anim.c [firmware] – myClockx.c [firmware/clock]) and its link to glcd.c [firmware] is unaffected, conform the emulator requirement that clock plugins should be as much as possible free from hardware stubs.

- Most important is that code in `myClockx.c` [firmware/clock] and `glcd.c` [firmware] does not require any stub functionality.
- In `monomain.c` [firmware] the `main()` has been renamed into `monoMain()` but besides that effectively remains the same function.
Note that the `mchron` command line tool can start a stubbed Monochron application using a call to `monoMain()`.
 - All hardware has been stubbed by `stub.c` [firmware/emulator] and `lcd.c` [firmware/emulator] and is emulated using off-the-shelf Linux libraries.
 - Monochron modules like `i2c.c` [firmware], `button.c` [firmware] and `utils.c` [firmware] are not part of the Emuchron environment. Their functionality has been incorporated in `stub.c` [firmware/emulator]. This means that changes in these modules cannot be tested in Emuchron.
 - There are two LCD stub devices defined, being OpenGL2/GLUT and `ncurses`. Select the device to use on `mchron` startup, or use both, thus showing duplicate output in two separate windows. Each of these devices has its pros and cons.

The following new modules apply:

Module	Description
<code>expr.c/expr.l/expr.y</code> [firmware/emulator]	The flex (<code>expr.l</code>) and bison (<code>expr.y</code>) modules implement an expression evaluator. The code generated by flex and bison code is included in the master module (<code>expr.c</code>) and compiled into a separate expression evaluator object. The following elements are supported: <ul style="list-style-type: none"> - Operators +, -, *, /, % (modulo), ^ (power), =, () - Logic operators <, >, <=, >=, ==, !=, &&, , ?: ('C'-style ternary operator) - Functions <code>abs()</code>, <code>cos()</code>, <code>frac()</code>, <code>int()</code>, <code>sin()</code> - Constants <code>null</code>, <code>pi</code>, <code>true</code>, <code>false</code>
<code>interpreter.h</code> [firmware/emulator]	This module defines the core structures and constants for the <code>mchron</code> interpreter.
<code>listvarutils.c</code> [firmware/emulator]	This module implements the utilities to build and cleanup command lists, as well as the administration of the named interpreter variables.
<code>lcd.c</code> [firmware/emulator]	The <code>lcd</code> module implements the stubbed LCD data structures and acts as a driver for the two LCD device stubs. It initializes the requested LCD stub devices and dispatches LCD updates to each of those. Note: As <code>lcd.c</code> implements fixed function calls to each of the two LCD devices, such a device can be considered as an LCD plugin. Another LCD device type can be added to <code>lcd.c</code> as long as it publishes functions similar to the GLUT and <code>ncurses</code> modules.
<code>lcdglut.c</code> [firmware/emulator]	This module implements an OpenGL2/GLUT LCD device. The GLUT device is implemented using a separate thread, meaning that the GLUT window is updated asynchronously from the <code>mchron</code> application. As a result, the GLUT interface is less suited for use in a debugging session when LCD output is essential. The upside however is that the GLUT interface does not require end-user setup, the GLUT window can be resized at will while retaining the 2:1 aspect ratio and that the interface supports changes in LCD backlight settings.

Module	Description
lcdncurses.c [firmware/emulator]	This module implements an ncurses LCD device. The ncurses device runs in the same main thread as mchron. As such, LCD updates need to be actively flushed in ncurses at the end of an application cycle, thus making the LCD device always in-sync with the mchron application. This makes the ncurses interface much better suited for use in a debugging session when LCD output is essential. Disadvantages of the ncurses device are that in order to make the ncurses device work properly it requires (one-time only) configuration steps in GNOME, its window cannot be freely resized (but we can use keyboard shortcuts instead), it does not support LCD backlight changes and the ncurses library does not play nice with gdb (refer to section 6.3.2) and the readline library (refer to section 7.7).
mchron.c [firmware/emulator]	The mchron module implements the command line interface to the Emuchron emulator environment and all command handlers. Each mchron command will have its associated command handler in this module. The command line interface supports the use of named variables, basic repeat loop and if-then-else logic constructs, basic expression evaluation for numeric command arguments and executing scripts that are prepared in plain text files. An overview of the command set is found in section 5.8.
mchroundict.h [firmware/emulator]	The mchroundict header module creates the mchron command dictionary. The command dictionary is an aggregated set of structures of domain values, command arguments, commands in a command group and finally the dictionary itself that consists of a collection of command groups.
mchronutil.c [firmware/emulator]	Whereas the mchron module implements the command handlers, this module implements several mchron utility functions, as well as mchron initialization and signal handler functionality.
scanutil.c [firmware/emulator]	The scanutil module implements command input streams (command line and file), the command line scanner, and functions to access the command dictionary.
stub.c [firmware/emulator]	The stub module is the heart of the Emuchron emulator functionality. It contains stubs replacing all Monochron hardware except the LCD.

Table 3: The Emuchron modules

2.4 Monochron main loop, buttons and clocks

The Monochron main loop is coded in `main()` in `monomain.c` [firmware]. In combination with functionality in `anim.c` [firmware] it handles initializing clocks, updating clocks, switching between clocks and handling button presses. The functional behavior of clocks as implemented in these two modules depends on how many clocks have been configured in the static `monochron[]` array in `anim.c` [firmware], and whether or not for a clock a public button handler function is exposed. Refer to section 2.12 where the structure of the static `monochron[]` array is explained.

Generic functionality in `main()`:

- A single loop cycle is executed every 75 msec.
This is defined by `#define ANIMTICK_MS` in `monomain.h` [firmware].

- In a single loop cycle button presses are scanned after which one or more functions in `anim.c` [firmware] are called to update the current active clock, to switch to and initialize the next clock or to handle a button press.

Per application loop cycle when not in alarming/snoozing state, in case only a single clock is configured in the static `monochron[]` array:

Event	Action
Press 'M' button	Enter the clock configuration menu in <code>config.c</code> [firmware]. After exit of configuration menu: invoke <code>init()</code> for clock with <code>DRAW_INIT_FULL</code> invoke <code>cycle()</code> for clock
Press 'S' button	if <code>button()</code> is defined for clock then invoke <code>button()</code> for clock end-if invoke <code>cycle()</code> for clock
Press '+' button	if <code>button()</code> is defined for clock then invoke <code>button()</code> for clock end-if invoke <code>cycle()</code> for clock
No button pressed	invoke <code>cycle()</code> for clock

Table 4: Single loop cycle actions for a single-clock configuration

Per application loop cycle when not in alarming/snoozing state, in case multiple clocks are configured in the static `monochron[]` array:

Event	Action
Press 'M' button	Enter the clock configuration menu in <code>config.c</code> [firmware]. After exit of configuration menu: invoke <code>init()</code> for clock with <code>DRAW_INIT_FULL</code> invoke <code>cycle()</code> for clock
Press 'S' button	if <code>button()</code> is defined for clock then invoke <code>button()</code> for clock else select next clock in <code>monochron[]</code> (round-robin) invoke <code>init()</code> for clock with <code>monochron[].initType</code> end-if invoke <code>cycle()</code> for clock
Press '+' button	select next clock in <code>monochron[]</code> (round-robin) invoke <code>init()</code> for clock with <code>monochron[].initType</code> invoke <code>cycle()</code> for clock
No button pressed	invoke <code>cycle()</code> for clock

Table 5: Single loop cycle actions for a multi-clock configuration

Per application cycle when in alarming/snoozing state, regardless the number of clocks configured in the static `monochron[]` array:

Event	Action
Press 'M' button	stop alarming/snoozing (by forcing alarm timeout) invoke <code>cycle()</code> for clock

Event	Action
Press 'S' button	reset snooze timer timeout invoke <code>cycle()</code> for clock
Press '+' button	reset snooze timer timeout invoke <code>cycle()</code> for clock
No button pressed	invoke <code>cycle()</code> for clock

Table 6: Single loop cycle actions when in alarming/snoozing state

Note: For more information on the snooze timer timeout value refer to section 2.8.

2.5 Monochron variables for clock plugins

When any of the published clock functions is invoked, it can make use of the following variables below. These variables are defined in `anim.c` [firmware] and represent a stable software representation of the state of the Monochron clock.

Variable	Description
<code>mcAlarmH</code> <code>mcAlarmM</code>	The active alarm time (hour, min), regardless whether the alarm switch position is on or off.
<code>mcAlarming</code>	Value: <code>GLCD_TRUE</code> / <code>GLCD_FALSE</code> Indicates whether the clock is alarming/snoozing (<code>GLCD_TRUE</code>) or not (<code>GLCD_FALSE</code>).
<code>mcAlarmSwitch</code>	Value: <code>ALARM_SWITCH_NONE</code> / <code>ALARM_SWITCH_ON</code> / <code>ALARM_SWITCH_OFF</code> Current on/off state of the alarm switch. <ul style="list-style-type: none"> <code>ALARM_SWITCH_NONE</code> This value clears the stored value and forces an alarm switch change event in <code>mcUpdAlarmSwitch</code>. It is normally set by a clock in its <code>init()</code> function. Note: The clock <code>cycle()</code> should only see values <code>ALARM_SWITCH_ON</code> and <code>ALARM_SWITCH_OFF</code>. <code>ALARM_SWITCH_ON</code> Indicates that the alarm switch position is switched on. <code>ALARM_SWITCH_OFF</code> Indicates that the alarm switch position is switched off.
<code>mcBgColor</code> <code>mcFgColor</code>	Value: <code>ON</code> (white pixel) / <code>OFF</code> (black pixel) The variables holding the background and foreground draw color. The value of both variables are mutually exclusive. The Monochron configuration menu can swap the values between the two variables. A clock, when it has properly implemented its drawing graphics with these variables, can freely swap between showing itself white-on-black and black-on-white without any code changes.
<code>mcClockInit</code>	Value: <code>GLCD_TRUE</code> / <code>GLCD_FALSE</code> Indicates that a clock must initialize itself. It is set prior to calling the clock <code>init()</code> and is reset after executing a clock <code>cycle()</code> .
<code>mcClockNewTH</code> <code>mcClockNewTM</code> <code>mcClockNewTS</code> <code>mcClockNewDD</code> <code>mcClockNewDM</code> <code>mcClockNewDY</code>	The new Monochron clock time (hour, min, sec) and date (day, month, year).

Variable	Description
mcClockOldTH mcClockOldTM mcClockOldTS mcClockOldDD mcClockOldDM mcClockOldDY	The previous Monochron clock time (hour, min, sec) and date (day, month, year).
mcClockPool mcMchronClock	mcClockPool is a pointer to the clock array and mcMchronClock is the current index in that array. In Monochron the clock array being used is <code>monochron[]</code> in <code>anim.c</code> [firmware]. In Emuchron the clock array being used is <code>emuMonochron[]</code> in <code>mchron.c</code> [firmware/emulator].
mcClockTimeEvent	Value: GLCD_TRUE / GLCD_FALSE Indicates that the time has changed. This event must be handled in the clock <code>cycle()</code> as it is reset every clock cycle.
mcCycleCounter	A counter that is incremented every clock cycle. It can be used as input for a random number generator or serve as a base for blinking LCD elements.
mcU16Util[1..4] mcU8Util[1..4]	Value: Free for use in an active clock Whenever a clock plugin has a need for global data, instead of defining that in its own module, these variables can be used. There are in total eight variables, of which four are 16 bit wide and four are 8 bit wide. An example of its usage can be found in most demo clocks where <code>mcU8Util1</code> is used to store the blinking state of the alarm draw area when alarming or snoozing. Not that these variables are under control of the active clock and as such must be initialized, set and processed in clock code.
mcUpdAlarmSwitch	Value: GLCD_TRUE / GLCD_FALSE Signals a change in the alarm switch position. This event must be handled in the clock <code>cycle()</code> as it is reset every clock cycle. Use it in combination with <code>mcAlarmSwitch</code> .

Table 7: The Monochron variables for clock plugins

In a clock plugin the population of variables `mcClockNewXY` and `mcClockOldXY` are tied to variables `mcClockTimeEvent` and `mcClockInit` as described below.

Variables	Impact
mcClockTimeEvent = GLCD_FALSE mcClockInit = GLCD_FALSE	mcClockOldXY holds the previous set timestamp mcClockNewXY holds the last created timestamp
mcClockTimeEvent = GLCD_TRUE mcClockInit = GLCD_FALSE	mcClockOldXY holds the previous set timestamp mcClockNewXY holds the current timestamp
mcClockTimeEvent = GLCD_FALSE mcClockInit = GLCD_TRUE	mcClockOldXY holds the last created timestamp mcClockNewXY holds the last created timestamp
mcClockTimeEvent = GLCD_TRUE mcClockInit = GLCD_TRUE	mcClockOldXY holds the current timestamp mcClockNewXY holds the current timestamp

Table 8: The Monochron time and initialization variables

2.6 The glcd graphics library enhancements

This project is based on the original Monochron pong firmware. To enhance the graphics capabilities of clocks a number of glcd functions have been added, modified or enhanced. In general, a high-level glcd graphics function can be accessed directly via the mchron command line tool for testing purposes.

To test these enhancements, a dedicated clock plugin has been created that runs glcd performance tests on Monochron hardware.

2.6.1 Overview of high-level glcd functions

The functions are found in glcd.c [firmware]. Please find below a rough overview of the changes when compared to the original Monochron pong firmware.

Function	Description
-Generic-	The interface and code of legacy glcd functions is updated to include parameter color that is required for implementing the mcBgColor and mcFgColor functionality.
glcdCircle()	Superseded by glcdCircle2().
glcdCircle2()	Similar to glcdCircle() but in addition supports drawing a dotted (1:2 and 1:3) circle outline.
glcdClearDot() glcdSetDot()	Superseded by glcdDot().
glcdDot()	Draw a dot.
glcdFillCircle2()	Draw a filled circle with several fill patterns. Note that this function does not draw the circle outline. An additional call to glcdCircle2() is required for drawing a complete filled circle.
glcdFillRectangle2()	Similar to the existing glcdFillRectangle() function that is retained, yet supports several fill patterns.
glcdGetWidthStr()	Utility function that returns the width of a string in unscaled display pixels.
glcdPrintNumberBg() glcdPrintNumberFg() glcdPutStrFg() glcdWriteCharFg()	Proxy functions for legacy functions glcdPrintNumber(), glcdPutStr() and glcdWriteChar() with a reduced interface regarding the draw color, allowing to optimize code on object size.
glcdPutStr2()	For background information consider function glcdPutStr(). It draws text very fast but is limited in use as the text y-position is limited to eight character lines (multiple of 8 vertical pixels) and supports a non-proportional 5x7 font only. In contrast, the new glcdPutStr2() function draws horizontal text at any (x,y) pixel location and supports an additional 5x5 proportional font. It returns the string width of horizontal pixels drawn. Note that glcdPutStr() is still supported as it is lightweight, fast and heavily used in config.c [firmware].
glcdPutStr3()	This is an extension to glcdPutStr2() and supports independent horizontal and vertical font scaling.
glcdPutStr3v()	Similar to glcdPutStr3(). However, this function draws text vertically (top-down or bottom-up). It returns the string width of vertical pixels drawn.

Table 9: Enhancement overview of the high-level glcd library

2.6.2 The lcdLine[] buffer

It turns out that the Monochron firmware and/or the LCD display is slow in switching between LCD read and LCD write operations.

To reduce switching between LCD read and write operations, several graphics functions have implemented a method to read all relevant LCD bytes from a single LCD byte row in buffer lcdLine[] first, then apply changes to the buffered data and then write the modified data back to the LCD.

This method greatly reduces switching between LCD read and LCD write operations and significantly improves the speed of the graphics interface to the LCD. The downside of this method is that 128 bytes of stack RAM (out of 2K) is constantly allocated for this purpose.

2.6.3 Text fonts and font scaling

Specific glcd text functions allow painting text in two fonts.

In glcd.h [firmware] the following fonts are defined:

Font	Description
FONT_5X5P	A 5x5 proportional font. It supports only uppercase characters. The font is defined in font5x5p.h [firmware]. Note: A few non-standard characters in this font are remapped to special graphics characters as required by clocks.
FONT_5X7N	A 5x7 non-proportional font. It supports both uppercase and lowercase characters. The font is defined in font5x7.h [firmware]. Note: This is the unmodified original Monochron font.

Table 10: Text font overview

Next to that, specific glcd text functions allow individual horizontal and vertical font scaling.

Refer to the screenshots below. All text is drawn using a single glcd graphics function, being `glcdPutStr3()`.



2.6.4 Text orientation

The glcd text functions allow painting text in several orientations. The (x,y) start location for text to be painted is linked to the position of the top-left font pixel of the first character.

In glcd.h [firmware] the following text orientations are defined:

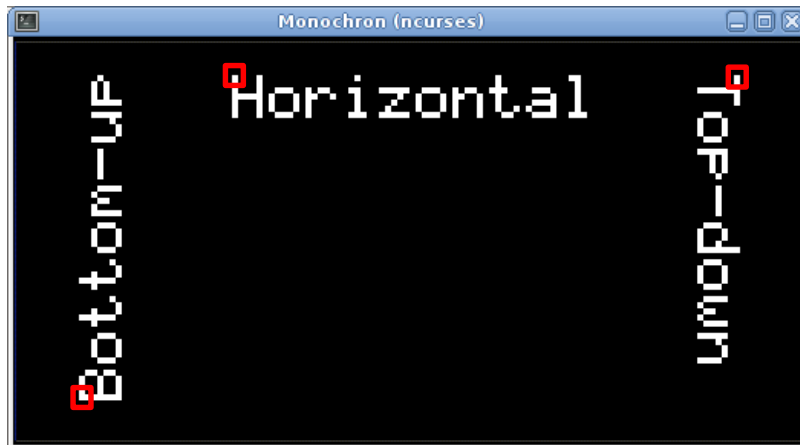
Text orientation	Description
ORI_HORIZONTAL	Paint the text horizontally.
ORI_VERTICAL_BU	Paint the text vertically in a bottom-up direction.
ORI_VERTICAL_TD	Paint the text vertically in a top-down direction.

Table 11: Text orientation overview

Enter the following mchcron commands.

```
mchcron> pa f 35 5 5x7n h 1 1 Horizontal
mchcron> pa f 10 57 5x7n b 1 1 Bottom-up
mchcron> pa f 117 5 5x7n t 1 1 Top-down
```

This will yield the following output. Note the markers identifying the pixel draw start location for each string.



2.6.5 Fill patterns

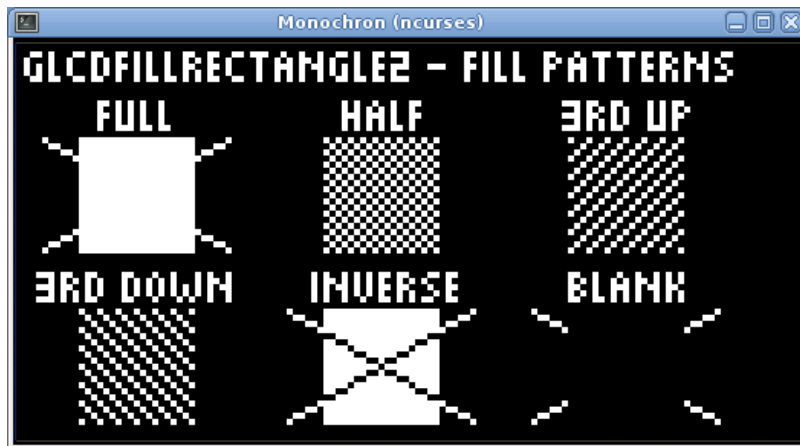
The `glcdFillRectangle2()` and `glcdFillCircle2()` functions provide a method to fill an area with several fill patterns.

In `glcd.h` [firmware] the following fill patterns are defined:

Pattern	Description
FILL_FULL	The area is filled with the given paint color.
FILL_HALF	The area is filled with a 50% fill pattern using the given paint color.
FILL_THIRDDUP	The area is filled with a 1/3 rd pattern using the given paint color giving an upward illusion.
FILL_THIRDDOWN	The area is filled with a 1/3 rd pattern using the given paint color giving a downward illusion.
FILL_INVERSE	The area is inverted. Note: This fill pattern is not supported in <code>glcdFillCircle2()</code> .
FILL_BLANK	The area is filled with the inverted value of the given paint color.

Table 12: Fill pattern overview

Refer to the screenshot below for examples of each fill pattern.



2.6.6 Fill alignment

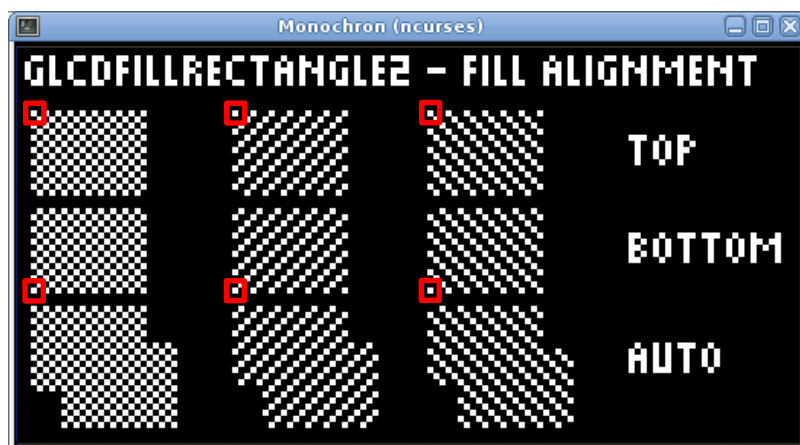
The `glcdFillRectangle2()` function supports a fill alignment option for fill patterns `FILL_HALF`, `FILL_THIRDDUP` and `FILL_THIRDDOWN`.

In `glcd.h` [firmware] the following fill alignments are defined:

Alignment	Description
<code>ALIGN_TOP</code>	The top-left pixel of the fill area is filled with the given paint color.
<code>ALIGN_BOTTOM</code>	The bottom-left pixel of the fill area is filled with the given paint color.
<code>ALIGN_AUTO</code>	A pixel in the fill area is filled with the given paint color relative to pixel (0,0) being assumed to be filled. This alignment will make fill areas overlap properly.

Table 13: Fill alignment overview

Refer to the screenshot below for an example for every fill alignment option. Note the markers identifying the fill alignment pixels.



2.6.7 Circle draw patterns

The `glcdCircle2()` function provides a method to draw a circle using several patterns.

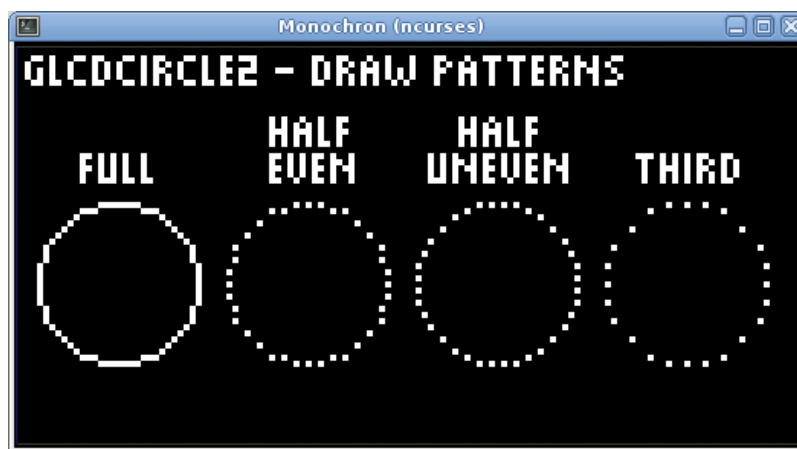
Note that the method to determine which pixels are being drawn is rather crude. The quality of the non-full draw patterns will vary depending on the radius and center of the circle being drawn.

In glcd.h [firmware] the following circle draw patterns are defined:

Pattern	Description
CIRCLE_FULL	The circle is fully drawn with the given paint color.
CIRCLE_HALF_E	The circle is drawn 50% with the given paint color. Only the even circle pixels are drawn, making it the inverse of CIRCLE_HALF_U when drawn at the same location.
CIRCLE_HALF_U	The circle is drawn 50% with the given paint color. Only the uneven circle pixels are drawn, making it the inverse of CIRCLE_HALF_E when drawn at the same location.
CIRCLE_THIRD	The circle is drawn with 1/3 rd of the pixels with the given paint color.

Table 14: Circle draw pattern overview

Refer to the screenshot below for examples for each draw type.



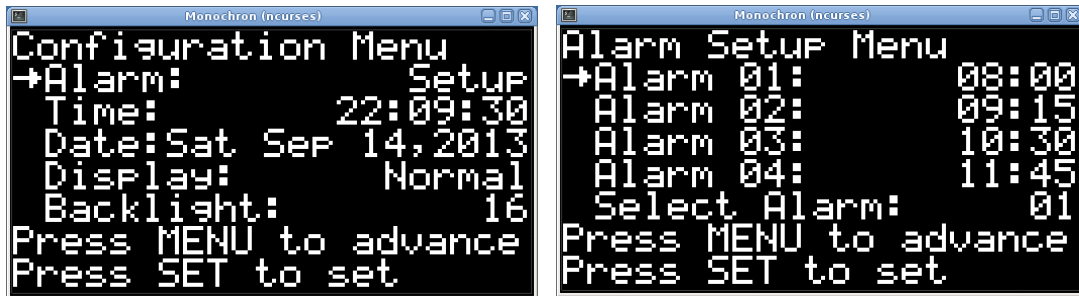
2.7 Monochron configuration screens

In Emuchron the method of navigating through the configuration menu, selecting items for editing and modifying values has not changed. However, compared to the original Monochron firmware, a number of changes in the configuration module are applied.

- The keypress hold and increment timers have been modified to decrease the keypress hold delay and increase the value scrolling speed. For minute, second and year elements, increments will double after 10 regular press-hold increments.
- The configuration screen no longer 'blinks' upon pressing a button.
- The backlight setting is put under keypress hold control.
- Whereas in the original firmware every incremental change is saved in eeprom, it now applies only to the final value.
- Whereas the original firmware supports a single alarm time only, it now supports a separate alarm setup menu page that allows setting four independent alarm times and a selector determining which alarm is active.
- The original firmware allows configuring the format of the time and date within the configuration module. This is no longer supported. Time will now use the 24 hour HH:MM format. Date will now use a full day of the week, month, day and year format. See below.
- The new firmware supports configuring the display behavior of the application which is either 'Normal' (white pixels on black background) or

'Inverse' (black pixels on white background).

For code refer to config.c [firmware].



Note: In the main configuration menu (left screendump), upon pressing the 'Set' button at the 'Alarm' item, the alarm setup menu (right screendump) is accessed.

2.8 Monochron two-tone and Mario alarm melodies

The original firmware supports a simple yet effective single-tone alarm. In Emuchron this has been replaced by two distinctive alarm melodies.

The first is a two-tone alarm, which is basically an enhancement of the single-tone alarm. The tones and tone duration are configured using the definitions in monomain.h [firmware] below.

```
// Two-tone alarm beep
#define ALARM_FREQ_1 4000
#define ALARM_FREQ_2 3750
#define ALARMTICK_MS 325
```

The second melody is Mario, the world's most famous chiptune. For this refer to mario.h and mariotune.h [firmware].

The two alarm melodies are mutually exclusive. Switching between the two is done by enabling or disabling build option `-DMARIO` in the two makefiles. See an excerpt from the Monochron Makefile [firmware] below where is chosen to use the Mario alarm. Refer to chapter 4 on how to build Monochron firmware and the emulator.

```
# Uncomment this if you want a Mario tune alarm instead of a two-tone alarm.
# There is a similar switch in MakefileEmu that should be in sync with this one.
# Note: This will cost you ~615 bytes of Monochron program and data space.
MARIO = -DMARIO
```

Alarming and snoozing timeouts are controlled by the following defines in monomain.h [firmware]. Note that for the emulator reduced timeouts are specified.

```
// Set timeouts for snooze and alarm (in seconds)
#ifndef EMULIN
#define MAXSNOOZE 600
#define MAXALARM 1800
#else
// In our emulator we don't want to wait that long
#define MAXSNOOZE 25
#define MAXALARM 65
#endif
```

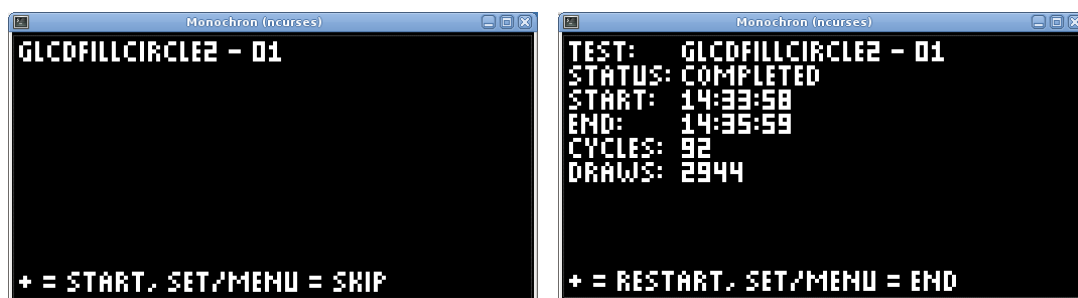
2.9 Performance tests for high-level glcd functions

Modifying a high level glcd function is mostly done for performance and/or object size optimization reasons. In order to verify whether code changes impact the draw performance of a glcd function, a dedicated clock plugin has been created that, instead of providing a functional clock, allows running glcd performance tests on Monochron hardware.

The performance test module covers most of the high-level glcd functionality. The tests are split-up in tests suites per glcd function where a test suite contains one or more individual tests. Using the Monochron buttons one can navigate through a menu-like structure of test suites and individual tests within a suite, or abort a running test.

In appendix B test results are described and discussed for several performance test runs.

For code refer to `perftest.c` [firmware/clock].



2.10 Demo and test mchcron command scripts

In node [script] mchcron demo and test command scripts are available. Refer to section 5.8.6 on how to execute a command script.

Below is an overview of those considered most relevant.

Script	Description
alarm.txt [script]	This script is used for testing a clock plugin. It will run through all minutes in a day and have each minute displayed in the alarm area of the clock of choice. It requires preset values for two variables that control the minute skip size and the display time per generated timestamp. Refer to the script itself for an example on how to use it. See also time-hm.txt that focuses on the clock time instead of alarm time.
circleX.txt [script]	A total of five scripts for testing high-level glcd graphics. The first three scripts verify the correctness of the circle functions.
demo.txt [script]	This script is a shell that executes other scripts that demo the graphic capabilities of the enhanced high-level glcd library. Some of the other scripts listed here are executed via demo.txt.
lineX.txt [script]	A total of seven scripts for testing high-level glcd graphics. It verifies the correctness of the line function.
rectangleX.txt [script]	A total of seven scripts for testing high-level glcd graphics. It verifies the correctness of the rectangle functions.
time-hm.txt [script]	This script is used for testing a clock plugin. It will run through all minutes in a day and have each minute displayed in the clock of choice. It requires preset values for two variables that control the minute skip size and the display time per generated timestamp. Refer to the script itself for an example on how to use it. See also time-ms.txt and alarm.txt.

Script	Description
time-ms.txt [script]	This script is used for testing a clock plugin. It will run through all seconds in one hour and have each second displayed in the clock of choice. It requires preset values for two variables that control the seconds skip size and the display time per generated timestamp. Refer to the script itself for an example on how to use it. See also time-hm.txt.
year.txt [script]	This script is used for testing a clock plugin. It will run through all days in a leap year and have a clock display each day in its date area. It requires a preset value for a variable that controls the display time per generated date. Refer to the script itself for an example on how to use it.

Table 15: Relevant command scripts overview

2.11 The pre-built monochron.hex firmware

This project contains a pre-built monochron.hex [firmware] firmware file using avr-gcc 4.3.5 (Debian 6).

As all clocks [firmware/clock] combined will result in a firmware file that exceeds the Monochron firmware size limit a selection has been made. Refer to the contents of `monochron[]` in `anim.c` [firmware] to see which clocks are configured and `Makefile` [firmware] to see which alarm melody is used. Refer to section 4.3 on how to upload firmware to Monochron.

2.12 Quick guide into the `clockDriver_t` structure

The `clockDriver_t` structure is the basis of the static `monochron[]` and `emuMonochron[]` arrays and contains the public functions of configured clock plugins. Below is detailed info on the structure members.

Refer to `anim.c` [firmware] and `mchron.c` [firmware/emulator] for examples on how the arrays are populated.

The structure elements are as follows.

Element	Description
<code>clockId</code>	This is the unique clock Id assigned to a clock.
<code>initType</code>	The initialization mode that is forwarded to the <code>init()</code> function of a clock. It has two distinctive values as defined in <code>anim.h</code> [firmware]. <ul style="list-style-type: none"> <code>DRAW_INIT_FULL</code> The clock must begin from scratch, so it should clear the entire LCD display and make a complete graphic build-up of the clock. <code>DRAW_INIT_PARTIAL</code> The preceding clock in the clock array has a shared clock layout with the new one. So, instead of rebuilding the clock from scratch we can keep certain graphic elements as-is and therefore need to clear and draw only those elements that differ. This will result in a faster and smoother graphic build-up of the new clock. For examples refer to the clocks defined in <code>analog.c</code> [firmware/clock] and <code>digital.c</code> [firmware/clock].
<code>init()</code>	This is the published initialization function for a clock. It is invoked via <code>anim.c</code> [firmware] when the clock needs to initialize itself.
<code>cycle()</code>	This is the published cycle function for a clock. It is invoked via <code>anim.c</code> [firmware] every main loop cycle, thus giving the clock the opportunity to update itself. For example, it needs to handle changes in time, changes in the position of the alarm on/off switch and changes in the alarming/snoozing state of the clock.

Element	Description
<code>button()</code>	This is the optional published function for a clock. When published, it is invoked via <code>anim.c</code> [firmware] in a main loop cycle when a button is pressed.

Table 16: The clockDriver_t clock driver structure elements

2.13 Quick guide into adding a new clock plugin

Find below an overview of the files to be created/modified when adding a new clock in the Emuchron clock plugin framework.

File	Description
<code>anim.h</code> [firmware]	– Create a unique id for the clock <code>#define CHRON_MYCLOCK</code>
<code>anim.c</code> [firmware]	– Include the new clock header <code>#include "clock/myClock.h"</code> – When you want to test or upload your new clock in Monochron, add the clock id and public <code>init()</code> , <code>cycle()</code> and (optional) <code>button()</code> functions for <code>myClock</code> in static array <code>monochron[]</code> .
<code>help.txt</code> [firmware/emulator]	– Modify the help text for command 'cs' by adding the numeric id and description of the new clock. See also changes for <code>mchron.c</code> .
<code>Makefile</code> [firmware]	– When appropriate add the <code>myClock.c</code> file in variable <code>SRC</code> . This is needed for building Monochron firmware that includes the new clock.
<code>MakefileEmu</code> [firmware]	– Add the <code>myClock.c</code> file in variable <code>CSRC</code> . This is needed for building Emuchron and the <code>mchron</code> command line tool.
<code>mchron.c</code> [firmware/emulator]	– Include the new clock header <code>#include "../clock/myClock.h"</code> – Add the clock id and public <code>init()</code> , <code>cycle()</code> and (optional) <code>button()</code> functions for <code>myClock</code> in static array <code>emuMonochron[]</code> . – Verify if the clock needs special handling in <code>doAlarmSet()</code> .
<code>mchronutil.c</code> [firmware/emulator]	– Verify if the clock needs special handling in <code>emuClockUpdate()</code> .
<code>myClock.c</code> [firmware/clock]	– Create a new clock source file that implements the public and private functions for the clock.
<code>myClock.h</code> [firmware/clock]	– Create a new clock header file that publishes the public <code>init()</code> , <code>cycle()</code> and (optional) <code>button()</code> functions for the clock.

Table 17: What to create/modify when adding a new clock plugin

3 Setting up the software environment

3.1 Introduction

Emuchron is supported on Debian 6, 7 and 8. In order to be able to build and upload Monochron firmware, and to build the mchron emulator we need compilers and several Linux libraries. Next, in order to be able to use the ncurses LCD device we need to configure a terminal profile and create a shortcut to start a Gnome terminal with a specific command line.

3.2 Configuring Debian

3.2.1 General Debian requirements

In order to be able to use Emuchron configure Debian with GNOME or GNOME Classic. Apart from this, Emuchron does not require out of the ordinary CPU, memory or graphics card performance.

When running Debian in a VM it is highly recommended that in the BIOS of the host machine the CPU is enabled to use Intel (VT-x) or AMD (AMD-V) Virtualization Technology as this will significantly improve VM performance. On Intel Macs this is enabled by default.

Also, make sure that the VM accepts USB devices. In general, if you're able to see the contents of a plugged-in USB flash disk, the VM is able to successfully attach to the FTDI USB device as well.

3.2.2 Configuring a Debian VM in VirtualBox

As OpenGL2/GLUT performance benefits from the availability of basic hardware acceleration, enable the 3D acceleration tick box for the Debian VM. See below. If not ticked on, the GLUT LCD device will show less fluent video behavior.

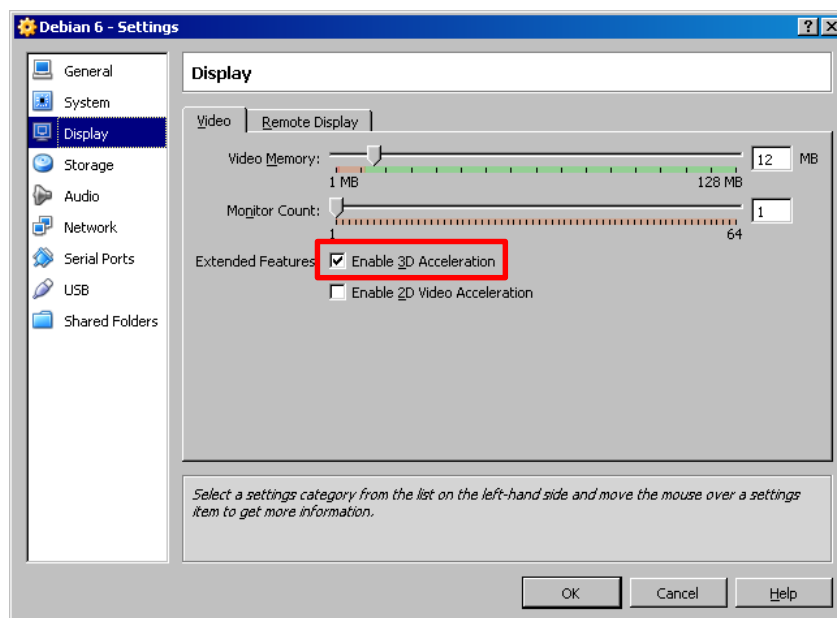


Figure 4: Enable 3D acceleration for a Debian VM in VirtualBox

Next, in the VM itself, after logging on, add the following line at the bottom of file `$HOME/.bashrc`.

```
export LIBGL_ALWAYS_SOFTWARE=1
```

Adding this line will prevent warnings, crashes and coredumps caused by OpenGL2 upon starting the mchcron tool with the OpenGL2/GLUT LCD device.

3.2.3 Configuring a Debian VM in VMware Fusion

As OpenGL2/GLUT performance benefits from the availability of basic hardware acceleration, enable the Acceleration 3D Graphics option for the Debian VM. See below. If not enabled, the GLUT LCD device will show less fluent video behavior.

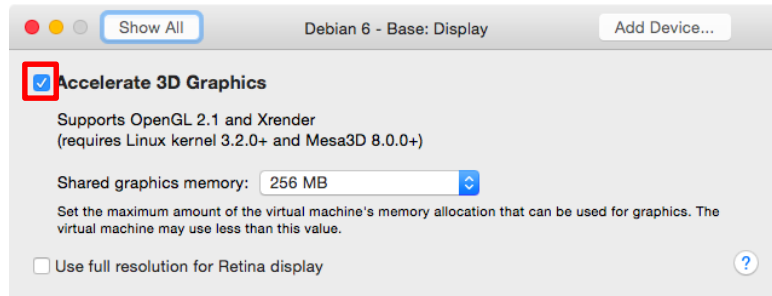


Figure 5: Enable 3D acceleration for a Debian VM in VMware Fusion

3.3 Unpacking the project software

The Emuchron project package can be downloaded via github location <https://github.com/tceulema/Emuchron> and can be unpacked in any location. Make sure that full read and write access is available on the project root and its structure below. The project root location is referenced in command shell examples as `<install_dir>`.

3.4 Installing required Linux packages

Setting up an AVR toolchain environment for Linux is described on <http://www.ladyada.net/learn/avr/setup-unix.html> and includes instructions to manually download and build several packages.

Fortunately, for Debian Linux there is no need to do all of this. Instead, all required packages can be retrieved and installed using `apt-get`. This also applies to installing the required libraries for the Emuchron environment, LCD and piezo stub devices, debugging tools and, for Debian 8, glibc source files.

In the Emuchron node a shell script is available to download and install all required packages.

For this start a command shell and execute the commands below.

```
$ # Only an admin user is allowed to install stuff
$ su - root
$ # When logged in as root first update the list of package sources
$ apt-get update
$ # Then execute the script to install required packages
$ cd <install_dir>/support
$ . ./packages.txt
```

Note: For the 'su' command you need to supply the root password to acquire administrator rights.

Note: During the installation of several packages you are asked to confirm installing dependency packages. As the default is 'Y', all that is needed is to press the enter key.

Note: Depending on the configuration of apt-get it is possible that the tool asks the end-user to insert the original installation media. If the installation media is not inserted, the installation of several packages will fail. To prevent apt-get using any installation media, the end-user can manually comment out the reference(s) to physical installation media in sources.list [/etc/apt]. This will require admin rights. When needed, rerun the packages script.

3.5 Copying configuration file for minicom

The minicom application is used for debugging the Monochron clock. It allows making a connection to Monochron via the FTDI port and, when proper firmware is uploaded to Monochron, to extract runtime debug text strings from the port. It is installed as part of the software installation procedure as described in section 3.4. The specifics for connecting minicom to Monochron using FTDI Friend v1.1 are saved in a configuration profile in [support] that needs to be copied to the minicom environment. For more information on how to use minicom refer to section 6.1.

To copy the Monochron profile for minicom execute the commands below.

```
$ # Only an admin user is allowed to install stuff
$ su - root
$ cd <install_dir>/support
$ cp minirc.Monochron /etc/minicom
```

Note: For the 'su' command you need to supply the root password to acquire administrator rights.

3.6 Setting up and using an ncurses Monochron terminal

Emuchron supports two LCD stub devices, being a GLUT device and an ncurses device. The GLUT device requires no setup. The ncurses device however does.

Ncurses is a terminal type of device. In order to be used for Emuchron it needs to reproduce square pixels with geometry 128x64.

GNOME allows creating so-called terminal profiles in which characteristics like font and font size, foreground and background colors and scrollbar behavior can be configured. By creating a dedicated profile for a Monochron ncurses terminal, a one-time only action, we can create a GNOME terminal that can be used as an ncurses Monochron LCD stub device.

3.6.1 Creating a Monochron terminal profile

The instructions for creating a Monochron terminal profile in Debian 6 and 7 is found in appendix C.1.

The instructions for creating a Monochron terminal profile in Debian 8 is found in appendix C.2.

3.6.2 Starting a Monochron ncurses terminal

Once a terminal profile is created we can start a Monochron ncurses terminal by executing the proper shell command.

A command shortcut named Monochron [support] or 'gnome-terminal.desktop' [support] is available that will do this. Copy this shortcut to the desktop for easy access.

Note: Although the actual name of the shortcut is 'Monochron' it is very well possible that it is named 'gnome-terminal.desktop' [support]. GNOME may see the shortcut as a potential security risk and as such will initially refuse to see it

as a legitimate file. Upon copying or double-clicking the shortcut you may be asked to confirm the validity of the shortcut. When confirmed, GNOME will rename the file to a shortcut named 'Monochron'.

When double-clicked, the Monochron shortcut will execute the following command:

```
gnome-terminal --window-with-profile=Monochron --hide-menubar --geometry=258x66  
-e "bash -c \"tty > ~/.mchron; bash\""
```

This command implements the following functionality:

1. Start a GNOME terminal.
2. The terminal will use terminal profile "Monochron", as configured according instructions in appendix C.
3. The terminal will hide its menubar.
4. The terminal geometry is 258x66 characters. This is quite big, but as the font size in the profile is set to 2, the terminal itself will have about the same size as a regular bash terminal.
5. Upon startup, a bash is started that will copy the tty info of the window in file \$HOME/.mchron. The mchron command line tool will then use this info to automatically link the ncurses LCD stub device output to that tty.
For more info on the mchron command line arguments refer to section 5.2.

When the Monochron terminal profile is properly setup, double-clicking the Monochron shortcut will create a blank black Monochron ncurses terminal. The terminal header as shown in the screendump below is not supported in Debian 8. Note the small command prompt at the top left of the window, caused by the very small font point size.

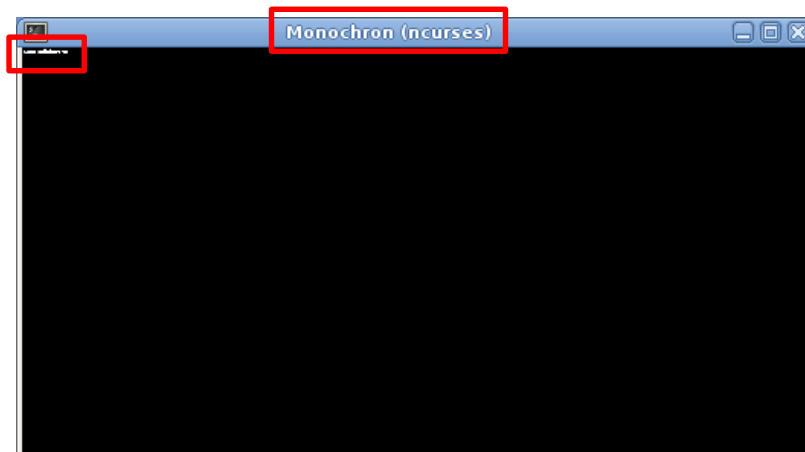


Figure 6: A blank Monochron terminal

In addition to that, a file .mchron will appear in the home folder, containing the tty of the Monochron terminal. See below.

```
$ cat ~/.mchron  
/dev/pts/1
```

3.6.3 Changing the size of a Monochron ncurses terminal

Once created, a Monochron ncurses terminal may not be increased or decreased in size in terms of the number of horizontal columns or vertical rows. This will confuse ncurses and will permanently disturb the layout of the window.

However, the window size can be increased or decreased by means of changing the character font size that is used within the terminal.

- To increase the font size in a Monochron terminal activate the window and type '<ctrl>+'.
- To decrease the font size in a Monochron terminal activate the window and type '<ctrl>-'.

Note that only a limited number of font sizes will reproduce square 'pixels'.

3.7 Debian 8 issues and regression in functionality

Debian 8, at the time of writing this document, suffers from a number of inconveniences as well as regression in functionality when compared to Debian 6 and 7. These issues combined make Debian 8 a less friendly environment to setup in general and for Emuchron in particular.

3.7.1 ALSA audio is getting ever less responsive

To illustrate the problem, in mchron, execute script beep.txt [script] in Debian 6, 7 and 8.

In Debian 6 the beeps are fluid and there is hardly any delay in between individual beeps. In Debian 7 the time between individual beeps is increased significantly, but ALSA remains stable as a means to generate audio. In Debian 8 however, in between beeps random underflow buffer errors occur, requiring a code change in stub.c [firmware/emulator] to redirect the audio `play` error stream to `/dev/null`. Similar buffer errors have occasionally been seen while playing alarm audio.

In general, the ALSA audio interface gets less and less responsive with each Debian release.

3.7.2 There may not be audio at all

In certain circumstances audio in Debian 8 may be totally missing, caused by an erroneous setup of audio devices during installation. This issue was seen after installing Debian 8 in both a VirtualBox and VMware Fusion VM.

In case audio is missing first try this:

1. Via the main menu or activities overview launch the Settings tool and next select Sound.
Verify that audio is unmuted. If muted, unmute and retry audio.

If audio is still missing then try the following:

1. In a command shell execute the following command: `alsamixer`
2. In `alsamixer` press the F6 key.
This will provide an overview of available audio devices.
3. Use cursor up/down to navigate to the non-default sound card (for example: Intel 82801AA-ICH) and press Enter to select.
This will display the playback settings for the non-default audio device.
4. The two most left bars identify the Master Mono and Master Surround controllers.
Use cursor left/right to navigate to these bars.
5. In each of these bars use cursor up to increase the volume to 100%.
6. Also, at the bottom of the bar a marker indicates the current mute status where 'MM' indicates mute and '00' indicates unmute.
Press the 'm' key to toggle the mute state. Unmute both masters.
7. After applying these changes, the result should look similar to graph below.
8. Use the esc key to exit and retry audio.

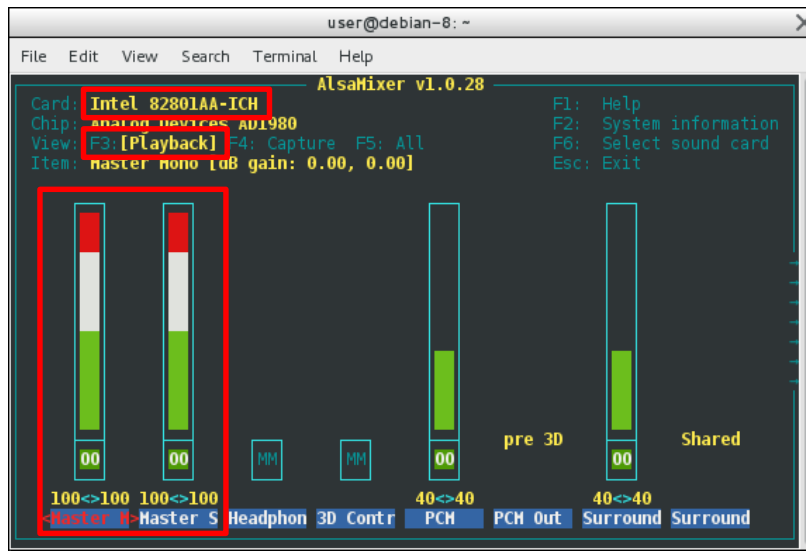


Figure 7: ALSA mixer with unmuted playback Master controllers

3.7.3 A terminal profile can no longer set a terminal header

In section 3.6.1 a Monochron terminal profile is created for use in a Monochron LCD ncurses stub device. In Debian 8 it is no longer possible to set the header of the terminal to "Monochron (ncurses)".

3.7.4 The gdb debugger cannot find file "syscall-template.S"

In Debian 8, using DDD or Nemiver as a graphical front-end for gdb is suffering from multiple popups at every debugger breakpoint indicating that file "syscall-template.S" cannot be found.

The issue originates from within the gdb debugger that forwards the problem to the graphical front-end. In short, as of Debian 8, gdb/DDD/Nemiver like to have glibc sources available.

This in itself is not really an issue as long as we're able to install the glibc sources using apt-get, and the packages.txt script [support] will do this specifically for Debian 8 systems.

The main problem however is that the location where gdb expects to find the sources is not static and may differ per Debian 8 installation. Note that only Nemiver will provide the full path of the file not being found. Refer below for an example.

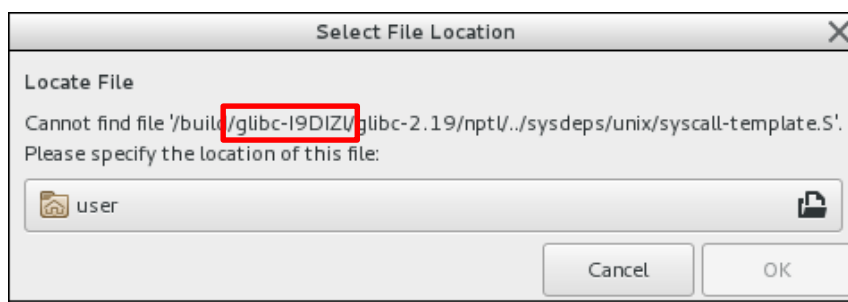


Figure 8: The gdb debugger cannot find file syscall-template.S

The part in the path that may vary per Debian 8 system is highlighted, in this case "glibc-19DIZI".

The problem can be resolved by creating a symbolic link that points to the actual installation folder of the glibc sources. The packages.txt [script] that

installs the glibc sources already creates two of these symbolic links by default, being "glibc-Ir_s5K" and "glibc-I9DIZI" that are commonly used.

In case another symbolic link needs to be created, depending on the information shown in the Nemiver popup, refer to the instructions below that create a symbolic link for imaginary folder "glibc-ABC".

```
$ # Only an admin user is allowed to install stuff
$ # If needed, remove a symbolic link using: rm glibc-ABC
$ su - root
$ cd /build
$ ln -s glibc glibc-ABC
```

Note: For the 'su' command you need to supply the root password to acquire administrator rights.

When a proper symbolic link has been created, gdb and its graphical frontends will no longer complain about not being able to access the source file.

4 Building firmware and the emulator

4.1 Building Monochron firmware

The `make` command builds Monochron firmware. For Monochron firmware it is driven by the default file named `Makefile [firmware]`.

The Monochron firmware build needs to be configured:

- `Makefile [firmware]`
Verify that variable `SRC`, next to the common modules in `[firmware]`, contains the proper list of clock plugin modules. Also select which alarm melody to use as described in section 2.8.
- `anim.c [firmware]`
Verify that static array `monochron[]` contains the correct set of clocks, limited by the `Makefile SRC` variable.

When configured enter the following commands:

```
$ cd <install_dir>/firmware
$ make <all | clean | rebuild>
```

Details for the Monochron firmware make command:

- `make all`
Build all modules that require a (re)build and generate the Monochron firmware in file `monochron.hex [firmware]`.
- `make clean`
Clean all object and dependency files.
- `make rebuild`
A combination of 'clean' followed by 'all'.

When the build has successfully completed an overview is provided of the firmware memory map. See below for an example.

```
Size after:
monochron.elf
section      size      addr
.data        1230      8388864
.text        29014      0
.bss         297       8390094
.stab        64380      0
.stabstr     16069      0
Total       110990
```

The Monochron Atmel CPU contains 32KB flash memory, of which 30KB is available for Monochron firmware. Verify that the sum of `.data` and `.text` does not exceed 30720 bytes (=30KB). If it does you need to optimize code, save space by using the two-tone alarm instead of the Mario alarm, make sure the debug output flag is switched off (refer to section 6.1.1), or remove one or more clocks from the `monochron[]` array and the `Makefile [firmware] SRC` variable.

Note: There is a substantial difference between `avr-gcc 4.3.5` (Debian 6) and `4.7.2/4.8.2` (Debian 7/8) with respect to the size of generated object code. Upon nearing the 30Kb limit of available flash storage for a hex file, `avr-gcc` version `4.7.2/4.8.2` generates a hex file that is ~500 bytes smaller than a hex file based on identical code that is built with version `4.3.5`. Unfortunately, smaller code does not imply faster execution. Code generated by `avr-gcc 4.7.2`

tends to run consistently slower than code generated by `avr-gcc 4.3.5`. Refer to appendix B.1 for detailed information on compiler object size and performance tests.

Note: When the previous build was for Emuchron, use `'make clean'` first or use `'make rebuild'` to clean up the build environment. The reason for this is that Emuchron x86 object code is incompatible with Monochron AVR Atmel object code, resulting in link failures.

Note: The Monochron firmware and clock plugin code as downloaded from github will build warning free.

4.2 Building Emuchron and mchron command line tool

Emuchron and its mchron command line tool will use its dedicated make file, being `MakefileEmu [firmware]`.

The Emuchron build needs to be configured:

- `MakefileEmu [firmware]`
Select which alarm tune to use as described in section 2.8.

In Monochron code the build switch `EMULIN` is used to build dedicated Emuchron stubs. This build switch is by default enabled.

Building Emuchron and mchron is done using the `make` command below.

```
$ cd <install_dir>/firmware
$ make -f MakefileEmu <all | clean | rebuild>
```

Details for Emuchron and the mchron command line tool make command:

- `make -f MakefileEmu all`
Build all modules that require a (re)build and build the mchron tool.
- `make -f MakefileEmu clean`
Clean all object and dependency files.
- `make -f MakefileEmu rebuild`
A combination of `'clean'` followed by `'all'`.

Note: When the previous build was for Monochron firmware, use `'make -f MakefileEmu clean'` first or use `'make -f MakefileEmu rebuild'` to clean up the build environment. The reason for this is that Monochron AVR Atmel object code is incompatible with Emuchron x86 object code, resulting in link failures.

Note: The Emuchron, emulator and clock plugin code as downloaded from github will build warning free.

4.3 Uploading Monochron firmware to Monochron clock

Use the `avrdude` command to upload Monochron firmware to the Monochron clock. Installing `avrdude` is described in section 3.4.

More information on configuring and using `avrdude` is found on:

<http://www.ladyada.net/learn/avr/setup-unix.html>

<http://www.ladyada.net/learn/avr/avrdude.html>

Specific information on updating Monochron firmware is found on:

<https://learn.adafruit.com/monochron/updating>.

Please note the following regarding the use of `avrdude` on Linux and Linux VM's, in combination with FTDI Friend v1.1 (<https://learn.adafruit.com/ftdi-friend>).

- When using a Debian VM, make sure that the VM is setup to support USB devices. If not, the USB FTDI device will not be recognized.
- Plugin the FTDI device in Monochron with the chip and USB port facing down, and the settings jumpers facing up. When plugged in and seen from above you'll notice the settings jumpers on the FTDI circuit board.
- The USB FTDI device will appear as logical device `/dev/ttyUSBx`. In normal circumstances the USB FTDI device will be the only USB terminal device that is connected to your machine. If so, it will map to logical device `/dev/ttyUSB0`.
- To prevent confusion on which hardware USB device is which logical `/dev/ttyUSBx` device, unplug all other USB devices. If you do need other USB devices as well you need to verify which logical `/dev/ttyUSBx` device will be assigned to the USB FTDI device.
- When using Debian Linux as a VM, after plugging in the USB FTDI device you need to attach it to your VM. The device to attach to will show up with a name similar to 'FTDI FT232R USB UART'. Note that both VirtualBox and VMware Fusion have succeeded in using `avrdude` on the USB FTDI device to upload firmware to Monochron.
- Getting the USB FTDI device to attach to your machine or VM may take some time, especially the first time as Linux may need to do configuration tasks. If you have no other USB devices plugged in, wait for device `/dev/ttyUSB0` to pop up in `/dev`.
In one case when the USB FTDI device was plugged in for the very first time, it did not get fully recognized at first. In case this occurs, by un/replugging or rebooting Linux the device eventually becomes visible for `avrdude`. Be patient and give Linux time to get its act together.
- By default the USB device can be accessed by root only, meaning that only the root user is allowed to use `avrdude` on the FTDI device. By using the appropriate `chmod` command you can open up this device to other user groups as well. The examples below however will use the root user to upload the firmware.

Find below the Linux commands needed to upload firmware to Monochron. A text copy, including those for Windows, is available in `avrdude.txt` [support].

```
$ # You must have admin rights or you'll be denied access to /dev/ttyUSBx
$ su - root
:
$ # You must be in the same folder where monochron.hex firmware resides
$ cd <install_dir>/firmware
:
$ # First verify whether avrdude can talk to the Monochron clock
$ # Device /dev/ttyUSB0 may differ depending on which USB devices are attached
$ # For parameter -p use either "m328p" or "atmega328p"
$ avrdude -c arduino -p m328p -P /dev/ttyUSB0 -b 57600
:
$ # Then upload firmware to the Monochron clock
$ # Device /dev/ttyUSB0 may differ depending on which USB devices are attached
$ # For parameter -p use either "m328p" or "atmega328p"
$ avrdude -c arduino -p m328p -P /dev/ttyUSB0 -b 57600 -U flash:w:monochron.hex
```

If an attempt is made to upload firmware that is larger than 30KB, a firmware verification error is reported at the 30KB memory address. See the example below. Don't be surprised when your clock will hang soon after it has been started.

```
:  
avrdude: verifying  
avrdude: verification error, first mismatch at byte 0x7800  
0x00 != 0x0c  
avrdude: verification error; content mismatch  
  
avrdude: safemode: Fuses OK  
  
avrdude done. Thank you.
```

5 The mchron command line tool

5.1 Introduction

Emuchron is controlled via its command line tool mchron. It provides commands to access clock plugins at will, feed clocks with a continuous stream of time and keyboard events, change the time/date/alarm, access the graphics library to draw on the stubbed LCD display, and run a stubbed Monochron application ahead of building and uploading actual firmware.

5.2 Starting mchron

Find below an excerpt from the help file as found in help.txt [support].

```
mchron - Emuchron emulator command line tool

Use: mchron [-l <device>] [-t <tty>] [-g <geometry>] [-p <position>]
      [-d <logfile>] [-h]
  -d <logfile>  - Debug logfile name
  -g <geometry> - Geometry (x,y) of glut window
                  Default: "520x264"
                  Examples: "130x66" or "260x132"
  -h           - Give usage help
  -l <device>   - Lcd stub device type
                  Values: "glut" or "ncurses" or "all"
                  Default: "glut"
  -p <position> - Position (x,y) of glut window
                  Default: "100,100"
  -t <tty>      - tty device for ncurses of 258x66 sized terminal
                  Default: get <tty> from $HOME/.mchron

Examples:
  ./mchron
  ./mchron -l glut -p "768,128"
  ./mchron -l ncurses
  ./mchron -l ncurses -t /dev/pts/1 -d debug.log
```

When Emuchron is successfully built, the mchron command line tool can be started.

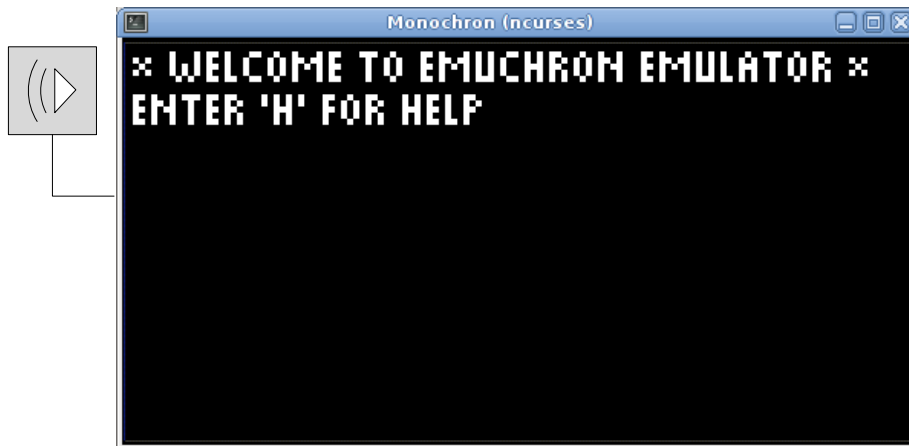
When using the ncurses LCD stub device, first read and execute all the necessary steps in sections 3.6.1 and 3.6.2 on how to setup and start a Monochron ncurses terminal.

```
$ # When using the (default) OpenGL2/GLUT LCD stub device
$ # Note: No additional configuration is needed
$ cd <install_dir>/firmware
$ ./mchron

$ # When using the ncurses LCD stub device
$ # Note: Refer to 3.6.1 and 3.6.2 to setup and start an ncurses terminal
$ cd <install_dir>/firmware
$ ./mchron -l ncurses

$ # When using both the OpenGL2/GLUT and ncurses LCD stub devices
$ # Note: Refer to 3.6.1 and 3.6.2 to setup and start an ncurses terminal
$ cd <install_dir>/firmware
$ ./mchron -l all
```

Starting mchron should result in an audible startup beep and the following screen layout in the LCD stub device(s).



The mchron command terminal will show tool and runtime information and provides a command entry prompt. See below.

```
$ ./mchron -l ncurses

*** Welcome to Emuchron command line tool (build Oct 22 2013, 13:32:00) ***

mchron PID = 3382
ncurses tty = /dev/pts/1

time  : 13:38:35 (hh:mm:ss)
date   : 22/10/2013 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off

Enter 'h' for help.
mchron>
```

Note: In the unfortunate event that mchron crashes or properly fails to initialize at startup refer to section 7.6.

5.3 Interrupting and stopping mchron

Within mchron there are several ways to interrupt command execution and to stop it. Also, mchron has a built-in mechanism to protect itself against invalid LCD operations requested by end-user commands, incorrect clock code or incorrect graphics code.

Note: Regardless the event causing an intended or unintended shutdown, mchron will always try to shutdown gracefully. A graceful shutdown however cannot be guaranteed at all times and may cause the mchron terminal to stop echoing keyboard input. Refer to section 8.1 for its symptoms and a simple method to resolve this.

The following options are available to interrupt and stop mchron:

- Interrupt command execution by using keypress 'q'.
The execution of a command file or multi-command list (refer to section 5.10) or a wait command is interrupted by using a 'q' keypress. When appropriate a stack trace of nested load commands is reported for informational purposes. Internally, the interpreter will properly clean-up the entire stack after which the mchron prompt will re-appear. For a stack trace example refer to section 5.5.
- Stop mchron at any moment using '<ctrl>c'.
This keypress will generate a `SIGINT` signal.

- Stop mchcron at command prompt level using command 'x' or '<ctrl>d' on an empty line.

Example:

```
mchcron> # Press '<ctrl>d' on an empty line to exit mchcron
mchcron>
<ctrl>d - exit
$
```

Press '<ctrl>d'

- Quit mchcron multi-line command mode using '<ctrl>d' on an empty line.

Example:

```
mchcron> # Press '<ctrl>d' on an empty line to quit multi-line command mode
mchcron> rf x=0 x<128 x=x+1
2>> pd f x y
3>>
<ctrl>d - quit
mchcron>
```

Press '<ctrl>d'

- Stop mchcron at any moment using '<ctrl>z'.
- This keypress will generate a SIGTSTP signal. The effect of this method is similar to using keypress '<ctrl>c'.
- Force a coredump at any moment using '<ctrl>\'.
- This keypress will generate a SIGQUIT signal that on its turn will generate a SIGABRT signal that will cause mchcron to coredump.

5.4 Pre-emptive coredump of mchcron

Mchcron coredumps itself when it detects an invalid LCD operation.

It is very well possible to enter an mchcron command that attempts to draw pixels outside the boundaries of the LCD area. Also, it is very well possible that, due to a bug, clock code attempts to do the same.

Whenever an attempt is made to read or write pixels outside the boundaries of the LCD display, mchcron will actively force itself to coredump since this is an unacceptable situation that needs to be resolved.

In case only the OpenGL2/GLUT LCD device is used, the user is presented a confirmation prompt prior to the actual coredump, allowing to inspect the display as-is and/or create a screendump before it is closed down. When using the ncurses LCD device, the current display is retained after the coredump.

Note: In case mchcron will coredump, an actual coredump file will be created in [firmware] only when in the command shell the following command is executed once prior to starting mchcron: `ulimit -c unlimited`

```
$ # Make sure a coredump file will be created in this shell upon coredumping
$ ulimit -c unlimited
$ # Start mchcron
$ ./mchcron -l ncurses
:
Enter 'h' for help.
mchcron> # Let's try to paint outside the LCD display boundaries :-0
mchcron> pr f 120 60 50 50

*** invalid lcd api request in glcdSetAddress()
api info (controller:x:y:data) = (0:128:7:0)
*** registered variables
variables in use: 0
*** debug by loading coredump file (when created) in a debugger
Aborted (core dumped)
$ ls -l core
-rw----- 1 user user 25608192 Oct 28 18:35 core
$
```

5.5 The mchcron stack trace

When executing commands from a command file or multi-command input, mchcron provides a stack trace for informational purposes whenever it is interrupted or encounters an error.

A stack trace line consists of 4 items separated by a colon. For an example, see below.

```
mchcron> # Demo execution interrupt using 'q' keypress on wait
mchcron> e s ../script/demo.txt
<wait: q = quit, other key will continue>
quit
--- stack trace ---
2:../script/paint.txt:17:w 0
1:../script/demo.txt:12:e i ../script/paint.txt
0:mchcron--:e s ../script/demo.txt
mchcron>
```

Press 'q'

File depth File name or 'mchcron' Line number in file Command

5.6 Recovering from command syntax and parse errors

Whenever mchcron detects a syntax or parse error in a command it will abort its execution. Information will be provided on the circumstances causing the command to abort. A command stack trace will be provided when appropriate. For an example of a stack trace refer to section 5.5.

Refer to the example below.

```
mchcron> # The paint dot x position argument is beyond the LCD display boundaries
mchcron> pd f 153 30
x? invalid: 153
mchcron>
```

5.7 The mchcron command line history log

In case only the OpenGL2/GLUT LCD stub device is used, mchcron will use the readline library for command line input and caching and flushing the mchcron command line history in a file. The readline library functionality is not active when the ncurses LCD device is used, caused by interoperability issues between the readline and ncurses libraries. For this refer to section 7.7.

The default command history log file is \$HOME/.mchcron_history that is created by mchcron when not present. Reset the history by stopping mchcron and then removing the file. Its configuration is found in scanutil.c [firmware/emulator]. See below.

```
// The readline unsaved cache and history file with size parameters
#define READLINE_CACHESIZE 15
#define READLINE_HISFILE  "/.mchcron_history"
#define READLINE_MAXHISTORY 250
```

Some examples of functionality provided by the readline library:

Browsing the command log is done using the up/down arrows. Navigating in a command line is done using the left/right arrows, or '<ctrl-a>' and '<ctrl-e>' for respectively the beginning and end of a command line. A reverse-order search in the log is started using '<ctrl-r>'.

5.8 The mchron command groups

The structure of an mchron command is simple.

```
<command> <arg1> <arg2> .. <argn>
```

Note the following:

- A command is always a single text word. An argument can be a single character, a text word, a text string (many words) or a numeric expression.
- An mchron command line contains a single command only.
- Command and arguments are separated by white space (space or tab). The only exception is an argument of type text string that consists of all remaining text on a command line.
- As arguments are not named, it will have a negative impact on the readability. Consider this a learning curve. The purpose of mchron is to provide a command line interface with a simple syntax structure.
- Mchron supports named numeric variables that are identified by a word of mixed upper/lowercase characters in the range 'a'..'z'.
- Numeric type arguments are read as a text word that is fed through an expression evaluator. In combination with named variables it provides great flexibility in passing calculated numeric values to mchron command arguments.
Note that command handlers that implement numeric arguments are responsible for casting expression evaluator results, being of type `double`, into an integer type. Casting macros are provided for this purpose in `mchron.c` [firmware/emulator]. These macros also take care of value rounding to the nearest integer number.
- An mchron command line is not limited in length.

An example of several commands can be seen on the front page of this document. On the top of the front page a script is listed that results in the Monochron screendump at the bottom.

Below is an overview of all main command groups. A command group consists of one or more individual commands. Many examples of commands are found in script files in [script]. The command description text boxes contain an excerpt from `help.txt` [support].

5.8.1 '#' – Comments

The comment command serves no other purpose than to provide information to the end-user.

```
Command:  
'#' - Comments
```

Usage specifics:

- The comments command and the actual comments must be separated by a white space character.
- Comments are optional.
- When a comment command is entered on the mchron command line in combination with debug logging being active, the comments are added in the debug log to serve as a debug log marker.

Example:

```
mchron> # This is a comment  
mchron> #  
mchron> # An empty comment in the line above is also allowed  
mchron>
```

5.8.2 'a' – Alarm

The alarm commands allow setting the alarm time and the alarm switch position. Related command groups are date ('d') and time ('t').

```
Commands:
'ap' - Set alarm switch position
      Argument: <position>
            position: 0 = off, 1 = on
'as' - Set alarm time
      Arguments: <hour> <min>
            hour: 0..23
            min: 0..59
'at' - Toggle alarm switch position
```

Usage specifics:

- When an alarm command is used, an active clock is called to update itself using the modified settings.

Example:

```
mchron> # Set alarm time to 14:51
mchron> as 14 51
time  : 17:03:34 (hh:mm:ss)
date   : 28/10/2013 (dd/mm/yyyy)
alarm  : 14:51 (hh:mm)
alarm  : off
mchron> # Set alarm switch to 'on'
mchron> ap 1
time  : 17:03:50 (hh:mm:ss)
date   : 28/10/2013 (dd/mm/yyyy)
alarm  : 14:51 (hh:mm)
alarm  : on
mchron> # Toggle alarm switch
mchron> at
time  : 17:03:55 (hh:mm:ss)
date   : 28/10/2013 (dd/mm/yyyy)
alarm  : 14:51 (hh:mm)
alarm  : off
mchron>
```

5.8.3 'b' – Beep

The beep command plays a beep with a specific frequency and duration.

```
Command:
  'b'    - Play audible beep
          Arguments: <frequency> <duration>
                  frequency: 150..10000 (Hz)
                  duration: 1..255 (msec)
```

Usage specifics:

- The stubbed piezo interface spawns a Linux play process for each individual beep, making it relatively slow. When playing multiple beeps in a script file, you will hear a pause between each beep. The duration of this pause will grow progressively worse between Debian 6 versus Debian 7 and 8.
- The quality of the actual piezo speaker is worse than miserable. It has a very narrow frequency range in which tones are played with a decent volume without audible distortion. So, tones that are played in mchron are likely to sound near-horrible when played by the actual piezo speaker.

Example:

```
mchron> # Play a 4000 Hz tone lasting 150 msec
mchron> b 4000 150
mchron>
```

5.8.4 'c' – Clock

The clock commands allow selecting a clock in the Emuchron test environment and feeding it with a continuous stream of time and keyboard events.

```
Commands:
  'cf' - Feed clock with time and keyboard events
        Argument: <mode>
                mode: 'c' = start in single cycle mode, 'n' = start normal
  'cs' - Select clock
        Argument: <clock>
                clock: 0 = [detach], 1 = analogHMS, 2 = analogHM, 3 = digitalHMS,
                       4 = digitalHM, 5 = mosquito, 6 = nerd, 7 = pong,
                       8 = puzzle, 9 = slider, 10 = cascade, 11 = speed,
                       12 = spider, 13 = traffic, 14 = bigdigOne, 15 = bigdigTwo,
                       16 = qrHMS, 17 = qrHM, 18 = perftest
```

Usage specifics:

- For the clock commands, mchron uses the clocks defined in the `emuMonochron[]` array in `mchron.c` [firmware/emulator].
- In case no clock is selected (clock 0), changing the mchron date/time/alarm will still work, but these changes will not be reflected in the LCD display as there is no clock to update.
- When selecting a clock, the time displayed in the clock will most likely not be the actual mchron time. Effectively it will be the timestamp from the last executed time command or the last known timestamp in the 'cf' and 'm' emulator commands. This is per design and allows the user to switch between clocks while displaying the same time for comparison purposes. Flushing the current mchron time to a selected clock is done using the 'tf' command.
- When the alarm is audible and the clock is moved into the single application cycle mode using keypress 'c', audible alarm is temporarily stopped. Audible alarm resumes upon switching back to normal mode.
- Audible alarm can be stopped by using keypress 'a' to toggle the alarm switch position, or by keypress 'q' to quit the clock emulator.
- Clock 18, perftest, is a special clock plugin used for running high-level glcd performance tests. For this, refer to section 2.9.

Example:

```
mchron> # Select the analog HMS clock
mchron> cs 1
mchron> # Start this clock in a testbed environment
mchron> cf n
emuchron clock emulator:
  c = execute single application cycle
  h = provide emulator help
  p = print performance statistics
  q = quit
  r = reset performance statistics
  t = print time/date/alarm
hardware stub keys:
  a = toggle alarm switch
  s = set button
  + = + button
```

Clock emulator specifics:

- Keypress 'a' is identical to command 'at'.
- Keypress 'p' is identical to command 'sp'.
- Keypress 'r' is identical to command 'sr'.
- Keypress 't' is identical to command 'tp'.

5.8.5 'd' – Date

The date commands allow setting a dedicated date or reset the date to the current system date. Related command groups are alarm ('a') and time ('t').

```
Commands:
'dr' - Reset clock date to system date
'ds' - Set clock date
      Arguments: <day> <month> <year>
              day: 1..31
              month: 1..12
              year: 0..99
```

Usage specifics:

- When a date command is used, an active clock is called to update itself using the modified settings.
- The year is placed in 20xx.
- When setting a date, an offset is calculated between the system date and the requested date. Daylight savings settings are taken into account to compensate for time offsets between the old and new date. The calculated offset will be used as a delta between the system date and the mchron date.
- To determine the delta between the current and requested date the `mktime()` system call is used. This system call allows specifying a date into the future up to approx. 25 years. When the requested date is beyond that range an error message is provided.
- The 'ds' command verifies whether the requested date is valid. For example, date April 31 will be rejected.

Example:

```
mchron> # Set our own date to Jan 27 2015
mchron> ds 27 1 15
time   : 17:08:10 (hh:mm:ss)
date   : 27/01/2015 (dd/mm/yyyy)
alarm  : 14:51 (hh:mm)
alarm  : off
mchron> # Reset to system date
mchron> dr
time   : 17:08:26 (hh:mm:ss)
date   : 28/10/2013 (dd/mm/yyyy)
alarm  : 14:51 (hh:mm)
alarm  : off
mchron> # September 31 does not exist
mchron> ds 31 9 15
date? invalid
mchron> # Year 2065 is too far ahead into the future
mchron> ds 1 8 65
date? beyond system range
mchron>
```

5.8.6 'e' – Execute

The execute command loads the content of a plain text file and executes it as mchron commands. Refer to section 5.10 where is described how this is internally accomplished.

```
Command:
  'e' - Execute commands from file
      Arguments: <echo> <filename>
               echo: 'e' = echo commands, 'i' = inherit, 's' = silent
               filename: full filepath or relative to startup folder mchron
```

Usage specifics:

- Upon loading the file contents, each line is checked whether it contains a valid command, being part of linking it to a command dictionary entry. However, at loading time no check is made whether any command arguments are complete and valid. Command argument validation is performed at command list execution time.
- The depth level of nested command files is supported up to the value of `#define CMD_FILE_DEPTH_MAX` in `scanutil.h` [firmware/emulator].
- The echo argument value 'e' indicates that all commands, accompanied by file line number, are echoed in the mchron command shell. Especially in combination with repeat command 'rf' this may generate lots of output.
- The echo argument value 's' indicates that no command echoing will occur. Normally this is the value to use upon typing in the 'e' command on mchron command prompt level.
- The echo argument value 'i' is used in case of a nested command file. Using this setting the echo value that is used in the current command depth level (either 'e' or 's') is forwarded to the next level.
- The execution of a command file can be interrupted at any depth level by using a 'q' keypress immediately or via a 'q' keypress in a wait command.

Example:

```
mchron> # Run script to test all 1440 minutes of a day in about 30 seconds
mchron> # for an analog clock
mchron> cs 2
mchron> vs s=1
mchron> vs w=20
mchron> e s ../script/time-hm.txt
(wait ~30 seconds for the script to finish)
mchron>
```

5.8.7 'h' – Help

The help commands provide generic help on mchron, its command dictionary and expression evaluator.

```
Commands:
'h'      - Help
'hc'     - Help command
           Argument: <command>
           command: mchron command or '*' for all commands
'he'     - Help expression
           Argument: <number>
           number: expression
```

Usage specifics:

- The help commands 'h' and 'hc' can only be used at mchron command prompt level.
- The 'h' command displays the included help.txt [support] file using the Linux `more` command.
- The 'hc' command reports mchron command and command argument information based on the command dictionary as built in `mchroundict.h` [firmware/emulator].
- The 'he' command passes the expression argument to the expression evaluator and prints its result, making it a kind of built-in calculator.

Example:

```
mchron> # Print command dictionary info for command 'pd' (paint dot)
mchron> hc pd
command: pd (paint dot)
usage  : pd <color> <x> <y>
         color: 'b','f' (b = background, f = foreground)
         x: 0..127
         y: 0..63
handler: doPaintDot()
mchron> # Print the result of an expression
mchron> he sin(pi/6)
0.500000
mchron>
```

5.8.8 'i' – If

The if-then-else commands provide branching capabilities in mchron command blocks.

```
Commands:
  'iei' - If else if then
          Arguments: <condition>
                   condition: expression determining block execution
  'iel' - If else
  'ien' - If end
  'iif' - If then
          Arguments: <condition>
                   condition: expression determining block execution
```

Usage specifics:

- An if-then-else construct start with an 'iif'(if-then) command, followed by optionally one or more 'iei' (else-if) commands, followed by an optional 'iel' (else) command and always closes with an 'ien' (if-end) command.
- When used in a command file, each if command must be matched with an if-end command in the very same file.
- If-then-else commands can be nested without any limitation.
- When an 'iif' command is entered at the mchron command prompt, the interpreter will enter a multi-line mode that is completed when an 'ien' command is entered that matches the 'iif' that invoked the multi-line command buildup.
To abort the entry of a multi-line mode 'iif' command, type '<ctrl>d' on an empty line. For an example of this refer to section 5.3.

Example:

```
mchron> # If-then-else logic that results in value 20 for variable y
mchron> vs x=2
mchron> iif x==1
2>> vs y=10
3>> iei x==2
4>> vs y=20
5>> iel
6>> vs y=30
7>> ien
mchron> vp y
y=20
mchron>
```


5.8.9 'I' – LCD

The LCD commands allow setting the LCD backlight and erase or inverse its contents.

```
Commands:
  'lbs' - Set lcd backlight brightness (glut support only)
          Argument: <backlight>
              backlight: 0..16
  'le'  - Erase lcd display
  'li'  - Inverse lcd display
```

Usage specifics:

- Only the GLUT LCD device can process changes in the backlight brightness. The ncurses LCD device will ignore the 'lbs' command.
- The 'li' command will, next to inverting the contents of the LCD display, also swap the LCD foreground and background colors. As a result, clocks and graphics functions, when implemented correctly, will automatically swap their painting behavior.

Example:

```
mchron> # Set LCD backlight brightness to a low setting (glut LCD device only)
mchron> lbs 3
mchron> # Inverse LCD display
mchron> li
mchron> # Inverse back to original view
mchron> li
mchron> # Erase LCD display contents
mchron> le
mchron>
```

5.8.10 'm' – Monochron

The Monochron command will start an emulated Monochron application.

```
Command:
'm'      - Start Monochron emulator
          Arguments: <mode> <eeprom>
                mode: 'c' = start in single cycle mode, 'n' = start normal
                eeprom: 'k' = keep, 'r' = reset
```

Usage specifics:

- The Monochron eeprom settings are initialized at startup of mchcron and are changed when using the stubbed Monochron application.
- When the 'm' command is used more than once, value 'k' for eeprom will keep the stubbed eeprom settings as they were when the previous stubbed Monochron application session was stopped.
Note: The behavior of value 'k' for eeprom is similar to unplugging and replugging the Monochron power adapter.
- When using value 'r' for eeprom it will reset the eeprom contents back to its default values.

Example:

```
mchcron> # Start the emulated Monochron application
mchcron> m n k
emuchron monochron emulator:
  c = execute single application cycle
  h = provide emulator help
  p = print performance statistics
  q = quit (valid only when clock is displayed)
  r = reset performance statistics
  t = print time/date/alarm
hardware stub keys:
  a = toggle alarm switch
  m = menu button
  s = set button
  + = + button
```

Monochron emulator specifics:

- Keypress 'a' is identical to command 'at'
- Keypress 'p' is identical to command 'sp'.
- Keypress 'r' is identical to command 'sr'.
- Keypress 't' is identical to command 'tp'.

5.8.11 'p' – Paint

The paint commands provide access to high-level glcd graphics functions.

```

Commands:
'pa' - Paint ascii
      Arguments: <color> <x> <y> <font> <orientation> <xscale> <yscale>
                <text>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                font: '5x5p' = 5x5 proportional, '5x7n' = 5x7 non-proportional
                orientation: 'b' = bottom-up vertical, 'h' = horizontal,
                't' = top-down vertical
                xscale: 1..64
                yscale: 1..32
                text: ascii text
'pc' - Paint circle
      Arguments: <color> <x> <y> <radius> <pattern>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                radius: 1..31
                pattern: 0 = full line, 1 = half (even), 2 = half (uneven),
                3 = 3rd line
'pcf' - Paint circle with fill pattern
      Arguments: <color> <x> <y> <radius> <pattern>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                radius: 1..31
                pattern: 0 = full, 1 = half, 2 = 3rd up, 3 = 3rd down
                4 = <unsupported>, 5 = clear
'pd' - Paint dot
      Arguments: <color> <x> <y>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
'pl' - Paint line
      Arguments: <color> <xstart> <ystart> <xend> <yend>
                color: 'f' = foreground, 'b' = background
                xstart: 0..127
                ystart: 0..63
                xend: 0..127
                yend: 0..63
'pn' - Paint number
      Arguments: <color> <x> <y> <font> <orientation> <xscale> <yscale>
                <number> <format>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                font: '5x5p' = 5x5 proportional, '5x7n' = 5x7 non-proportional
                orientation: 'b' = bottom-up vertical, 'h' = horizontal,
                't' = top-down vertical
                xscale: 1..64
                yscale: 1..32
                number: expression
                format: 'c'-style format string containing '%f', '%e' or '%g'
'pr' - Paint rectangle
      Arguments: <color> <x> <y> <xsize> <ysize>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                xsize: 1..128
                ysize: 1..64
'prf' - Paint rectangle with fill pattern
      Arguments: <color> <x> <y> <xsize> <ysize> <align> <pattern>
                color: 'f' = foreground, 'b' = background
                x: 0..127
                y: 0..63
                xsize: 1..128
                ysize: 1..64
                align (for pattern 1-3): 0 = top, 1 = bottom, 2 = auto
                pattern: 0 = full, 1 = half, 2 = 3rd up, 3 = 3rd down
                4 = inverse, 5 = clear

```

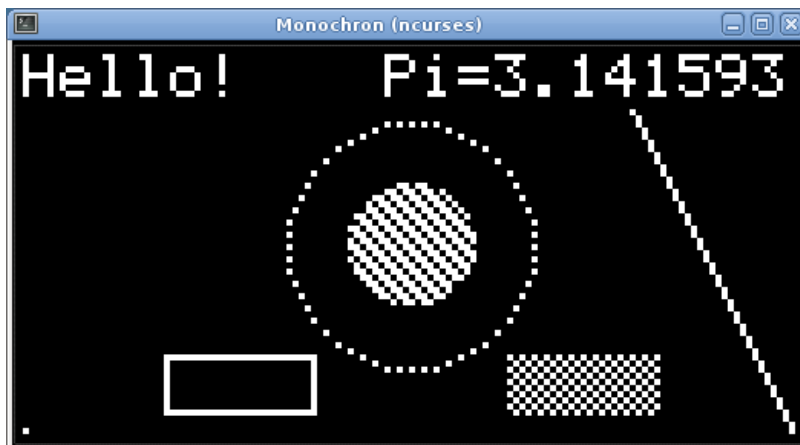
Usage specifics:

- Many script examples are available in [script] that use paint commands. See also the script on the front page of this document.
- The 'pn' command does not have an equivalent glcd function but is meant to provide a simple mechanism to print numbers in an mchron LCD stub device. For using the 'C'-style '%f', '%e' or '%g' formatting options refer to the many resources on the web. Examples are also found in paintnum.txt [script].

Example:

```
mchron> # Paint ascii
mchron> pa f 1 1 5x7n h 1 1 Hello!
hor px=36
mchron> # Paint dotted circle
mchron> pc f 64 32 20 1
mchron> # Paint filled circle
mchron> pcf b 64 32 10 3
mchron> # Paint dot at bottom left
mchron> pd f 1 62
mchron> # Paint line
mchron> pl f 100 10 126 62
mchron> # Paint number
mchron> pn f 60 1 5x7n h 1 1 pi Pi=%f
mchron> # Paint rectangle
mchron> pr f 24 50 25 10
mchron> # Paint filled rectangle
mchron> prf f 80 50 25 10 0 1
mchron>
```

These commands will produce the following output.



5.8.12 'r' – Repeat

The repeat commands implement a command loop mechanism.

A repeat loop is setup with a repeat-for ('rf') command. Each 'rf' command must be matched with a repeat-next ('rn') command.

```
Commands:
  'rf' - Repeat for
        Arguments: <init> <condition> <post>
                  init: expression executed once at initialization
                  condition: expression determining loop continuation
                  post: expression executed after each loop
  'rn' - Repeat next
```

Usage specifics:

- A repeat loop is skipped immediately when the repeat condition is false at attempting to enter the first loop.
- When used in a command file, each 'rf' must match an 'rn' command in the very same file.
- Repeat loops can be nested without limitation.
- When an 'rf' command is entered at the mchron command prompt, the interpreter will enter a multi-line mode that is completed when an 'rn' command is entered that matches the 'rf' that invoked the multi-line command buildup.
To abort the entry of a multi-line mode 'rf' command, type '<ctrl>d' on an empty line. For an example of this refer to section 5.3.
- Refer to section 5.10 for a detailed description on what will happen internally within mchron upon building up and executing repeat constructs.

Example:

```
mchron> # Demo multi-line 'rf' commands to quickly paint all minutes in a day
mchron> cs 2
mchron> rf h=0 h<24 h=h+1
2>>   rf m=0 m<60 m=m+1
3>>   ts h m 30
4>>   w 20
5>>   rn
6>>   rn
mchron>
```

5.8.13 's' – Statistics

The statistics commands provide performance information on the Emuchron clock stub, the Monochron glcd interface and the Emuchron LCD stub device(s).

```
Commands:
'sp' - Print statistics on stub, glcd and lcd device
'sr' - Reset statistics on stub, glcd and lcd device
```

Usage specifics:

- The stub section provides info on the emulator clock cycle wait stub that is used while executing the 'cf' and 'm' commands.
- The sections on the GLUT and ncurses LCD stubs are provided only when the device is actually being used.

Example:

```
mchron> sp
statistics:
stub   : cycle=75 msec, inTime=4649, outTime=0
        avgSleep=74 msec, minSleep=72 msec
glcd   : dataWrite=65218, dataRead=253648, setAddress=51679
glut   : lcdByteRx=65218, byteEff=52%, bitEff=24%
        msgTx=65219, msgRx=65219, maxQLen=4532, avgQLen=176
        redraws=374, cycles=11115, updates=370, fps=29.5
ncurses: lcdByteRx=65218, byteEff=52%, bitEff=24%
mchron> sr
statistics reset
mchron>
```

The statistics KPI's for the Emuchron stub are as follows:

KPI	Description
avgSleep	The average duration of the time that the emulator is at sleep per cycle. This should be as close as possible to the value of the cycle KPI. Only cycles that are completed as being inTime are taken into account for calculating its value.
cycle	This value represents the duration of a clock cycle as defined by <code>#define ANIMTICK_MS</code> in <code>monomain.h</code> [firmware].
inTime	The number of clock cycles that were completed within the given cycle KPI duration. A clock plugin requires CPU to complete a clock cycle, and in normal operation it should complete way within the cycle duration. Note: Emulator cycles that are run in single cycle mode are not taken into account for calculating the inTime KPI.
minSleep	The duration of the cycle that took most time to complete, resulting in the shortest cycle sleep. Only cycles that are completed as being inTime are taken into account for calculating its value.

KPI	Description
outTime	<p>The number of clock cycles that that were not completed within the given cycle KPI duration. In normal operation this value should be zero as a clock plugin should finish a single cycle way before 75 msec of raw CPU power. If a clock plugin is not able to complete a clock cycle when run in Emuchron on a modern Intel class CPU, it is likely it will not be able to complete the same cycle on a simple 8 Mhz Atmel CPU.</p> <p>Note: As the ncurses LCD interface runs in the same thread as mchcron, flushing the ncurses display will have a negative impact on the clock cycle performance.</p> <p>Note: Emulator cycles that are run in single cycle mode are not taken into account for calculating the outTime KPI.</p> <p>Note: As Emuchron runs as a standard Linux process, it can be interrupted by high priority processes. In an unlikely scenario it may result in outTime to be incremented from time to time even when a clock plugin is perfectly able to complete its clock cycle well within the given timeframe.</p>

Table 18: Emuchron stub statistics

The statistics for the glcd interface are as follows:

KPI	Description
dataWrite	<p>The number of pixel bytes written to the LCD. It is administered by counting the number of calls to <code>glcdDataWrite()</code>. Its value should normally be identical to KPI <code>lcdByteRx</code> in both the glut and ncurses statistics sections.</p>
dataRead	<p>The number of pixel byte read operations from the LCD. It is administered by counting the number of calls to <code>glcdDataRead()</code>.</p> <p>Note: This number does not fully represent the actual number of LCD pixel bytes read. After a write operation or when switching between LCD controllers, the hardware requires two sequential read operations of which the first is a dummy read.</p>
setAddress	<p>The number of explicit requests to set the LCD display cursor. It is administered by counting the number of calls to <code>glcdSetAddress()</code>.</p> <p>Note: Upon calling a <code>glcdDataWrite()</code> or a non-dummy <code>glcdDataRead()</code> the internal LCD display cursor is automatically incremented to the next LCD byte. The automatic hardware increment action is not included in this KPI.</p>

Table 19: Monochron glcd interface statistics

The statistics KPI's for the GLUT LCD stub are as follows:

KPI	Description
avgQLen	<p>This KPI is calculated by dividing KPI <code>msgRx</code> by KPI <code>updates</code>. It gives the average length of the GLUT message queue to be processed.</p>
bitEff	<p>The percentage of bits in a processed LCD byte that will lead to a change in the LCD display. Only LCD bytes that will lead to a change in the LCD are taken into account for calculating this KPI. In the example above, out of 8 bits/pixels per byte, on average about two pixels per LCD byte will lead to a change in the LCD display.</p>

KPI	Description
byteEff	The percentage of processed LCD bytes that will lead to at least one pixel change in the LCD display. An LCD byte contains 8 bits/pixels. If at least one of these pixels differs from the actual content in the LCD display the byteEff KPI will increase. In the example above, $100-52=48\%$ of all processed LCD bytes will not change anything on the LCD display.
cycles	The number of GLUT thread cycles in which internal GLUT events and the GLUT message queue are processed. Such a cycle may or may not lead to a GLUT window redraw.
fps	This is the frames per second redraw rate of the GLUT window. The GLUT thread has a sleep cycle of 33 msec, giving a theoretical refresh rate of ~ 30 fps. In practice this will be lower due to the processing power needed to process the GLUT message queue and to redraw its window, in combination with latency caused by the Linux thread and process scheduler.
lcdByteRx	The number of LCD bytes (with 8 pixel bits) that are received in the interface. This value is incremented whenever a byte is written to the LCD display.
maxQlen	The GLUT interface runs in its own thread. The GLUT thread can be at sleep while mchron or clock plugins send messages to the GLUT interface. This queue of messages will be waiting to be processed when the GLUT thread wakes up. This KPI shows the maximum length of the GLUT message queue that is waiting to be processed.
msgRx	The number of LCD commands processed by the GLUT interface. Note that in the example above the msgRx KPI is one higher than the lcdByteRx KPI. This is explained by the single backlight set command sent to the GLUT interface at mchron initialization time.
msgTx	The number of LCD commands sent to the GLUT interface. It includes commands to process an LCD byte, to process a change in LCD backlight and shutting down the GLUT interface. In the example above notice that msgTx and msgRx are identical, which is normally the case. The values may differ when the statistics are reset while GLUT messages are still waiting to be processed.
redraws	This KPI shows the total number of GLUT window redraws. The GLUT thread is forced to redraw its display in two scenarios. The first is by processing the messages in the GLUT message queue as sent by mchron and/or a clock plugin. When all messages from the queue have been processed and at least one display change is detected, the GLUT window is instructed to redraw itself. The second is internal to GLUT itself. Whenever the GLUT window is resized, when another window moves over the GLUT window or when the GLUT window is minimized or restored, an internal GLUT redraw event is generated.
updates	This KPI shows the total number of GLUT window redraws caused by processing messages in the GLUT message queue. Note: As the redraws KPI also includes updates caused by messages in the GLUT message queue, the difference between the updates and redraws KPI's will give the number of GLUT redraws caused by internal GLUT events.

Table 20: Emuchron GLUT statistics

The few statistics KPI's for the ncurses LCD stub are identical to their counterparts in the GLUT interface.

Note that in the example output above the values of the ncurses statistics are identical to their GLUT counterparts. This is explained by the fact that both stub

devices have implemented identical mechanisms to optimize draw behavior and do statistics administration.

<i>KPI</i>	<i>Description</i>
bitEff	The percentage of bits in a processed LCD byte that will lead to a change in the LCD display. Only LCD bytes that will lead to a change in the LCD are taken into account for calculating this KPI. In the example above, out of 8 bits/pixels per byte, on average about two pixels per LCD byte will lead to a change in the LCD display.
byteEff	The percentage of processed LCD bytes that will lead to at least one pixel change in the LCD display. An LCD byte contains 8 bits/pixels. If at least one of these pixels differs from the actual content in the LCD display the byteEff KPI will increase. In the example above, $100\% - 52\% = 48\%$ of all processed LCD bytes will not change anything on the LCD display.
lcdByteRx	The number of LCD bytes (with 8 pixel bits) that are received in the interface. This value is incremented whenever an LCD byte is written to the LCD display.

Table 21: Emuchron ncurses statistics

5.8.14 't' – Time

The time commands allow setting, resetting and reporting the time as used in mchron and forcing a clock to update itself using the mchron time. Related command groups are alarm ('a') and date ('d').

```
Commands:
'tf' - Flush Monochron time and date to active clock
'tp' - Print time/date/alarm
'tr' - Reset time to system time
'ts' - Set time
      Arguments: <hour> <min> <sec>
                hour: 0..23
                min: 0..59
                sec: 0..59
```

Usage specifics:

- When a time command is used, except for 'tp', an active clock is called to update itself using the modified settings.
- When setting a time manually, an offset is calculated between the system time and the requested time. This offset will then be used as a delta between the system time and the mchron time.

Example:

```
mchron> # Get a basic digital clock
mchron> cs 3
mchron> # Print the current time/date/alarm (clock layout is not updated)
mchron> tp
time   : 11:10:55 (hh:mm:ss)
date   : 30/10/2013 (dd/mm/yyyy)
alarm   : 22:09 (hh:mm)
alarm   : off
mchron> # Set time to near happy hour (clock layout will update)
mchron> ts 16 45 00
time   : 16:45:00 (hh:mm:ss)
date   : 30/10/2013 (dd/mm/yyyy)
alarm   : 22:09 (hh:mm)
alarm   : off
mchron> # Reset to system time (clock layout will update)
mchron> tr
time   : 11:12:07 (hh:mm:ss)
date   : 30/10/2013 (dd/mm/yyyy)
alarm   : 22:09 (hh:mm)
alarm   : off
mchron> # Wait a few minutes...
mchron> # Flush current mchron time to active clock (clock layout will update)
mchron> tf
time   : 11:14:32 (hh:mm:ss)
date   : 30/10/2013 (dd/mm/yyyy)
alarm   : 22:09 (hh:mm)
alarm   : off
mchron>
```

5.8.15 'v' – Variable

Mchron supports named variables representing a `double` type value that can be used in expressions for numeric command arguments.

```
Commands:
'vp' - Print value of variable(s)
      Argument: <variable>
           variable: word of [a-zA-Z] characters, '*' = all
'vr' - Reset variable(s)
      Argument: <variable>
           variable: word of [a-zA-Z] characters, '*' = all
'vs' - Set value of variable
      Argument: <assignment>
           assignment: <variable>=<expression>
```

Usage specifics:

- A variable name is identified by any mixed combination of upper/lowercase characters in the range 'a'..'z', excluding reserved function and constant keywords.
Examples: x (=ok), radius (=ok), myLocalVar (=ok), a1 (=bad), ab_c (=bad), true (=bad)
- Variables must explicitly be set a value before being allowed to be used in expressions.
- Refer to the script on the front page for an example on using variables hor, ver and factor in multiple commands.

Example:

```
mchron> # Try to initialize a few variables
mchron> vs rank=10
mchron> vs f=key
variable not in use: key
assignment? parse error
mchron> vs key=rank*4
mchron> # Show all variables currently in use
mchron> vp *
key=40    rank=10
variables in use: 2
mchron> # Set another variable and reset an active one
mchron> vs index=key*rank
mchron> vr rank
mchron> # Show what is left
mchron> vp *
key=40    index=400
variables in use: 2
mchron> # Reset all active variables
mchron> vr *
mchron> vp *
variables in use: 0
mchron>
```

5.8.16 'w' – Wait

The wait command will make mchron wait.

```
Command:
'w' - Wait for keypress or amount of time
Argument: <delay>
        delay: 0 = wait for keypress, 1..1000000 = wait delay*0.001 sec
        When waiting for keypress, a 'q' will return control back to the
        mchron command prompt
```

Usage specifics:

- The wait command supports two flavors. One flavor will wait a dedicated amount of time and another waits for an end-user keyboard keypress. With respect to the latter, the 'q' key will return control back to the mchron command prompt.
- The wait command is used in many scripts to temporarily halt script execution or wait a while prior to updating the LCD display with new information.

Example:

```
mchron> # Wait one second
mchron> w 1000
mchron> # Wait for keypress
mchron> w 0
<wait: press key to continue>
mchron>
```

5.8.17 'x' – Exit

The exit command will exit mchron.

```
Command:  
'x' - Exit
```

Usage specifics:

- The 'x' command can only be used at mchron command prompt level.

Example:

```
mchron> # Exit mchron  
mchron> x  
$
```

5.9 Processing an mchcron 'hello world!' command

Mchcron supports many commands. For the sake of stability and consistency a common approach has been implemented to scan and parse commands and command arguments.

It is chosen not to implement the command scanner and parser in flex and bison. Instead, dedicated scanner and parser functionality has been created to fit mchcron purposes. The main reason for this is that flex and bison are considered by many to be not easy to comprehend and work with, including the author, making it difficult to find out how-it-all-works.

In the example below is depicted and explained on what will happen when an mchcron command is entered to paint a text string on the LCD display.



Figure 9: Processing an mchcron 'hello world!' command

Step 1:

The user enters a 'pa' (paint ascii) command using the keyboard, or has it prepared in an mchcron command file.

Main command processing takes place in `emuLineExecute()` in `mchcronutil.c` [firmware/emulator].

Command and argument scanning takes place in `cmdArgScan()` in `scanutil.c` [firmware/emulator].

Step 2:

In `emuLineExecute()` the main command will be scanned from the input string based on an `cmdArg_t` structure array. In this case, the scanner needs to scan a single word, as instructed by `ARG_WORD`. The functional name of the argument is "command" that can be used to provide end-user feedback in case an error occurs. The scanned argument is put in a dedicated array for storing argument text words, being `argWord[]`. The end result of the scan is that `argWord[0]` will contain the text "pa".

For an `mchron` command its associated command dictionary is retrieved using `cmdDictCmdGet()`. The command dictionary not only provides information regarding the command itself but also about the command arguments and its handler function. This means that the rest of the generic command handling is based upon information from the command dictionary.

Step 3:

In `emuLineExecute()` the remaining part of the command will be scanned, parsed and processed. In the command dictionary for the 'pa' command a dedicated `cmdArg_t` structure array is defined that will drive the scan of all its remaining arguments. For details refer to step 4a and 4b.

Step 4a:

Each command argument is now scanned and parsed. In case the data type of an argument is `ARG_NUM` or `ARG_UNUM` (unsigned number) its argument value is considered to be a mathematical expression with named variables. The text string of the argument will be fed into an expression evaluator that will return a `double` type value. Whenever the expression evaluator encounters a problem an error message is provided.

Additional functionality for an argument value is provided via structure `cmdArgDomain_t` where the argument value is matched with a domain profile. This prevents repetitive and error-prone argument value verification in the command handlers. In our example, the 'color' character argument must have either value 'b' or 'f', and the 'x' unsigned number argument may not exceed the maximum value 127. Each argument refers its own argument domain structure. Whenever an argument does not match its domain profile an error message is provided.

In general, a domain profile will take care of properly validated argument values, but in some cases additional domain value verification is required. If so, it needs to be implemented in the appropriate command handler in step 5b below.

Step 4b:

The end result value of each of the arguments is copied into dedicated argument arrays for characters, doubles and a string. They are respectively `argChar[]` and `argDouble[]` and `argString`. In the example above, the `ARG_WORD` font argument is added in `argWord[]` as an additional array element.

Step 5a+5b:

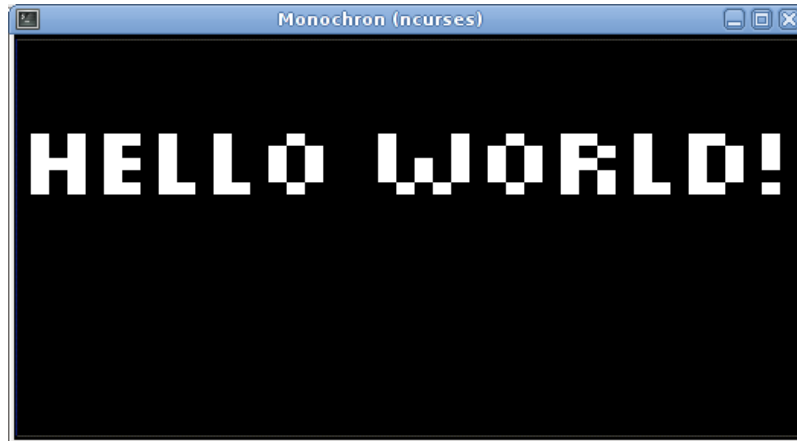
When the command line has been fully scanned and parsed, all command argument values are now available for final processing.

The command handler function for the 'pa' command is referenced via the command dictionary, in this case `doPaintAscii()` that is defined in `mchron.c` [firmware/emulator]. That function is now called.

In `doPaintAscii()`, after converting the color and font arguments into an enum value, function `glcdPutStr3()` is called to paint the requested text string on the LCD.

When the command has been processed, control is given back to the caller of `emuLineExecute()`.

When completed, the content of the LCD stub device will appear as below.



5.10 Building and executing an mchron command list

Single line commands in mchron are executed as described in section 5.9. However, mchron also supports executing multi-line commands.

Executing a multi-line command is invoked via two methods:

- Use the execute command 'e' to load and execute mchron commands prepared in a plain text file.
- Use the repeat-for 'rf' or if-then 'iif' command to enter and execute a list of mchron commands interactively via the command prompt.

With respect to the first method consider the following imaginary mchron script below as saved in a plain text file. From a functional point of view it is identical to the time-hm.txt [script] script, except that all variables, instructions, comments and white lines are removed.

```
# Demo script
cs 2
rf h=0 h<24 h=h+1
  rf m=0 m<60 m=m+1
    ts h m 30
    w 50
  rn
rn
```

This imaginary script can be invoked by the mchron execute command.

```
mchron> e s ../script/imaginary.txt
```

With respect to the second method consider the repeat-for 'rf' command below that will invoke an interactive buildup of the commands to be executed. The commands will be executed when an 'rn' command is entered that matches the 'rf' that invoked the interactive command buildup.

Note: To abort the entry of an interactive 'rf' command type '<ctrl>d' on an empty line.


```

mchron> # Demo multi-line command entry via 'rf' to paint all minutes in a day
mchron> cs 2
mchron> rf h=0 h<24 h=h+1
2>>   rf m=0 m<60 m=m+1
3>>   ts h m 30
4>>   w 50
5>>   rn
6>>   rn
mchron>

```

Using the demo script of the first method as an example, upon entering the 'e' (execute) command the following will take place.

Step 1: Load the file contents in linked lists.

- The 'e' command is interpreted in `emuLineExecute()`. This function will then invoke the handler of the execute command, being `doExecute()`.
- In `doExecute()` function `cmdListFileLoad()` is called to load the file contents into linked list structures as depicted in figure 10 below. Part of loading the file is matching each line with its associated command dictionary entry. Also, the integrity of the command list is checked by matching each 'rf' command with an 'rn' command. When an unknown command is encountered or repeat commands cannot be matched, file loading will abort.
- Two pointers are available that administer the root of the linked lists, being `cmdLineRoot` and `cmdPcCtrlRoot`.

`doExecute()/emuLineExecute() + emuListExecute()`

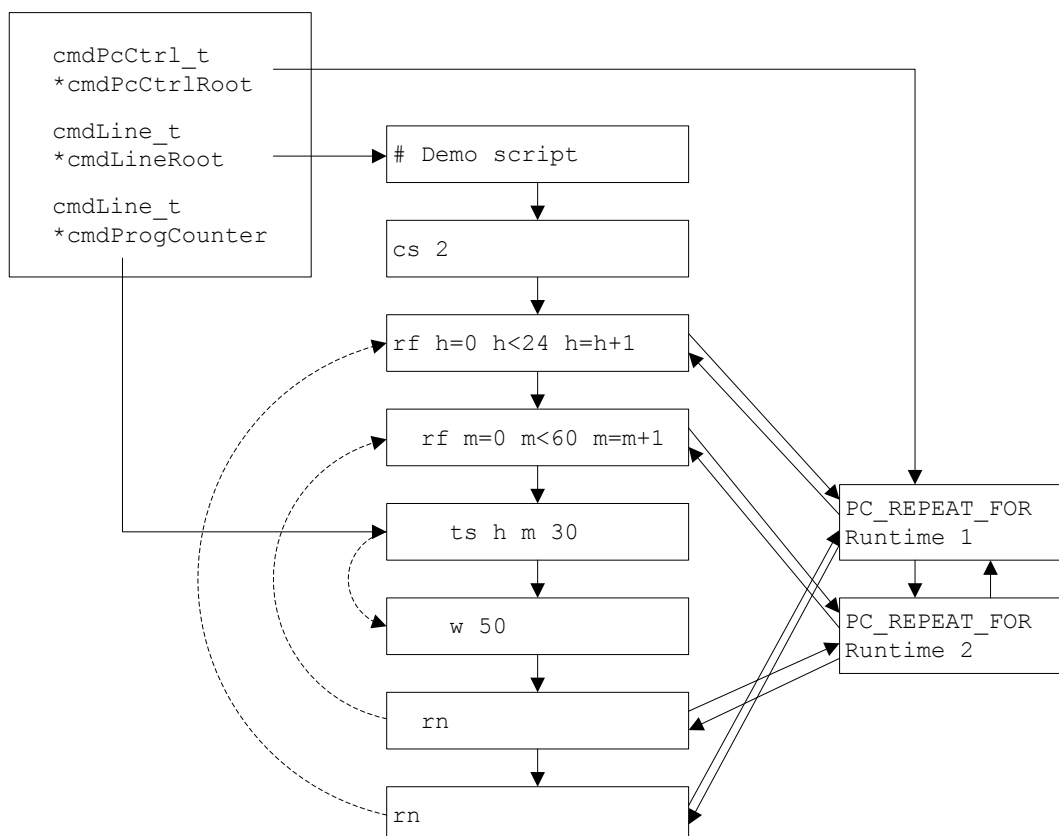


Figure 10: Creating and executing an mchron command list

Step 2: Execute the commands in the linked list.

- When loaded, in `doExecute()` the linked list structure is then executed via function `emuListExecute()`. In this function a third pointer, `cmdProgCounter`, is available that will serve as a list execution program counter.
- In `emuListExecute()` the program counter pointer is used to execute all the commands in the linked list one by one. The program counter will of course start at the top of the list using the root pointer.
- Execution of the list is interrupted by pressing the 'q' key.
- Each command in the list is executed in `emuLineExecute()`.
- When a non-repeat command in the list has been executed, the program counter is incremented to point to the next list element.
- However, for repeat commands its handler will initialize and/or process the repeat condition via the appropriate repeat-for runtime structure. Via this structure the program counter can be changed, thus making the linked list loop or continue at the 'rn' command of a repeat construct.
- List execution ends when a list element has no pointer to a next one.
- When list execution is completed, command and control block list cleanup will take place after which `doExecute()` returns control back to its caller.

Next to repeat-for constructs, mchron also support if-then-else constructs. The basics of creating a linked list using if-then-else logic is identical to repeat-for constructs; create appropriate `cmdPcCtrl_t` structures and link them to associated command line `cmdLine_t` structures. The runtime execution logic for if-then-else constructs will of course differ from repeat-for constructs. Repeat-for and if-then-else constructs can be mixed in the same command list into any depth. An example of this can be found in `circle4.txt` [script]. An example of nesting repeat-for commands with considerable depth is found in `nesting.txt` [script].

6 Debugging clock and graphics code

Prior to Emuchron the only method to debug clock and graphics function code was to build and upload firmware into the Monochron clock that produces debug output strings. These output strings are sent from the Monochron clock over the FTDI bus to the connected computer where they are picked up in a terminal program.

This debug method still applies to Emuchron. With Emuchron however the user can debug clock and graphics functions using the standard gdb debugger and any GUI on top of that, prior to having its resulting firmware uploaded to the Monochron clock. This makes it a superior debugging experience when compared to the FTDI method.

This does not mean that the FTDI method has become obsolete. It is possible that due to bugs in the stub layer of Emuchron or due to bugs in clock or graphics code, Emuchron will behave different than the Monochron low-level firmware. A good rule on this is as follows: as long as clock or graphics code does not directly interact with (stubbed) low-level firmware, the chance of mismatched behavior between Emuchron and Monochron is considered low. Furthermore, Emuchron provides a stub on the FTDI debug method, allowing the application to write debug strings in a plain text file, making it a useful addition to the gdb debug solution.

6.1 Debugging using the FTDI debug strings method

6.1.1 Requirements and limitations

By default, the debug string method is disabled in the firmware code. The reason for this is that it produces a much larger firmware file that depends on the amount of debug strings and the size of the debug library that needs to be linked into the final firmware.

The master switch for the debug string method is found in monomain.h [firmware].

```
// Debugging macros.  
// Note that DEBUGGING is the master switch for generating debug output.  
// 0 = Off, 1 = On  
#define DEBUGGING 0
```

When changed it is required to rebuild Monochron and/or Emuchron.

The several methods to generate debug strings are macros and functions as exposed in monomain.h [firmware] and util.h [firmware].

In Emuchron the stubs for these are found in stub.h [firmware/emulator].

Many examples of debug strings are found throughout the firmware and emulator source code.

6.1.2 Monochron debug strings via FTDI port on Debian Linux

The connection specifics for a terminal program that connects to Monochron are as follows:

```
FTDI debug string output connection settings:  
Bits per second: 38400  
Data bits: 8  
Parity: None  
Stop bits: 1  
Flow control: None
```

Note that the configuration profile connection specifics have proven to work in combination with FTDI Friend v1.1 (<https://learn.adafruit.com/ftdi-friend>). When using other means of connecting Monochron with a USB cable other connection settings may apply, such as a baudrate of 19200.

When proper debug string enabled firmware has been uploaded to Monochron connect it to the computer via a USB cable. When Debian is used as a VM, have the FTDI USB device attached to your VM.

The instructions below cover the use of the Linux `minicom` program. Refer to section 3.5 to install a pre-configured Monochron connection profile for minicom.

- By default the logical `/dev/ttyUSBx` device that represents the hardware FTDI USB device is accessible to root only. Decide to run minicom either as root, or use `chmod` on the `/dev/ttyUSBx` device to grant access to other users.
- Start minicom from a shell prompt. In the example below minicom is executed using the root user. Note the command line arguments for minicom.

```
$ su - root
$ # Make minicom capture output to logfile Monochron.log and use the
$ # Monochron profile (installed per instructions in section 3.4)
$ minicom -C Monochron.log Monochron
```

- When minicom is started it connects to Monochron. At that point Monochron will restart and debug strings should be pouring into the minicom terminal and the capture log file `Monochron.log`.

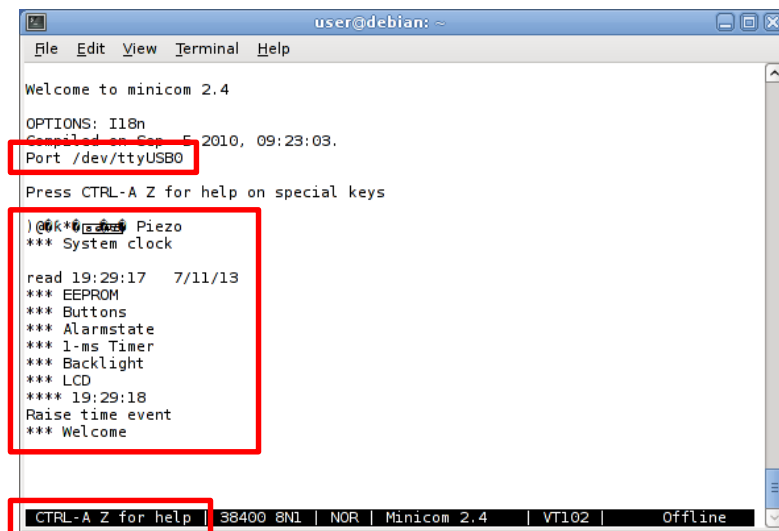


Figure 11: Minicom receiving Monochron debug string

- For help on minicom enter '<ctrl>az'. See below.

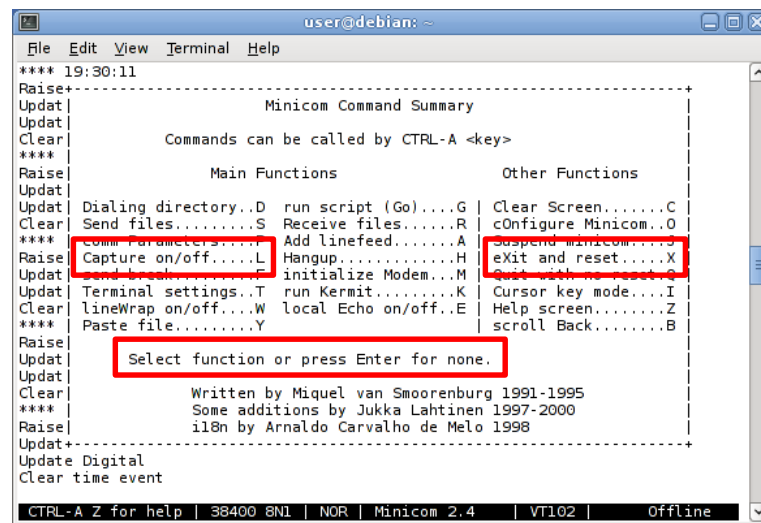


Figure 12: Minicom command summary via '<ctrl>az'

- In another command shell use the following command to trace the contents in the minicom capture log file.

```
$ su - root
$ tail -f Monochron.log
```

Note: Do not have an open connection in minicom or another terminal program while attempting to connect to Monochron via avrdude, or vice versa. The application that has access to Monochron will keep the connection locked and will prevent any other connection request to succeed.

6.2 Debugging using Emuchron stubbed FTDI debug strings

This is the stubbed version of the Monochron FTDI debug strings method. For general info on this method refer to section 6.1.

To re-iterate, to use the debug string output method in Emuchron a rebuild is required with the `DEBUGGING` master switch set to 1, causing the object size to grow. While object size is of great importance for Monochron firmware, for Emuchron it is of no concern.

When rebuilt, mchcron must be started with the `-d` flag to specify the debug log output file. See below.

```
$ ./mchcron -d debug.log
```

Note that if mchcron is built with the master switch set to 0, mchcron will report that debug output cannot be used when invoked with the `-d` flag. See red text below.

```
$ ./mchron -d debug.log

*** Welcome to Emuchron command line tool (build Oct 10 2015, 11:37:18) ***

WARNING: -d <file> ignored as master debugging is Off.
Assign value 1 to "#define DEBUGGING" in monomain.h [firmware] and rebuild
mchron.

mchron PID = 3157

time  : 11:39:02 (hh:mm:ss)
date   : 10/10/2015 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off

Enter 'h' for help.
mchron>
```

Assuming that mchron was properly built, to examine the output log being created open another terminal and type the following commands.

```
$ cd <install_dir>/firmware
$ tail -f debug.log
```

Example output that is generated in file debug.log after entering the mchron command 'm n k' (to start the stubbed Monochron application) is as follows. Note that the output is very identical to output when recorded via minicom as shown in section 6.1.2.

```
$ tail -f debug.log
**** logging started
Clear time event
Raise time event
*** UART
*** Piezo
*** System clock

read 9:25:26 25/10/13
*** EEPROM
*** Buttons
*** Alarmstate
*** 1-ms Timer
*** Backlight
*** LCD
*** Welcome
*** Start initial clock
Clear time event
**** 9:25:29
Raise time event
Init Digital
Alarm info -> Other
*** Init clock completed
(etc..)
```

6.3 Debugging Emuchron using gdb

Emuchron and its mchron frontend are built with gcc option `-g`, thereby always generating gdb-ready symbolic debugging object code.

The gdb debugger is command-line driven. However, there are many GUI frontends available. In this manual we consider the use of Nemiver and DDD.

For help on using Nemiver and DDD refer to its built-in help menu item. When using only the GLUT LCD device, the mchron program can be loaded and started in gdb with Nemiver or DDD immediately. In this sense, gdb is not limited by the GLUT device in mchron.

The downside of debugging with the GLUT LCD device is that GLUT runs in its own thread, making LCD updates asynchronous from glcd graphics requests from the clocks. This makes the GLUT LCD device less suited for debugging sessions when LCD output is relevant.

Things are different though when using the ncurses LCD device. This device runs in the same thread as mchron. And as the ncurses display is actively flushed in every clock cycle, it is therefore always in-sync with the mchron application. This makes the ncurses LCD display much better suited for debugging purposes when LCD output is relevant.

6.3.1 Requirements for Debian 8 when using gdb

When using Debian 8, there are gbd requirements with respect to referencing glibc sources. These requirements are described in section 3.7.4.

6.3.2 Limitations on using ncurses

There is a downside to using the ncurses library in combination with gdb. In short, gdb and the ncurses library don't like one another. In order to get ncurses properly working in gdb, it requires that ncurses is initialized prior to the gdb environment. If gdb initializes itself before ncurses can do so, ncurses will redirect its output always to the gdb command prompt shell, regardless the configured ncurses output tty.

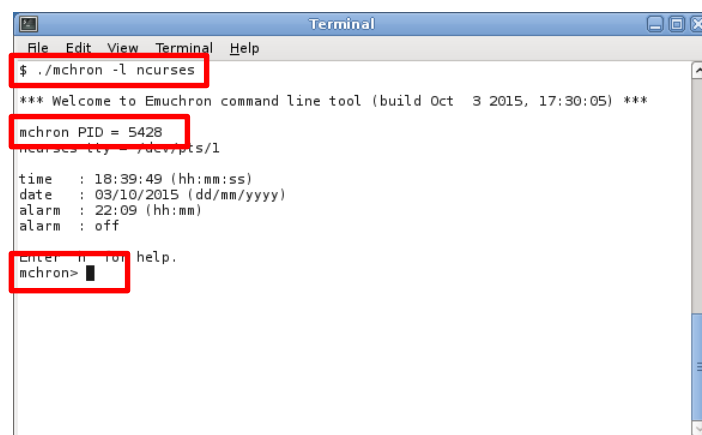
The only way to get ncurses to work with gdb properly is first to start mchron, thereby allowing ncurses to initialize itself properly, and only then attach gdb (with Nemiver or DDD) to the running mchron process.

When this ncurses/gdb debug startup sequence method is applied, no other limitations apply.

However, depending on the GUI front-end being used, different steps need to be taken. In the sections below is explained on a step-by-step basis how to get an ncurses LCD display functioning properly in a gdb debugging session.

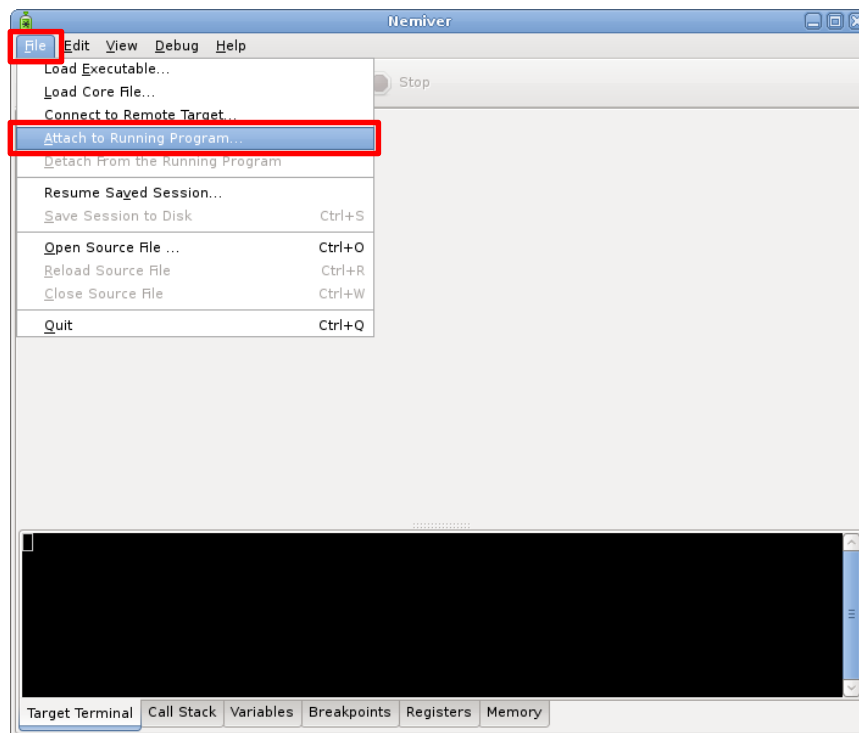
6.3.3 Debugging Emuchron with ncurses device using Nemiver

First startup mchron and make sure there is a command prompt.



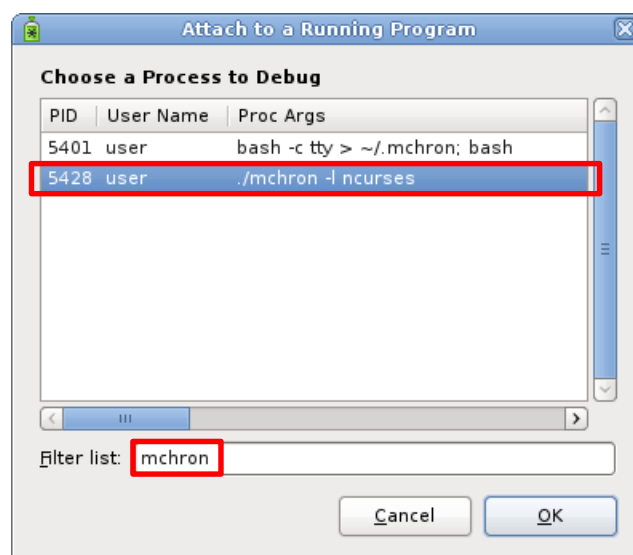
```
Terminal
File Edit View Terminal Help
$ ./mchron -l ncurses
*** Welcome to Emuchron command line tool (build Oct  3 2015, 17:30:05) ***
mchron PID = 5428
mchrones tty = /dev/tty/1
time   : 18:39:49 (hh:mm:ss)
date   : 03/10/2015 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off
Enter 'h' for help.
mchron>
```

Then, start Nemiver and select "File→Attach to Running Program..." to attach to a running process.



In the popup list search for the mchcron command and click 'OK'.

Note: You can use the reported mchcron PID in the mchcron shell as a cross reference in the list below. For quick process pre-filtering enter 'mchcron' in the filter list.

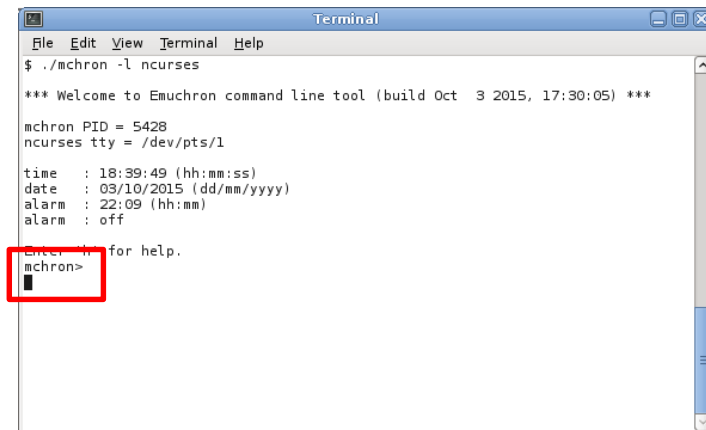


Nemiver now tries to attach to the process, but for now it cannot. The reason for this is that the mchcron process is not active at this time as it waits for a command on the command line.

So, what needs to be done is to enter a blank command by hitting the return key in the mchcron console. When hit, mchcron now seems to hang as the mchcron process is brought under control of Nemiver. See below.

Note: The cursor being at the beginning of the next line is optional. Upon pressing the return key, the cursor may remain static at its current location at

the end of the prompt. In any case, mchron seems to hang as it is brought under control of gdb.



```

Terminal
File Edit View Terminal Help
$ ./mchron -l ncurses

*** Welcome to Emuchron command line tool (build Oct  3 2015, 17:30:05) ***

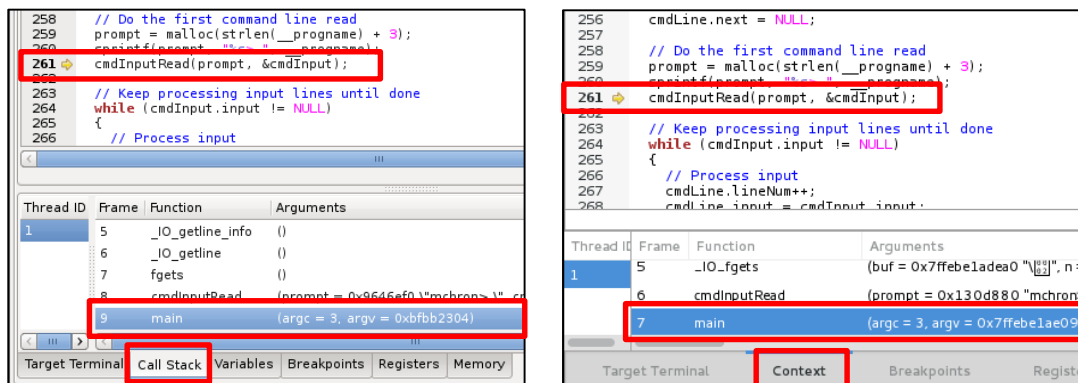
mchron PID = 5428
ncurses tty = /dev/pts/1

time  : 18:39:49 (hh:mm:ss)
date  : 03/10/2015 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : off

Enter '?' for help.
mchron>

```

In Nemiver we are now be able to browse the application sources. The easiest way to open the first source is to go to the tab that provides the runtime call stack and select the lowest call stack level available, which is `main()` in the `mchron.c` [firmware/emulator] source. See below. Note that in Nemiver in Debian 6 the tab is named 'Call Stack' (left screendump) whereas in Debian 7 and 8 the tab is named 'Context' (right screendump).



The left screenshot shows the 'Call Stack' tab in Nemiver. The source code for `mchron.c` is displayed at the top, with line 261 highlighted. Below the code, the call stack is shown with the following entries:

Thread ID	Frame	Function	Arguments
1	5	_IO_getline_info	()
	6	_IO_getline	()
	7	fgets	()
	8	cmdInputRead	(prompt = 0x9646ef0 "mchron>", n = 3)
	9	main	(argc = 3, argv = 0xbfb2304)

The 'main' function is selected, and the 'Call Stack' tab is active.

The right screenshot shows the 'Context' tab in Nemiver. The source code for `mchron.c` is displayed at the top, with line 261 highlighted. Below the code, the context is shown with the following entries:

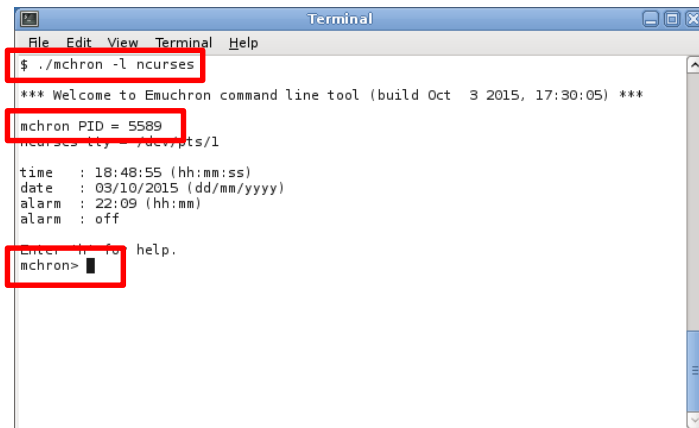
Thread ID	Frame	Function	Arguments
1	5	_IO_fgets	(buf = 0x7ffeb1adea0 "mchron>", n = 3)
	6	cmdInputRead	(prompt = 0x130d880 "mchron>", n = 3)
	7	main	(argc = 3, argv = 0x7ffeb1ae09)

The 'main' function is selected, and the 'Context' tab is active.

From this point on you are able to open any Emuchron source, set, disable and re-enable breakpoints, and verify local and global data. For more information on using Nemiver use the 'Help' menu.

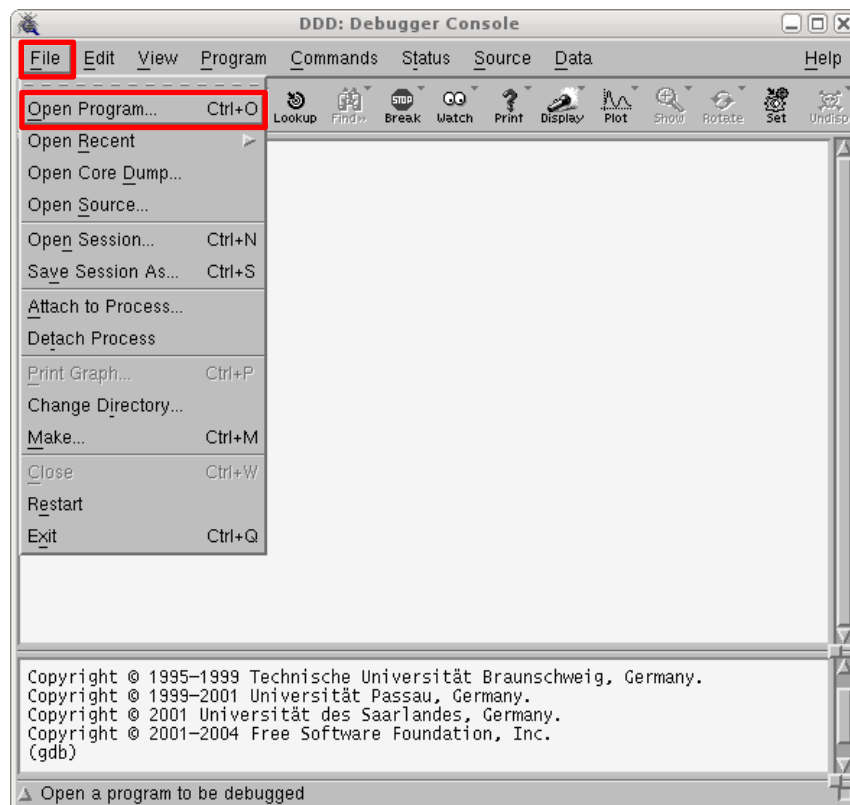
6.3.4 Debugging Emuchron with ncurses device using DDD

First startup mchcron and make sure there is a command prompt.

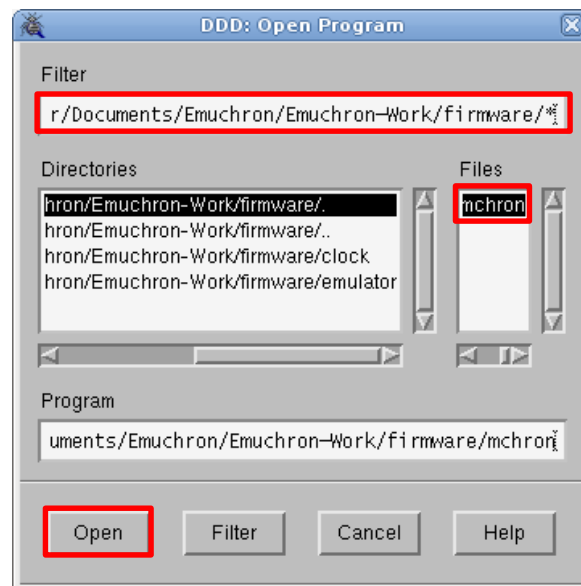


```
Terminal
File Edit View Terminal Help
$ ./mchcron -l ncurses
*** Welcome to Emuchron command line tool (build Oct  3 2015, 17:30:05) ***
mchcron PID = 5589
mchcron tty = /dev/pts/1
time   : 18:48:55 (hh:mm:ss)
date   : 09/10/2015 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off
Enter h for help.
mchcron>
```

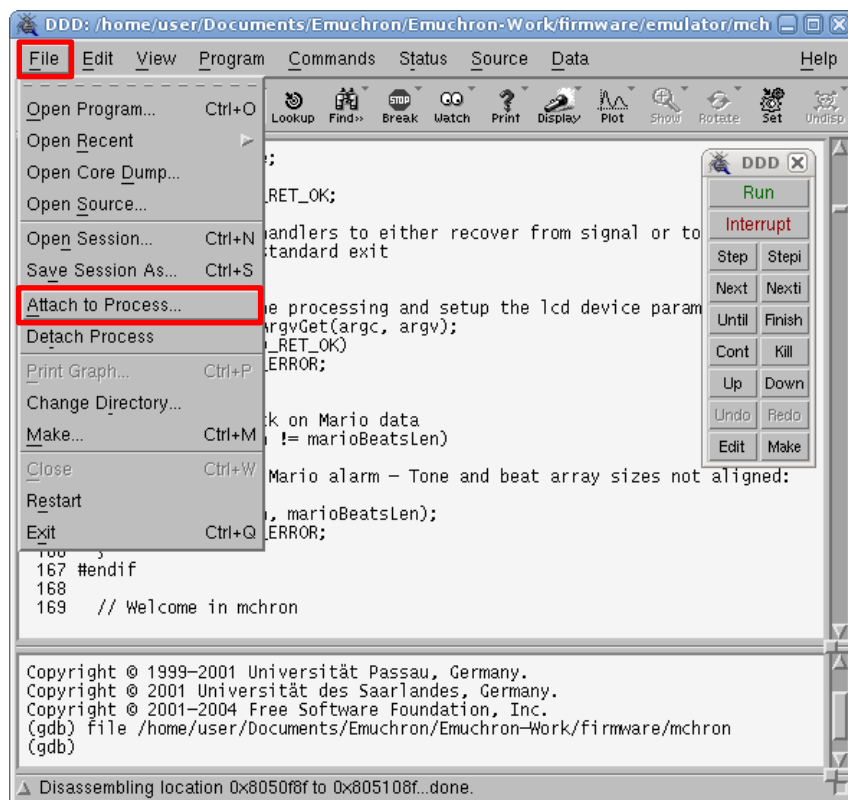
Then, start DDD and select "File→Open Program..." to open an executable program.



In the form browse to the <install_dir>/firmware folder, select the mchcron program and click 'Open'.

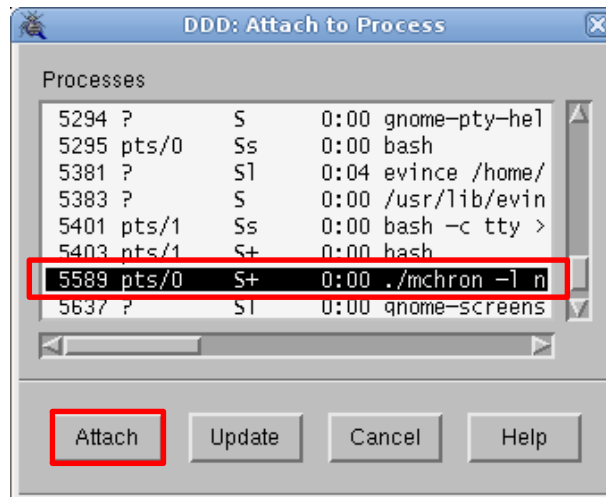


DDD should now display the mchron.c [firmware/emulator] source file, but we're not running an actual debug session yet. For this, attach to the running mchron process using "File→Attach to Process...".



In the popup list search for the mchron command and click 'Attach'.

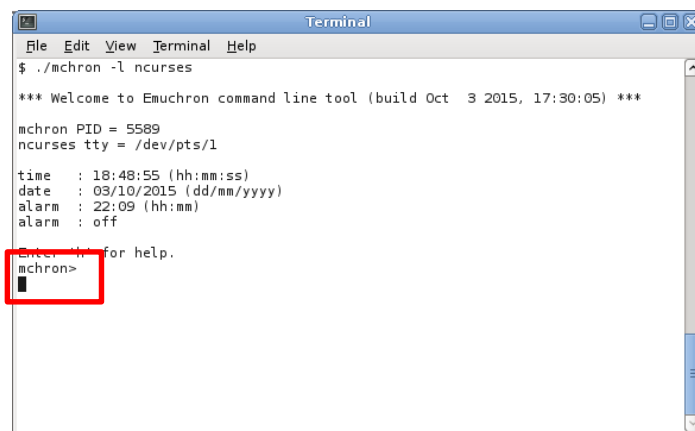
Note: You can use the reported mchron PID in the mchron shell as a cross reference in the list below.



DDD now tries to attach to the process, but for now it cannot. The reason for this is that the mchron process is not active at this time as it waits for a command on the command line.

So, what needs to be done is to enter a blank command by hitting the return key in the mchron console. When hit, mchron now seems to hang as the mchron process is brought under control of DDD. See below.

Note: The cursor being at the beginning of the next line is optional. Upon pressing the return key, the cursor may remain static at its current location at the end of the prompt. In any case, mchron seems to hang as it is brought under control of gdb.



In DDD, from this point on, you are able to open any Emuchron source, set, disable and re-enable breakpoints, and verify local and global data. For more information on using DDD use the "Help" menu.

6.3.5 Debugging an mchron coredump file

The method of debugging an mchron coredump file does not differ from coredumps of other applications. For this, refer to the "Help" menu of Nemiver and DDD and the `man` page of `gdb`.

7 Frequently asked questions

7.1 Differences between Monochron and Emuchron

To re-iterate, Emuchron is meant to be used to debug and test functionality implemented in clock plugins and high-level graphics code. Low-level Monochron firmware routines and interrupt handlers are out-of-scope. Refer to figure 2 and figure 3 that depict the two runtime environments.

Monochron uses several interrupt handlers to take care of button presses, scanning the real-time clock (RTC) and controlling the audible alarm. As such it is considered to be a kind of multi-threaded application. Emuchron does not implement this approach for the sake of simplicity.

This means that on a certain level the runtime behavior of both environments will start to differ. However, the areas in which both applications won't differ are the functional clock plugins and the high-level glcd graphics modules, and this is what matters most.

Because of this difference in implementation, the programmer must be aware of the fact that whenever low-level code is touched, code in Monochron may not work properly in Emuchron, or vice versa. But, again, when restricting oneself to clock plugin and high-level glcd graphics code, no impact is to be expected.

The most high-level example showing the consequences of the different runtime behaviors is found in `mchronTimeInit()` in `monomain.c` [firmware]. In this function the requested functionality requires fully dedicated code sections for Monochron and Emuchron.

7.2 Linux mathlib accuracy vs. AVR mathlib accuracy

Monochron is built using AVR libraries whereas Emuchron is built using Linux libraries. The AVR libraries are built keeping in mind that both memory and CPU capacity is limited. These restrictions are much less of a concern to Linux libraries where focus is also put on accuracy and completeness.

When using integer math, both the AVR and Linux libraries have shown to be completely compatible. However, when using mathematical functions based on `float` or `double` types, AVR and Linux libraries tend to differ.

In a nutshell, the AVR mathlib is much less accurate than the Linux mathlib.

A good example on how this will impact clock plugin code is found in `mosquito.c` [firmware/clock]. In this clock a `float` type is used to move a time element over the LCD display in separate x and y directions. To determine the cut-off values on which a floating time element will bounce off a display border, a certain threshold needs to be implemented to counteract the inaccuracy of the AVR mathlib.

See the example below where cut-off values 1.00 and 2.00 include a 1% inaccuracy compensation (1.01 and 2.02), which has proven to be far more than adequate.

```
// Check bouncing on left and right wall
if (mathPosXNew + element->textOffset - 1.01 <= 0L)
{
    mathPosXNew = -(mathPosXNew + 2 * element->textOffset - 2.02);
    element->dx = -element->dx;
}
```

Note that the code to compensate for inaccuracies is mostly not needed in Emuchron as it uses the very accurate Linux mathlib. The tricky part in here is to realize that a clock in Monochron may show a slightly different behavior in Emuchron, based on the mathematical functions used.

Giving another example:

You may see that the position of individually painted pixels in Emuchron and Monochron sometimes are off by one x and/or y value when `sin()` and `cos()` are used to determine its position. When pixel positions are well within the boundaries of the LCD display this is normally not of a concern. But, as the code example above shows, whenever a pixel position may result in an underflow or overflow value for LCD display locations this needs to be properly taken care of.

Important note:

All glcd graphics functions are implemented using integer math. As such, the graphics behavior of glcd functions will not differ between Monochron and Emuchron.

7.3 Accuracy and reliability of the expression evaluator

For numeric command arguments and variable assignment operations the mchron interpreter uses an expression evaluator implemented in flex and bison.

In the expression evaluator all calculations are done in type `double`. It will return an error in case of an overflow, a division by zero or a modulo by zero operation.

The logic for comparing two `double` values for being equal is based on relative accuracy cutoff value `epsilon`. Both the comparison function `exprCompare()` and `epsilon` are defined in `expr.y` [firmware/emulator].

```
// The relative accuracy of comparing values being equal in exprCompare().
// Current value 1E-7L is considered to provide a wide margin of error,
// but for our mchron purpose it is accurate enough.
#define EPSILON 1E-7L
```

7.4 Monochron real time clock (RTC) scanning

This section is related to section 7.1, but its information is important enough to warrant a separate one.

In Emuchron, the Linux system clock is scanned every clock cycle, being 75 msec that equals to a ~13.3Hz scan frequency. This results in a very smooth behavior of the seconds indicator in a clock. Using this scan frequency the timespan between two consecutive seconds time events may last up to 1.08 seconds.

In the original Monochron code, the timer interrupt handler that deals with the RTC has been designed such that the RTC scan frequency to generate time events is ~5.7Hz. This means that the timespan between two consecutive seconds time events may last up to 1.18 seconds. This scan frequency is sufficient for the original Monochron Pong clock that does not have a seconds indicator. However for clocks with a seconds indicator, every now and then this results in visually choppy behavior of the seconds indicator by showing an unusually long or short time to switch from one seconds value to the next one.

As this was deemed unacceptable, the timer interrupt handler firmware has been reconfigured such that the RTC scan frequency has been increased to

~8.5Hz. This leads to an acceptable worst-case timespan interval of 1.12 seconds.

The RTC scan frequency is controlled using the following defines in monomain.h [firmware].

```
// Uncomment to implement i2ctime readout @ ~5.7Hz
// #define TIMER2_RETURN_1      80
// #define TIMER2_RETURN_2      6
// Uncomment to implement i2ctime readout @ ~8.5Hz
#define TIMER2_RETURN_1      53
#define TIMER2_RETURN_2      9
```

7.5 The ncurses output appears somewhere else

By default, mchron reads its ncurses tty from file \$HOME/.mchron. The content of this file is created upon starting a Monochron ncurses terminal. For this, refer to section 3.6.2.

What mchron cannot anticipate is the situation where the Monochron terminal is deleted while \$HOME/.mchron still exists, and its tty gets re-used by another bash shell.

Upon starting mchron, it is detected that the tty as read from \$HOME/.mchron is in use and mchron will then redirect ncurses output to that particular shell. The result is that ncurses output will show up in a non-Monochron terminal, and is likely to be incomplete.

Note that the shell to receive ncurses output may even be the one in which mchron is started.

To recover from this, update the information in \$HOME/.mchron by starting a new Monochron terminal and next restart mchron. Another option is to start mchron using the -t flag to manually set the Monochron ncurses tty.

7.6 VirtualBox: mchron OpenGL warnings/failure/coredump

Starting mchron in a Debian 6 VM with the OpenGL2/GLUT LCD device in combination with 3D acceleration may cause OpenGL2 warnings appearing at random places when initializing the OpenGL2/GLUT LCD device. See below.

```
$ ./mchron

*** Welcome to Emuchron command line tool (build Nov 11 2013, 21:00:45) ***
OpenGL Warning: XGetVisualInfo returned 0 visuals for 0x98b2700
OpenGL Warning: Retry with 0xcb returned 1 visuals
OpenGL Warning: XGetVisualInfo returned 0 visuals for 0x98b2700
OpenGL Warning: Retry with 0xcb returned 1 visuals
OpenGL Warning: vboxCall failed with VBox status code -39

OpenGL Warning: vboxCall retry(1) succeeded

mchron PID = 2762

time   : 07:44:33 (hh:mm:ss)
date   : 12/11/2013 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off

Enter 'h' for help.
mchron>
```

Starting mchron in a Debian 7 VM with the OpenGL2/GLUT LCD device in combination with 3D acceleration may fail to start or abort due to a segmentation fault. See example below.

```
$ ./mchron

*** Welcome to Emuchron command line tool (build Nov 30 2013, 10:51:50) ***
OpenGL Warning: XGetVisualInfo returned 0 visuals for 0x93alf08
OpenGL Warning: Retry with 0x8002 returned 0 visuals
Segmentation fault
$
```

Starting mchron in a Debian 8 VM with the OpenGL2/GLUT LCD device in combination with 3D acceleration may fail and may also partially lock the mchron bash terminal. See example below.

```
$ ./mchron

*** Welcome to Emuchron command line tool (build Oct 2 2015, 21:14:45) ***
libGL error: pci id for fd 4: 80ee:beef, driver (null)
OpenGL Warning: glFlushVertexArrayRangeNV not found in mesa table
:
OpenGL Warning: XGetVisualInfo returned 0 visuals for 00007f6ad017f510
OpenGL Warning: Retry with 0x8003 returned 0 visuals
freelut (Monochron (glut)): ERROR: Internal error <visualInfo could not be
retrieved from FBConfig> in function fgOpenWindow
$
```

In general, these issues are prevented by defining a specific OpenGL environment variable in \$HOME/.bashrc, forcing a proper initialization of the OpenGL environment. For this refer to section 3.2.2.

In case mchron fails or crashes, despite the modification in \$HOME/.bashrc, unchecking the 3D acceleration checkbox for the VM has shown to be an effective workaround. For this refer to section 3.2.2.

The drawback of this workaround is that GNOME will not be able to use its graphics potentials to the fullest extent. As a result, the mchron OpenGL2/GLUT device will show a slightly less fluent graphics behavior that still remains very acceptable.

For unlocking a (partially) locked bash terminal refer to section 8.1.

7.7 No command history when using ncurses LCD device

Normally a command shell allows the cursor keys to be used to browse through the command history. Within Linux the standard readline library is available that supports this behavior. Within mchron code is written to tap into this library.

When mchron is used in combination with only the OpenGL2/GLUT LCD device, the mchron command history can be browsed as expected. However, this will not be the case when using the ncurses LCD device.

The reason for this is that, again, the ncurses library is the deal breaker. Next to the fact that ncurses doesn't play nice with gdb, ncurses also doesn't play nice with the readline library.

The combination of the ncurses and the readline libraries results in an mchron terminal that needs to be reset every time after exiting the mchron shell.

For this reason, mchron avoids the use of the readline library when the ncurses LCD device is used. For more information regarding the use of the readline library refer to section 5.7.

7.8 Performance of the mchron interpreter

It turns out that performance is good enough.

To illustrate this, execute either the commands below in mchron or execute script loop.txt [script] that provides the same functionality. Repeat the commands or script a few times to level out runtime differences.

```
mchron> # Do a dummy loop 1 million times
mchron> rf x=0 x<1000000 x=x+1
2>> # Dummy comments
3>> vs y=x+1
4>> rn
mchron>
```

On the Intel based hypervisor VMs that are used to develop and test Emuchron the repeat loop will take about 5 to 8 seconds to complete, depending on available CPU power.

As performance has never been an issue while developing mchron, no out of the ordinary efforts were made to optimize the interpreter code on speed. Instead, focus was put on accuracy, reliability and the prevention of memory leaks.

In case Linux is run in a VM and it takes much longer to complete the test script above, verify that in the BIOS of the host system the CPU has enabled Intel (VT-x) or AMD (AMD-V) Virtualization Technology. For this, refer to section 3.2.1.

7.9 After an mchron coredump there is no coredump file

A coredump will create a coredump file only after executing a one-time only command in the current shell prior to starting mchron: `ulimit -c unlimited`. Refer to section 5.3 for an example.

7.10 There is a delay in starting a stubbed Mario alarm

The audio stub that starts the Mario alarm generates a command consisting of almost 600 piped shells combining all individual Mario tune tones and tune pauses. It turns out it takes Linux about two seconds to start this up which is, considering its highly unusual length and structure, very acceptable.

In Emuchron it means that Mario alarm starts playing a rough two seconds after the alarm is tripped.

In case Linux is run in a VM and it takes much longer to start playing the Mario alarm, for example 25 seconds, verify that in the BIOS of the host system the CPU has enabled Intel (VT-x) or AMD (AMD-V) Virtualization Technology. For this, refer to section 3.2.1.

7.11 Firmware size penalty for new Emuchron functionality

Of course, the additional functionality provided by Emuchron, when added to the original Monochron firmware, will cost data and program space. One may expect that Emuchron, due to its implementation of a generic clock plugin framework, an additional configuration page, an additional font, and enhanced and optimized graphics functions, results in a substantially bigger firmware file when compared to the original Monochron firmware.

This turns out not to be the case. On the contrary, when building the original Monochron firmware and compare its size with Emuchron firmware that only

includes the migrated pong clock and a two-tone alarm, the Emuchron firmware size is over 4 KB smaller, a rough 15%, despite its enhancements. To be fair, the migrated pong clock in Emuchron has slightly reduced functionality, but that is mostly compensated by a much improved gameplay.

In general terms, within Emuchron a lot of data and program space is recovered by removing unused code and data, and optimizing original Monochron and clock code for object code size.

Emuchron firmware aims to keep its object code size small by testing multiple source code solutions for the same functionality. The object size optimized code should not, or only negligible, impact the overall performance, but may have some impact on code readability. It is considered to be an acceptable trade-off.

7.12 Is it required to build firmware on Debian Linux

No.

Only the Emuchron emulator will require Debian Linux to build and run. For building the Monochron firmware however, any machine and operating system can be used that supports an AVR toolchain. For example, if an AVR toolchain is installed on a machine running Windows 7, all that is needed is to copy the project firmware folder onto the machine and follow the build instructions in section 4.1. Refer to section 4.3 on how to upload the firmware to a Monochron clock.

For an actual example refer to appendix B.1 that discusses glcd performance tests. For these performance tests firmware is used that is built on a Fedora distro.

7.13 My debugger cannot find file "syscall-template.S"

This is an annoying Debian 8 'feature'. Although an attempt is made to fix this during the installation of required Debian packages, it may be needed to create a specific symbolic link resolving the missing link in the glibc source path. Instructions for this are found in section 3.7.4.

8 Known bugs

8.1 The mchcron terminal no longer echoes characters

When mchcron executes a command list or a wait command, it switches the terminal input behavior from using a readline input method where text input is to be completed with a newline, to a keypress input method where every keypress is regarded as a separate event. This allows the end-user to issue keypress commands and provides a convenient method to interrupt command or script execution. When command or script execution has completed, mchcron will automatically switch back to the default readline input method.

One of the features of the keypress method is that it will not echo keypress characters in the mchcron terminal.

When mchcron is interrupted or is about to crash, it attempts to clean up the environment and, most importantly, it attempts to switch back the terminal input mode to the readline method. Although great care has been given to make mchcron switch back to the readline method, a full guarantee of this always happening cannot be provided.

When the readline input method is not restored, the mchcron terminal appears to be dead as it no longer echoes keyboard characters. Input characters are buffered though, and when a newline character is entered it will make the un-echoed characters become the shell command to be executed.

To recover from this situation, the end-user can simply kill the current terminal and start a new one. Another option is to type a blind (remember, characters are not echoed) terminal `reset` command that will restore the default terminal behavior settings.

The use of the blindly typed terminal `reset` command turns out to be very effective.

8.2 Pending characters in the mchcron terminal input buffer

As explained in section 8.1, mchcron switches between a readline and keypress input method.

Upon exiting the clock or Monochron emulator (refer to respectively section 5.8.4 and 5.8.10), or completing the execution of a command list (refer to section 5.10), an attempt is made to clear the input buffer from remaining keypresses before control is given back to the mchcron command prompt. This may not always be successful, especially when the end-user holds down a single key, thereby generating multiple repeat keypresses in the input buffer.

Upon returning to readline mode, the buffer may still contain one or more remaining keypress characters in the input buffer that are not echoed, but are taken into account for the next mchcron command.

In case this occurs, the next mchcron command is likely to fail as the remaining input buffer characters are not expected to make up a correct mchcron command.

Note that hitting a keypress one at a time will result in proper keypress processing and will not leave a pending character in the terminal input buffer.

Currently there is no known way to circumvent the erroneous behavior described above.

A Screendumps of example clocks

The ncurses LCD device output screendumps below are taken using a standard Linux window screendump tool. The clocks id's as listed are defined in anim.h [firmware]. For the special performance test clock plugin refer to section 2.9.

How difficult is it to create the clock layouts in this appendix?

- First, we start mchron using the ncurses LCD device.
- Then, five respective mchron commands are used to select the digital HMS clock, set the position of the alarm switch to 'on', to make the clock display the alarm time, set the date to Sep 14th 2013, set the alarm to 06:45, and finally set the time to 22:09:30.
- As the resulting clock layout is static we have all the time to inspect the result and use a screendump tool. The resulting clock layout can be seen in appendix A.3.
- If we now want additional screendumps using the same date and time, just select another clock using command 'cs'.

```
$ ./mchron -l ncurses
:
mchron> cs 3
mchron> ap 1
time : 19:20:15 (hh:mm:ss)
date : 22/07/2013 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : on
mchron> ds 14 9 13
time : 19:20:33 (hh:mm:ss)
date : 14/09/2013 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : on
mchron> as 6 45
time : 19:20:40 (hh:mm:ss)
date : 14/09/2013 (dd/mm/yyyy)
alarm : 06:45 (hh:mm)
alarm : on
mchron> ts 22 9 30
time : 22:09:30 (hh:mm:ss)
date : 14/09/2013 (dd/mm/yyyy)
alarm : 06:45 (hh:mm)
alarm : on
mchron>
```

A.1 Analog clocks

Clock Ids: CHRON_ANALOG_HMS and CHRON_ANALOG_HM

These are basic analog clocks with h/m/s or h/m time notification.

When the alarm switch is on, the alarm time will appear at the bottom left in a small analog clock. When the alarm switch is off, the clock will show the current date. When alarming or snoozing, the alarm time will blink.

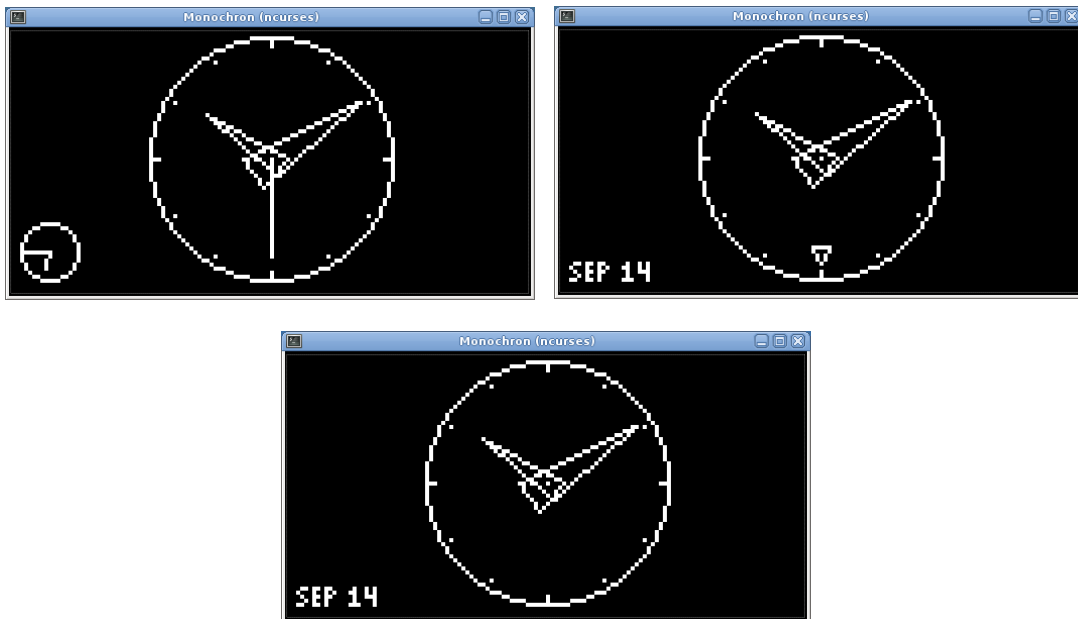
There are several build options for an analog clock, allowing eight different versions of the h/m/s flavor and two versions of the h/m flavor. See below.

```
// Determine the second indicator shape.
// 0 = Needle
// 1 = Floating arrow
#define ANA_SEC_TYPE          1

// Determine how the second indicator moves.
// 0 = Only at a full second stop
// 1 = Whenever the (x,y) position of a leg changes
#define ANA_SEC_MOVE          1

// Determine how the minute arrow moves.
// 0 = Only at a full minute stop
// 1 = Whenever the (x,y) position of the arrow tip changes
#define ANA_MIN_MOVE          1
```

For code refer to analog.c [firmware/clock].



A.2 Big Digit clocks

Clock Ids: `CHRON_BIGDIG_ONE` and `CHRON_BIGDIG_TWO`

These are clocks that display either a single or two digits from the current time and date. On the left and right side of the display the clock shows the available time and date elements, and highlights the one that is currently active. Upon pressing the Set button, or in case only a single clock is configured the '+' button as well, the clock will move to the next time or date element. When the alarm switch is on, the alarm time will appear at the bottom left. When alarming or snoozing, the alarm time will blink. For code refer to `bigdig.c` [firmware/clock].



A.3 Digital clocks

Clock Ids: `CHRON_DIGITAL_HMS` and `CHRON_DIGITAL_HM`

These are basic digital clocks with `hh:mm:ss` or `hh:mm` time notification. When the alarm switch is on, the alarm time will appear at the bottom left. When alarming or snoozing, the alarm time will blink. Note that all the text strings displayed in the screendumps below are, at its lowest level, generated using a single `glcdPutStr3()` function only, being `glcdPutStr3()`. For code refer to `digital.c` [firmware/clock].



A.4 Mosquito clock

Clock Id: `CHRON_MOSQUITO`

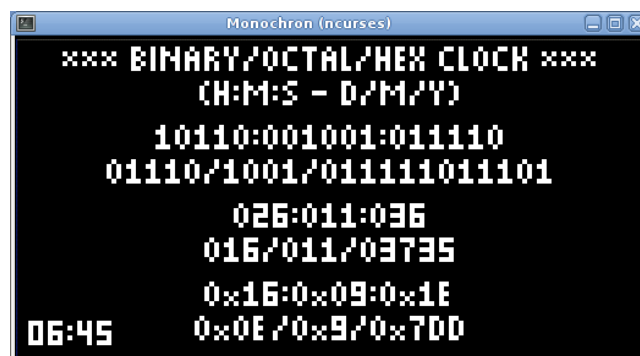
This clock implements the time as separate elements that randomly float over the LCD display. After starting the clock it will initially show the time with static elements. After a few seconds however, first the seconds element will start moving, then the minutes element and finally the hours element as well. Every minute the angle with which an element will move is randomly set. When the alarm switch is on, the alarm time will appear at the bottom left. When the alarm switch is off, it will show the current date. When alarming or snoozing, the alarm time will blink. For code refer to `mosquito.c` [firmware/clock].



A.5 Nerd clock

Clock Id: `CHRON_NERD`

This clock displays the time and date in binary, octal and hexadecimal format. When the alarm switch is on, the alarm time will appear at the bottom left. When alarming or snoozing, the alarm time will blink. For code refer to `nerd.c` [firmware/clock].



A.6 Pong clock

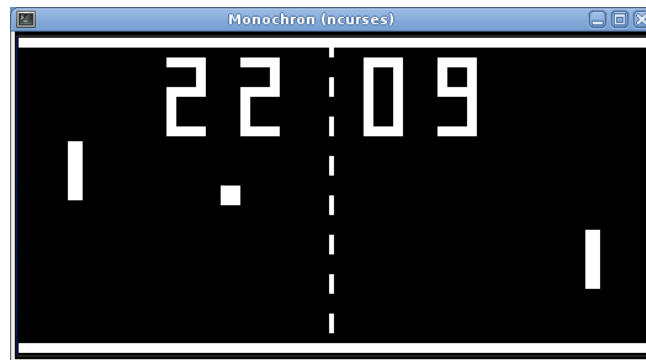
Clock Id: CHRON_PONG

This clock is the original Monochron pong clock, but is migrated to be used in the Emuchron framework. Functionality to process time, date and alarm has been re-implemented to use the Emuchron data environment. The basic migration of the clock code took about one day of efforts.

A number of functional changes have been applied though. Gameplay is much improved by changing the ball motion angle at every paddle bounce instead of only once per minute and by allowing shallow angles. Also, whenever a point is scored, the game is paused for two seconds before resuming. And finally, the built-in random generator is replaced by a much smaller and simpler algorithm, making a significant savings in firmware size.

When the clock is alarming, whereas the original code will inverse the clock layout every second, in the Emuchron framework the alarming state is identified by flashing the center of the paddles.

For code refer to pong.c [firmware/clock].



A.7 Puzzle clock

Clock Id: CHRON_PUZZLE

This clock combines the hour/min/sec time elements and day/mon/year date elements using filled circles.

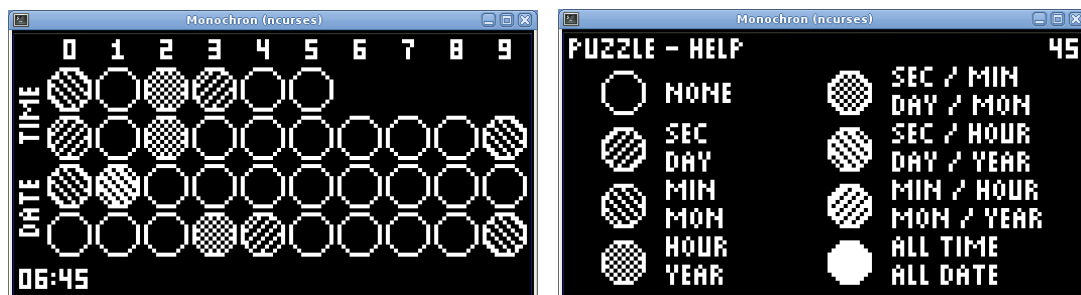
Upon pressing the Set button, or in case only a single clock is configured the '+' button as well, a help page is displayed with a display countdown timer.

Pressing the button again will restore the clock layout.

When the alarm switch is on, the alarm time will appear at the bottom left.

When alarming or snoozing, the alarm time will blink.

For code refer to puzzle.c [firmware/clock].



A.8 QR clocks

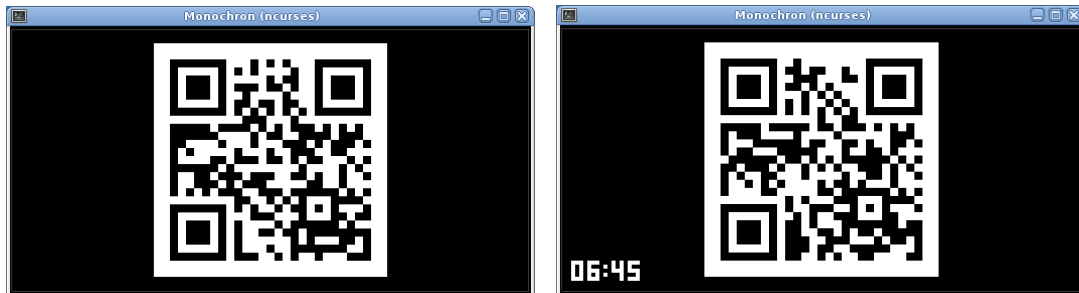
Clock Ids: `CHRON_QR_HMS` and `CHRON_QR_HM`

These clocks encode the date and either h/m/s or h/m into a QR code. The h/m flavor draws a new QR once a minute whereas the h/m/s flavor draws a new QR every second. Use your favorite smartphone QReader app to read the date and time. The clock has a hardcoded Easter egg on April 1st.

When the alarm switch is on, the alarm time will appear at the bottom left.

When alarming or snoozing, the alarm time will blink.

For code refer to `qr.c` and `qrencode.c` [firmware/clock]. The QR encode module uses code from project `qrduino` (<https://github.com/tz1/qrduino>).



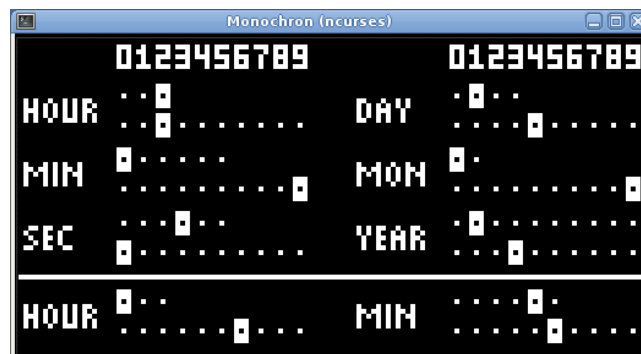
A.9 Slider clock

Clock Id: `CHRON_SLIDER`

This clock displays the time and date using slider elements.

When the alarm switch is on, the alarm time will appear at the bottom using similar slider elements. When alarming or snoozing, the alarm text labels will blink.

For code refer to `slider.c` [firmware/clock].



A.10 QuintusVisuals clocks

Clock Id: CHRON_CASCADE, CHRON_SPEEDDIAL, CHRON_SPIDERPLOT and CHRON_TRAFLIGHT

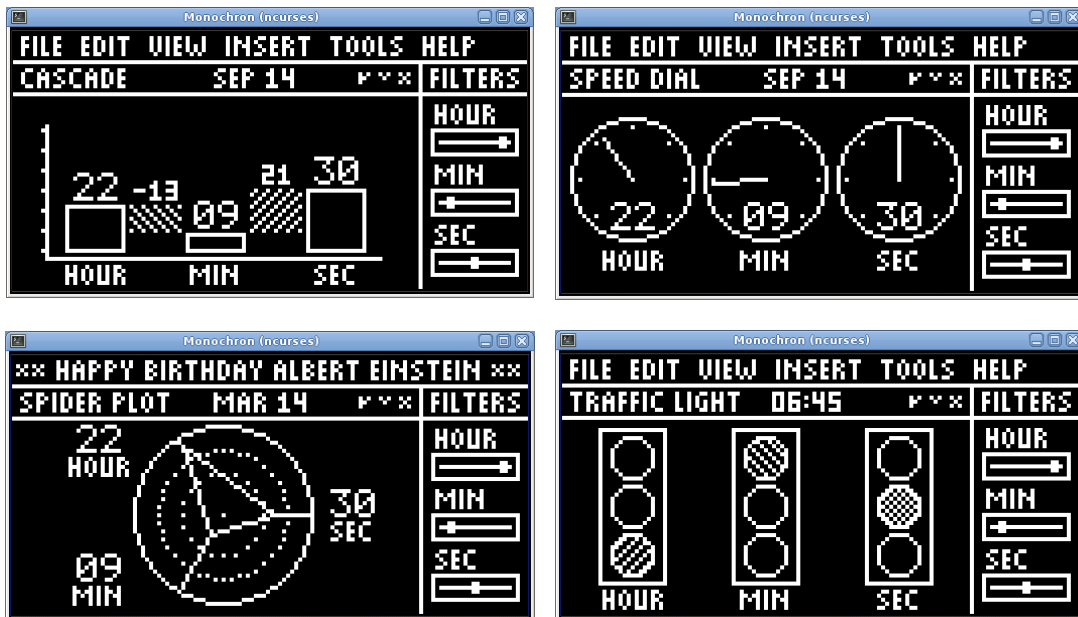
TIBCO Spotfire (<http://spotfire.tibco.com>) is a professional business analytics tool that provides insight in very large amounts of data using visualizations. QuintusVisuals (<http://www.quintusvisuals.com/en/home>) is an extension to TIBCO Spotfire and provides additional visualization types. The clocks below are minimalistic implementations of the QuintusVisuals visualizations showing the time, date and alarm.

The (non-functional) header of a QuintusVisuals visualization represents the header of TIBCO Spotfire. However, the QuintusVisuals clocks include a hard-coded calendar that will change the header on specific dates to a dedicated message. See the spider plot example for March 14th below.

The filter panel on the right side contains sliders for the hour, minutes and seconds elements that are similar to those in TIBCO Spotfire. They will move along as time progresses.

The date will appear in the center of the visualization header. When the alarm switch is on, the alarm time will replace the date at that location. When alarming or snoozing, the alarm time will blink.

For code refer to spotfire.c (generic module for all QuintusVisuals clocks, including the calendar), cascade.c, speeddial.c, spiderplot.c and traffilight.c [firmware/clock].



B High-level glcd performance tests

In Emuchron v1.3 and v2.0 several enhancements are made in high-level glcd functions to increase draw performance. In order to find out whether these modifications actually lead to an improved draw performance, a clock plugin is created that allows running high-level glcd performance tests on Monochron hardware. Some of these tests are written to highlight specific enhancements while some are written specifically to mimic glcd usage in actual Monochron clock code.

The performance clock plugin is originally created in Emuchron v1.3 and is enhanced in Emuchron v2.0.

B.1 Test results Emuchron v1.3 vs v1.2

In order to obtain test results for Emuchron v1.2 the code for the performance test plugin and emulator code for obtaining glcd interface statistics has been ported back to Emuchron v1.2.

The performance clock firmware is built and tested using three versions of avr-gcc. Find below an overview of version and build sizes.

Version avr-gcc	Emuchron v1.3 Object size (bytes)	Emuchron v1.2 Object size (bytes)
4.3.5 (Debian 6)	.data: 870 .text: 25,690 Total: 26,560	.data: 864 .text: 26,066 Total: 26,930
4.7.2 (Debian 7)	.data: 866 .text: 25,052 Total: 25,918	.data: 860 .text: 25,454 Total: 26,314
4.9.2 (Fedora 21)	.data: 866 .text: 24,928 Total: 25,794	.data: 860 .text: 25,316 Total: 26,176

Table 22: Performance test firmware build object size

Some remarks on the build statistics:

- The performance test firmware uses code from the analog clock. The analog module is built with the seconds indicator set to a needle (v1.3-only), a seconds update on a full stop (v1.3-only) and a minute arrow update whenever its tip changes position (both v1.3 and v1.2).
- The total object size of v1.3 builds is smaller than v1.2 builds. This is achieved by code optimizations in Emuchron v1.3 aimed at reducing object size.
- The higher the version of avr-gcc, the smaller the total object size. This can be a combination of the compiler becoming more clever at optimizing on object size, as well as size optimizations in the underlying avr-libc library.
- Smaller object code will not imply faster code execution. For this, see below.

Each test in the test plugin is run in both the emulator (to obtain glcd interface statistics) and on Monochron hardware (to obtain runtime statistics). For more information regarding the glcd dataWrite, dataRead and setAddress indicators refer to section 5.8.13. Note that the numbers provided for dataRead in the table below actually do represent the number of pixel bytes read from the LCD.

Find below a table with the results of the performance tests. The time reported is the time to complete a test in minutes and seconds for each version of avr-gcc.

For details on a specific test refer to the actual code in perftest.c [firmware/clock].

Test	Test Name	Emuchron v1.3	Emuchron v1.2
1	glcdCircle2-1	Time 4.3.5: 2:00 4.7.2: 2:09 4.9.2: 1:58 dataWrite: 467,526 dataRead: 500,032 setAddress: 967,558	Time 4.3.5: 2:04 4.7.2: 2:13 4.9.2: 2:02 dataWrite: 500,032 dataRead: 500,032 setAddress: 1,000,064
2	glcdCircle2-2	Time 4.3.5: 2:00 4.7.2: 2:09 4.9.2: 1:59 dataWrite: 453,600 dataRead: 518,400 setAddress: 972,000	Time 4.3.5: 2:08 4.7.2: 2:18 4.9.2: 2:07 dataWrite: 518,400 dataRead: 518,400 setAddress: 1,036,800
3	glcdDot-01	Time 4.3.5: 2:01 4.7.2: 2:11 4.9.2: 1:59 dataWrite: 491,520 dataRead: 491,520 setAddress: 983,040	Time 4.3.5: 2:01 4.7.2: 2:11 4.9.2: 1:59 dataWrite: 491,520 dataRead: 491,520 setAddress: 983,040
4	glcdDot-02	Time 4.3.5: 2:01 4.7.2: 2:10 4.9.2: 2:00 dataWrite: 319,488 dataRead: 638,976 setAddress: 958,464	Time 4.3.5: 2:37 4.7.2: 2:50 4.9.2: 2:35 dataWrite: 638,976 dataRead: 683,976 setAddress: 1,277,952
5	glcdLine-01	Time 4.3.5: 2:04 4.7.2: 2:13 4.9.2: 2:03 dataWrite: 369,369 dataRead: 557,523 setAddress: 926,870	Time 4.3.5: 2:25 4.7.2: 2:25 4.9.2: 2:24 dataWrite: 552,008 dataRead: 552,008 setAddress: 1,103,994
6	glcdFillCircle2-01	Time 4.3.5: 2:01 4.7.2: 2:08 4.9.2: 2:00 dataWrite: 571,136 dataRead: 237,728 setAddress: 808,864	Time 4.3.5: 2:29 4.7.2: 2:36 4.9.2: 2:27 dataWrite: 571,136 dataRead: 237,728 setAddress: 808,864
7	glcdFillRectangle2-01	Time 4.3.5: 1:59 4.7.2: 2:07 4.9.2: 1:58 dataWrite: 668,850 dataRead: 668,850 setAddress: 758,030	Time 4.3.5: 2:54 4.7.2: 2:59 4.9.2: 2:51 dataWrite: 668,850 dataRead: 668,850 setAddress: 758,030
8	glcdFillRectangle2-02	Time 4.3.5: 2:00 4.7.2: 2:09 4.9.2: 2:01 dataWrite: 1,595,000 dataRead: 580,000 setAddress: 611,900	Time 4.3.5: 4:05 4.7.2: 4:06 4.9.2: 4:00 dataWrite: 1,595,000 dataRead: 580,000 setAddress: 611,900
9	glcdFillRectangle2-03	Time 4.3.5: 2:00 4.7.2: 2:08 4.9.2: 2:01 dataWrite: 2,000,880 dataRead: 500,220 setAddress: 516,100	Time 4.3.5: 4:20 4.7.2: 4:20 4.9.2: 4:14 dataWrite: 2,000,880 dataRead: 500,220 setAddress: 516,100

Test	Test Name	Emuchron v1.3	Emuchron v1.2
10	glcdFillRectangle2-04	Time 4.3.5: 2:00 4.7.2: 2:08 4.9.2: 2:01 dataWrite: 1,968,624 dataRead: 492,156 setAddress: 507,780	Time 4.3.5: 4:31 4.7.2: 4:30 4.9.2: 4:24 dataWrite: 1,968,624 dataRead: 492,156 setAddress: 507,780
11	glcdPutStr3-01	Time 4.3.5: 2:00 4.7.2: 2:09 4.9.2: 2:00 dataWrite: 723,240 dataRead: 723,240 setAddress: 728,980	Time 4.3.5: 2:02 4.7.2: 2:09 4.9.2: 2:02 dataWrite: 723,240 dataRead: 723,240 setAddress: 728,980
12	glcdPutStr3-02	Time 4.3.5: 1:59 4.7.2: 2:06 4.9.2: 1:59 dataWrite: 1,219,680 dataRead: 609,840 setAddress: 619,520	Time 4.3.5: 2:06 4.7.2: 2:12 4.9.2: 2:05 dataWrite: 1,219,680 dataRead: 609,840 setAddress: 619,520
13	glcdPutStr3-03	Time 4.3.5: 1:59 4.7.2: 2:08 4.9.2: 2:00 dataWrite: 706,800 dataRead: 706,800 setAddress: 712,500	Time 4.3.5: 2:01 4.7.2: 2:08 4.9.2: 2:00 dataWrite: 706,800 dataRead: 706,800 setAddress: 712,500
14	glcdPutStr3-04	Time 4.3.5: 2:01 4.7.2: 2:08 4.9.2: 2:01 dataWrite: 1,092,000 dataRead: 655,200 setAddress: 664,300	Time 4.3.5: 2:06 4.7.2: 2:13 4.9.2: 2:06 dataWrite: 1,092,000 dataRead: 655,200 setAddress: 664,300
15	glcdPutStr3-05	Time 4.3.5: 2:00 4.7.2: 2:08 4.9.2: 2:00 dataWrite: 732,816 dataRead: 732,816 setAddress: 738,632	Time 4.3.5: 2:09 4.7.2: 2:16 4.9.2: 2:06 dataWrite: 732,816 dataRead: 732,816 setAddress: 738,632
16	glcdPutStr3v-01	Time 4.3.5: 2:00 4.7.2: 2:16 4.9.2: 2:07 dataWrite: 544,320 dataRead: 136,080 setAddress: 244,944	Time 4.3.5: 7:09 4.7.2: 7:27 4.9.2: 6:56 dataWrite: 1,496,880 dataRead: 1,496,880 setAddress: 2,993,760
17	glcdPutStr3v-02	Time 4.3.5: 2:00 4.7.2: 2:13 4.9.2: 2:06 dataWrite: 1,364,832 dataRead: 341,208 setAddress: 406,200	Time 4.3.5: 9:48 4.7.2: 10:22 4.9.2: 9:35 dataWrite: 2,047,248 dataRead: 2,047,248 setAddress: 4,094,496
18	glcdPutStr3v-03	Time 4.3.5: 2:01 4.7.2: 2:19 4.9.2: 2:10 dataWrite: 646,016 dataRead: 161,504 setAddress: 253,792	Time 4.3.5: 5:40 4.7.2: 5:59 4.9.2: 5:30 dataWrite: 1,130,528 dataRead: 1,130,528 setAddress: 2,261,056

Test	Test Name	Emuchron v1.3	Emuchron v1.2
19	glcdPutStr3v-04	Time 4.3.5: 2:00 4.7.2: 2:12 4.9.2: 2:05 dataWrite: 1,266,720 dataRead: 316,680 setAddress: 401,128	Time 4.3.5: 10:25 4.7.2: 10:54 4.9.2: 10:10 dataWrite: 2,216,760 dataRead: 2,216,760 setAddress: 4,433,520

Table 23: Performance test runs

Some remarks on the test results:

- Tests 1 thru 5 are based on a change in `glcdDot()` that no longer writes back an LCD byte when nothing has changed, as evidenced by the lower numbers for `dataWrite` and `setAddress` between v1.2 and v1.3. As a result, the modest performance improvements in v1.3 are the result of optimizing LCD cursor placement and LCD write operations.
- Test 5 shows a higher `dataRead` number for v1.3 than in v1.2. This is mainly caused by the fact that the v1.3 analog clock code uses a seconds needle that is 1 pixel longer than the analog clock code in v1.2. This means that in v1.3 more pixels will be read from the LCD.
- Test 6 thru 10 are based on a full rewrite of `glcdFillRectangle2()`. Notice that between v1.2 and v1.3 the numbers for `dataRead`, `dataWrite` and `setAddress` are unchanged. This means that the interaction logic with the LCD remained untouched. The significant performance gains in v1.3 is therefore fully attributed to more efficient code in `glcdFillRectangle2()`. To be more specific, the use of the CPU intensive modulo function (%) has been cut down drastically (improving efficiency of the `FILL_HALF`, `FILL_THIRDDUP` and `FILL_THIRDDOWN` fill types), and LCD bytes are now always processed on a per byte basis instead of, in specific cases, on a per bit basis (improving efficiency of all fill types).
- Test 11 thru 15 are based on optimizations in `glcdPutStr3()`, and aligning its code with the new body for `glcdPutStr3v()`. Notice that between v1.2 and v1.3 the numbers for `dataRead`, `dataWrite` and `setAddress` are unchanged. This means that the interaction logic with the LCD remained untouched.
Test 11 and 13 show hardly any performance gain for non-scaled text that crosses LCD y-byte lines.
Test 12 and 14 show modest improvements when font scaling is applied for both supported fonts.
Test 15 shows a higher performance gain for non-scaled text that does not cross an LCD y-byte line. The draw code has specifically been optimized for this situation.
- Test 16 thru 19 are based on a full rewrite of `glcdPutStr3v()`, showing a vast increase in draw performance. The new `glcdPutStr3v()` now uses the `glcdLine[]` buffer, and instead of drawing one character at a time it now draws a string per LCD y-byte line, thereby fully optimizing interaction with the LCD.
The optimized interaction with the LCD is evidenced by the numbers for `glcd` write/read/address operations. In v1.3 the usage of `dataWrite` has been decreased between 33%-64%, `dataRead` usage has been decreased between 83%-91%, and `setAddress` usage has been decreased between 89%-92%.

Some remarks on the several avr-gcc versions:

- In almost every test object code from avr-gcc 4.7.2 runs consistently and significantly slower than the two other avr-gcc versions. Especially when

compared to the older 4.3.5 this is surprising. So, while 4.7.2 is better at optimizing object code than 4.3.5 (refer to table 22 above) it does not do a good job in creating efficient object code. This may be caused by either the compiler or less optimized code in the avr-libc library.

- In v1.3 tests that focus on the changes in `glcdDot()`, 4.9.2 tends to run only slightly faster than 4.3.5.

In v1.3 tests that focus on the changes in `glcdFillRectangle2()` and `glcdPutStr3()` both 4.3.5 and 4.9.2 versions show near identical execution time.

In v1.3 tests that focus on the changes in `glcdPutStr3v()` however, 4.3.5 is surprisingly and consistently faster than 4.9.2. It is unclear what may be causing this.

So, despite the fact that avr-gcc 4.3.5 generates the largest object code size by a wide margin, its resulting object code is also (arguably) the most efficient in execution time.

B.2 Test results Emuchron v2.0 vs v1.3

In Emuchron v2.0 two performance tests are added, being test 6 and 8. These tests have been ported back to Emuchron v1.3.

The performance clock firmware is built and tested using two versions of avr-gcc. Find below an overview of version and build sizes.

Version avr-gcc	Emuchron v2.0 Object size (bytes)	Emuchron v1.3 Object size (bytes)
4.3.5 (Debian 6)	.data: 870 .text: 26,492 Total: 27,362	.data: 870 .text: 26,186 Total: 27,056
4.8.1 (Debian 8)	.data: 866 .text: 25,656 Total: 26,522	.data: 866 .text: 25,386 Total: 26,234

Table 24: Performance test firmware build object size

Some remarks on the build statistics:

- The performance test firmware uses code from the analog clock. The analog module is built using the default settings in the Monochron base code. These settings differ from those used in the performance test v1.3 vs v1.2.
- The firmware size of the v2.0 build is larger than the v1.3 build. This is caused by the draw optimizations in the `glcd.c` [firmware] module that result in a substantial larger module object file. However, much object space is reclaimed by specific code optimizations aimed at reducing the firmware file size. The net effect of both result in a v2.0 firmware file that has a reduced impact in size when compared to the v1.3 firmware file.
- The higher the version of avr-gcc, the smaller the total object size. This can be a combination of the compiler becoming more clever at optimizing on object size, as well as size optimizations in the underlying avr-libc library.
- Smaller object code will not imply faster code execution. For this, see below.

Each test in the test plugin is run in both the emulator (to obtain glcd interface statistics) and on Monochron hardware (to obtain runtime statistics). For more information regarding the `glcd` `dataWrite`, `dataRead` and `setAddress` indicators refer to section 5.8.13.

The optimizations implemented in Emuchron v2.0 are threefold:

- The entire code base makes consistent use of the 'C' keyword `static`. An early v2.0 performance test that was run prior to other optimizations resulted for most tests in an improved draw performance that averaged to about 6%, with a few notable exceptions that showed improvements up to 10%. Although web resources explain that proper use of the `static` keyword will contribute in creating smaller and more efficient object code, an average performance improvement of 6% is considered to be very high, let alone the observed 10% in a single test. It is unclear what is causing this specific performance improvement.
- Graphics draw optimizations were implemented in functions `glcdCircle2()`, `glcdFillCircle2()` and `glcdLine()`. With respect to `glcdCircle2()` and `glcdLine()`, both functions are no longer a wrapper on `glcdDot()`, but instead are now based on a method that uses the `lcdLine[]` buffer that is more complicated to implement yet more efficient at runtime. Code for `glcdFillCircle2()` now merges multiple calls to `glcdFillRectangle2()` into a single one while also preventing redundant calls to the same function.
- Code was implemented in graphics support function `glcdBufferRead()` that optimizes on reading a row of LCD pixel bytes in the `lcdLine[]` buffer. This optimization will benefit all graphics functions that make use of this buffer. Graphics functions that rely on the `lcdLine[]` buffer will see a significant reduction in calls to `dataRead` and especially `setAddress`.

Find below a table with the results of the performance tests. The time reported is the time to complete a test in minutes and seconds for each version of `avr-gcc`.

For details on a specific test refer to the actual code in `perfctest.c` [firmware/clock].

Test	Test Name	Emuchron v2.0	Emuchron v1.3
1	glcdCircle2-1	Time 4.3.5: 1:20 4.8.1: 1:15 dataWrite: 506,414 dataRead: 594,420 setAddress: 154,590	Time 4.3.5: 2:00 4.8.1: 2:08 dataWrite: 467,526 dataRead: 1,000,064 setAddress: 967,558
2	glcdCircle2-2	Time 4.3.5: 0:50 4.8.1: 0:49 dataWrite: 362,880 dataRead: 440,640 setAddress: 155,520	Time 4.3.5: 2:00 4.8.1: 2:08 dataWrite: 453,600 dataRead: 1,036,800 setAddress: 972,000
3	glcdDot-01	Time 4.3.5: 1:54 4.8.1: 2:02 dataWrite: 491,520 dataRead: 983,040 setAddress: 983,040	Time 4.3.5: 2:01 4.8.1: 2:09 dataWrite: 491,520 dataRead: 983,040 setAddress: 983,040
4	glcdDot-02	Time 4.3.5: 1:54 4.8.1: 2:03 dataWrite: 319,488 dataRead: 1,277,952 setAddress: 958,464	Time 4.3.5: 2:01 4.8.1: 2:09 dataWrite: 319,488 dataRead: 1,277,952 setAddress: 958,464
5	glcdLine-01	Time 4.3.5: 0:43 4.8.1: 0:44 dataWrite: 236,850 dataRead: 445,456 setAddress: 135,474	Time 4.3.5: 1:56 4.8.1: 2:03 dataWrite: 345,245 dataRead: 1,037,262 setAddress: 863,854

Test	Test Name	Emuchron v2.0	Emuchron v1.3
6	glcdLine-02	Time 4.3.5: 0:45 4.8.1: 0:45 dataWrite: 367,568 dataRead: 430,000 setAddress: 111,712	Time 4.3.5: 2:09 4.8.1: 2:18 dataWrite: 491,952 dataRead: 983,904 setAddress: 983,904
7	glcdFillCircle2-01	Time 4.3.5: 0:58 4.8.1: 1:00 dataWrite: 464,784 dataRead: 271,768 setAddress: 343,344	Time 4.3.5: 2:01 4.8.1: 2:09 dataWrite: 571,136 dataRead: 475,456 setAddress: 808,864
8	glcdFillCircle2-02	Time 4.3.5: 0:59 4.8.1: 1:02 dataWrite: 324,000 dataRead: 444,000 setAddress: 342,000	Time 4.3.5: 2:00 4.8.1: 2:07 dataWrite: 408,000 dataRead: 720,000 setAddress: 768,000
9	glcdFillRectangle2-01	Time 4.3.5: 1:00 4.8.1: 1:01 dataWrite: 668,850 dataRead: 764,400 setAddress: 178,360	Time 4.3.5: 1:59 4.8.1: 2:06 dataWrite: 668,850 dataRead: 1,337,700 setAddress: 758,030
10	glcdFillRectangle2-02	Time 4.3.5: 1:04 4.8.1: 1:02 dataWrite: 1,595,000 dataRead: 597,400 setAddress: 43,500	Time 4.3.5: 2:00 4.8.1: 2:08 dataWrite: 1,595,000 dataRead: 1,160,000 setAddress: 611,900
11	glcdFillRectangle2-03	Time 4.3.5: 1:11 4.8.1: 1:10 dataWrite: 2,000,880 dataRead: 508,160 setAddress: 19,850	Time 4.3.5: 2:00 4.8.1: 2:07 dataWrite: 2,000,880 dataRead: 1,000,440 setAddress: 516,100
12	glcdFillRectangle2-04	Time 4.3.5: 1:12 4.8.1: 1:10 dataWrite: 1,968,624 dataRead: 499,968 setAddress: 19,530	Time 4.3.5: 2:00 4.8.1: 2:06 dataWrite: 1,968,624 dataRead: 984,312 setAddress: 507,780
13	glcdPutStr3-01	Time 4.3.5: 0:47 4.8.1: 0:47 dataWrite: 723,240 dataRead: 734,720 setAddress: 11,480	Time 4.3.5: 2:00 4.8.1: 2:07 dataWrite: 723,240 dataRead: 1,446,480 setAddress: 728,980
14	glcdPutStr3-02	Time 4.3.5: 0:58 4.8.1: 0:58 dataWrite: 1,219,680 dataRead: 619,520 setAddress: 14,520	Time 4.3.5: 1:59 4.8.1: 2:07 dataWrite: 1,219,680 dataRead: 1,219,680 setAddress: 619,520
15	glcdPutStr3-03	Time 4.3.5: 0:47 4.8.1: 0:47 dataWrite: 706,800 dataRead: 718,200 setAddress: 11,400	Time 4.3.5: 1:59 4.8.1: 2:07 dataWrite: 706,800 dataRead: 1,413,600 setAddress: 712,500
16	glcdPutStr3-04	Time 4.3.5: 0:54 4.8.1: 0:53 dataWrite: 1,092,000 dataRead: 666,120 setAddress: 14,560	Time 4.3.5: 2:01 4.8.1: 2:08 dataWrite: 1,092,000 dataRead: 1,310,400 setAddress: 664,300

Test	Test Name	Emuchron v2.0	Emuchron v1.3
17	glcdPutStr3-05	Time 4.3.5: 0:46 4.8.1: 0:45 dataWrite: 732,816 dataRead: 744,448 setAddress: 11,632	Time 4.3.5: 2:00 4.8.1: 2:07 dataWrite: 732,816 dataRead: 1,465,632 setAddress: 738,632
18	glcdPutStr3v-01	Time 4.3.5: 1:39 4.8.1: 1:50 dataWrite: 544,320 dataRead: 164,592 setAddress: 136,080	Time 4.3.5: 2:00 4.8.1: 2:17 dataWrite: 544,320 dataRead: 272,160 setAddress: 244,944
19	glcdPutStr3v-02	Time 4.3.5: 1:26 4.8.1: 1:34 dataWrite: 1,364,832 dataRead: 357,456 setAddress: 81,240	Time 4.3.5: 2:00 4.8.1: 2:15 dataWrite: 1,364,832 dataRead: 682,416 setAddress: 406,200
20	glcdPutStr3v-03	Time 4.3.5: 1:43 4.8.1: 1:59 dataWrite: 646,016 dataRead: 184,576 setAddress: 115,360	Time 4.3.5: 2:01 4.8.1: 2:20 dataWrite: 646,016 dataRead: 323,008 setAddress: 253,792
21	glcdPutStr3v-04	Time 4.3.5: 1:24 4.8.1: 1:31 dataWrite: 1,266,720 dataRead: 340,808 setAddress: 105,560	Time 4.3.5: 2:00 4.8.1: 2:12 dataWrite: 1,266,720 dataRead: 633,360 setAddress: 401,128

Table 25: Performance test runs

Some remarks on the test results:

- In v1.3, function `glcdCircle2()` is basically a wrapper on `glcdDot()`. In v2.0 however, code for `glcdCircle2()` is now using the `lcdLine[]` buffer. The logic implemented in the code optimization should lead to a higher performance increase when drawing a smaller circle and drawing a circle of type FULL. Both are confirmed by the actual numbers for test 1 and test 2. Note that the optimized code for `glcdCircle2()` in v2.0, when compared to v1.3, shows a significant reduction in the number of calls to `dataRead` and especially `setAddress`. Also note that for test 1 the number of calls to `dataWrite` has actually increased, which is explained by the fact that in v1.3 circles of type HALF and THIRD require less dots to draw.
- In v1.3, function `glcdLine()` is basically a wrapper on `glcdDot()`. In v2.0 however, code for `glcdLine()` is now using the `lcdLine[]` buffer. Note that the optimized code for `glcdLine()` in v2.0, when compared to v1.3, shows a substantial reduction in the number of calls to `dataWrite`, `dataRead` and especially `setAddress`.
- Function `glcdFillCircle2()` is basically a wrapper on function `glcdFillRectangle2()`. In v2.0 code, `glcdFillCircle2()` optimizes the use of `glcdFillRectangle2()`. Note that the optimized code for `glcdFillCircle2()` in v2.0, when compared to v1.3, shows a substantial reduction in the number of calls to `dataWrite`, `dataRead` and `setAddress`.
- The performance increase in all `glcdFillRectangle2()` and `glcdPutStr3()` tests is almost completely attributed to the optimization in `glcdBufferRead()`. Note the substantial reduction in the number of calls to `dataRead` and especially `setAddress`.
- Compared to the draw performance increase observed in `glcdPutStr3()` tests, the results for the `glcdPutStr3v()` tests appear lower than expected.

The relatively low performance gain is explained by the fact that `glcdPutStr3v()` requires less use of the `lcdLine[]` buffer, thereby not profiting that much from the optimization in `glcdBufferRead()`.

Some remarks on the several avr-gcc versions:

- The consistent performance increase of 6% achieved in v2.0 for avr-gcc 4.3.5 by using the 'C' keyword `static` was seen in avr-gcc 4.8.1 as well.
- Similar to the performance test that was run for comparing v1.3 and v1.2, avr-gcc 4.3.5 generates larger object code than 4.8.1. However, 4.3.1 still outperforms 4.8.1 in quite a few tests, especially those for `glcdPutStr3v()`.

C Setting up a Monochron terminal profile

In order to be able to use an ncurses terminal in Monochron as an LCD device it is required to create a specific terminal profile. This is a one-time only configuration action. Below are the steps described to create such a terminal profile in respectively Debian 6+7 and Debian 8.

C.1 Setting up a terminal profile in Debian 6 and 7

- Start a terminal and select "Edit→Profiles...".

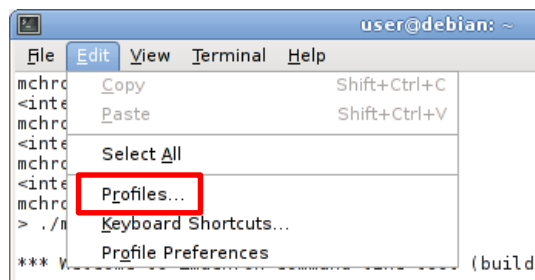


Figure 13: Access terminal profiles

- In the new window that pops up click the 'New' button to create a new profile. See below.

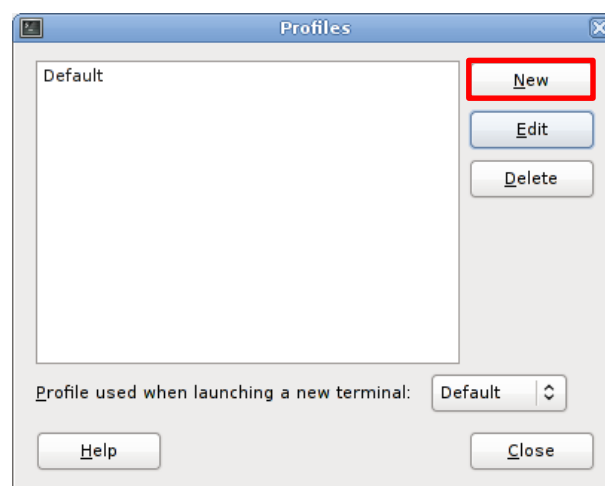


Figure 14: Create new terminal profile

- Name the new profile "Monochron" and select to base it on the 'Default' profile. Click 'Create' to continue.



Figure 15: Create profile 'Monochron'

Now a form is opened with several tabs.

Per tab set the options **exactly** as per screendump and info below.

- Tab 'General'.

Note: The font is 'Monospace' with point size 2. See below. The combination of the font and very small point size allows creating square pixels with a proper size.

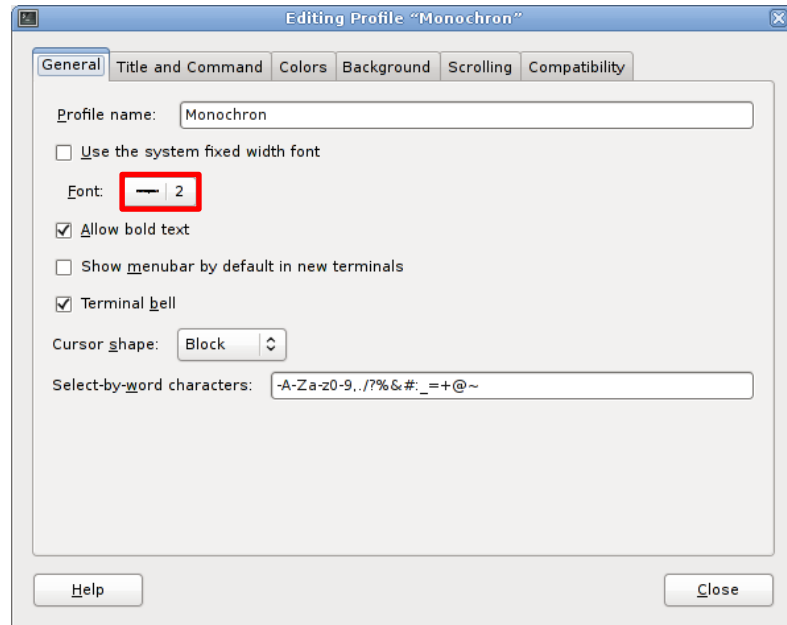


Figure 16: Terminal profile tab 'General'

- Tab 'Title and Command'.

Note: Set the initial title to 'Monochron (ncurses)'. See below.

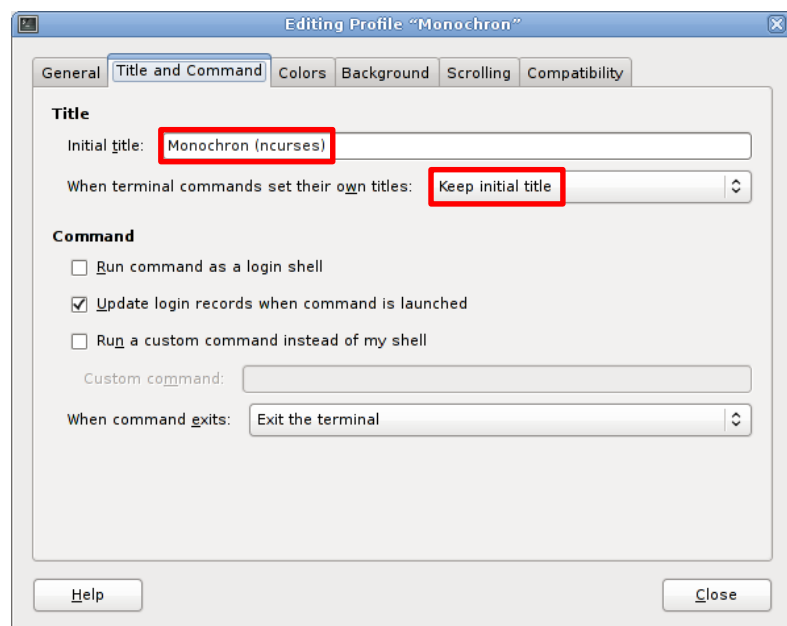


Figure 17: Terminal profile tab 'Title and Command'

- Tab 'Colors'.

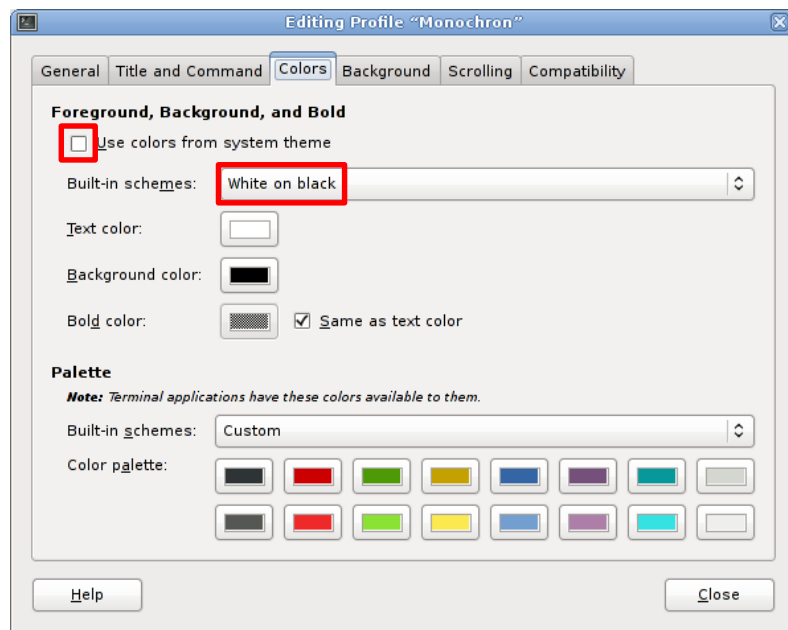


Figure 18: Terminal profile tab 'Colors'

- Tab 'Background'.

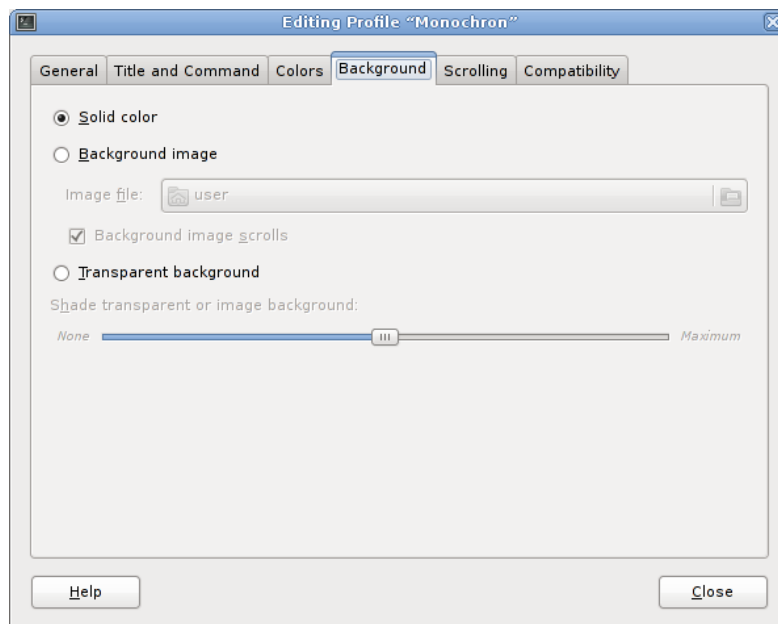


Figure 19: Terminal profile tab 'Background'

- Tab 'Scrolling'.

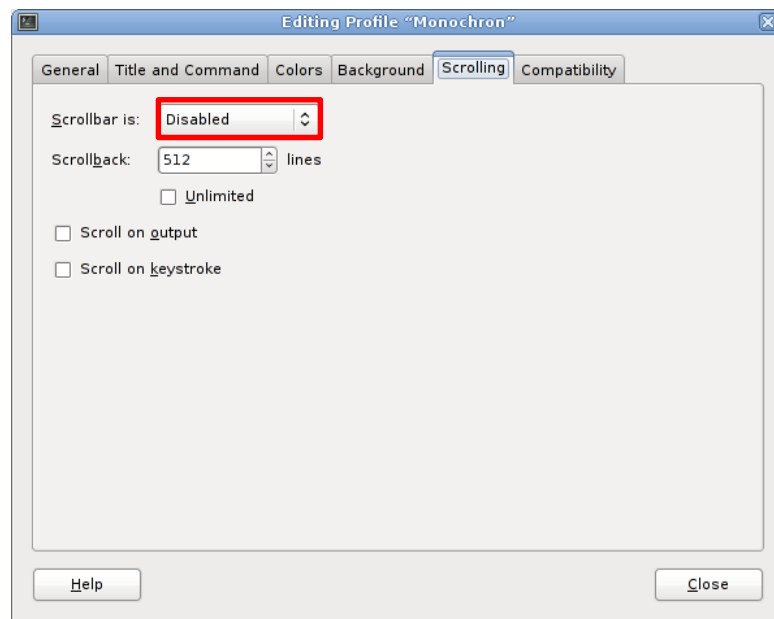


Figure 20: Terminal profile tab 'Scrolling'

- Tab 'Compatibility'.

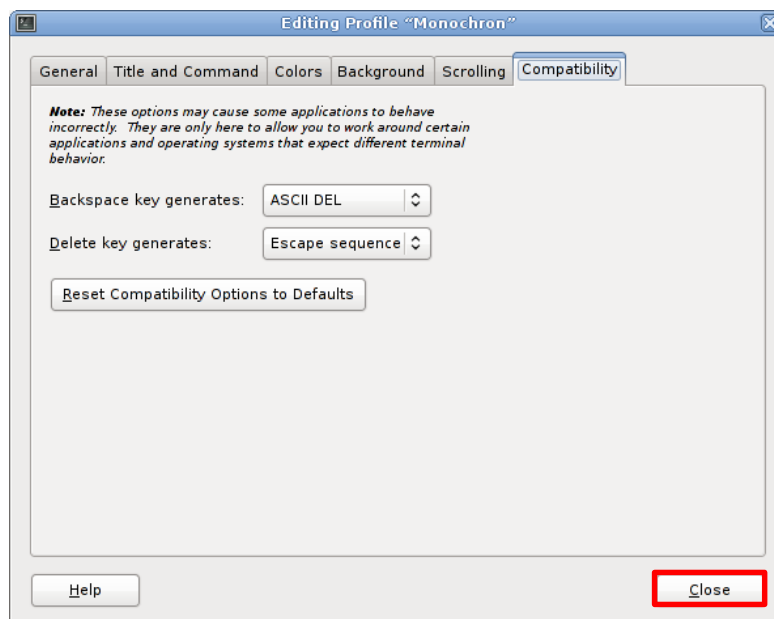


Figure 21: Terminal profile tab 'Compatibility'

As a final step click 'Close' to complete the setup of the Monochron terminal profile.

C.2 Setting up a terminal profile in Debian 8

- Start a terminal and select "Edit→Preferences".

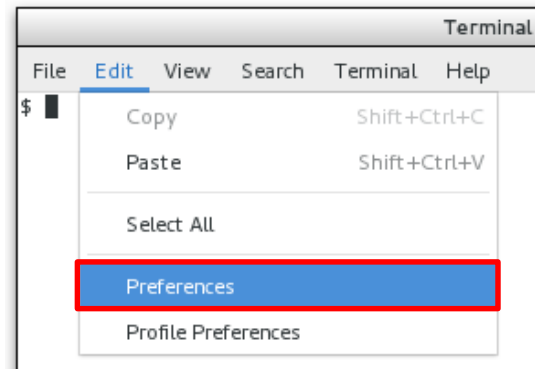


Figure 22: Access terminal profiles

- In the new window that pops up go to tab 'Profiles' and click the 'Clone' button to create a new profile based on the default. See below.

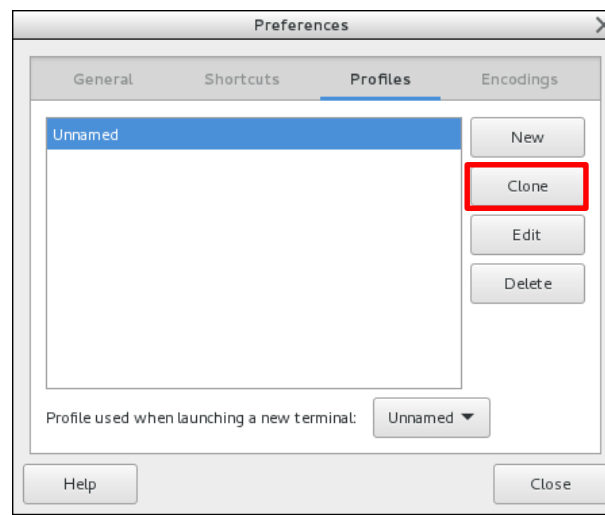


Figure 23: Create new terminal profile

Now a form is opened with several tabs.

Per tab set the options **exactly** as per screendump and info below.

- Tab 'General'.
Name the profile 'Monochron'.
Note: At the bottom select font 'Monospace Regular' with point size 2. The combination of the font and very small point size allows creating square pixels with a proper size.

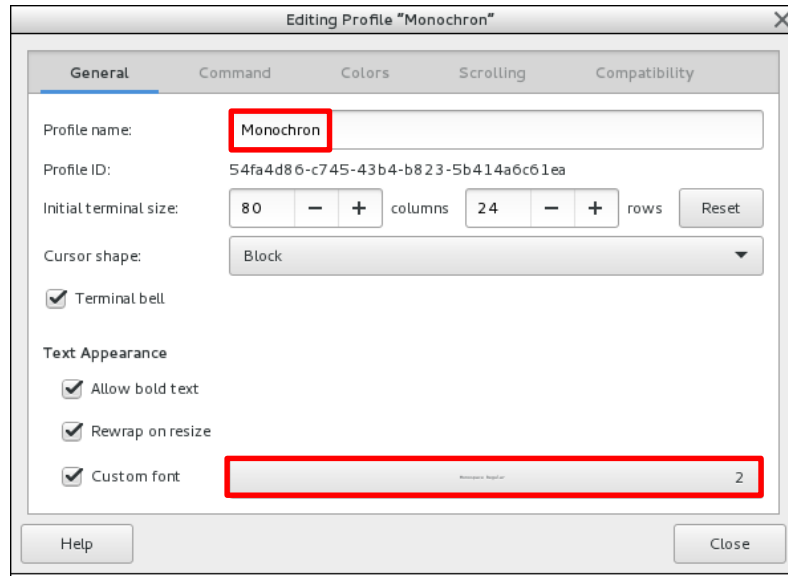


Figure 24: Terminal profile tab 'General'

- Tab 'Command'.

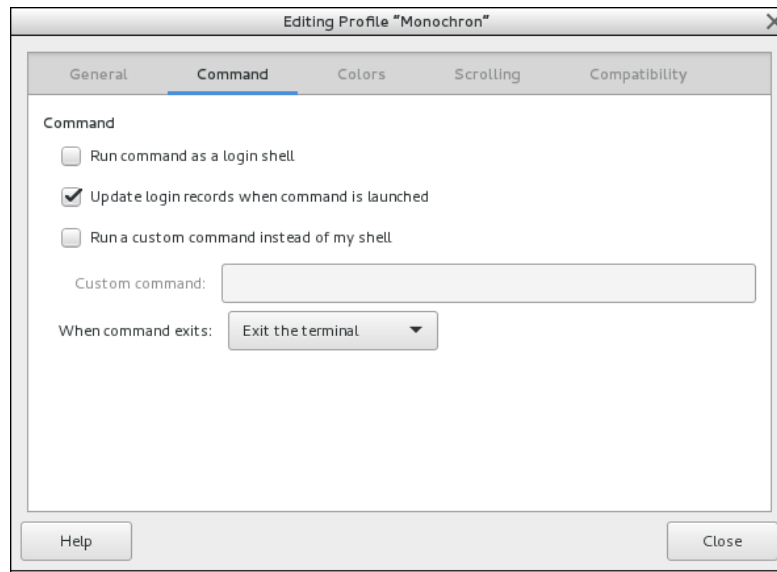


Figure 25: Terminal profile tab ' Command'

- Tab 'Colors'.

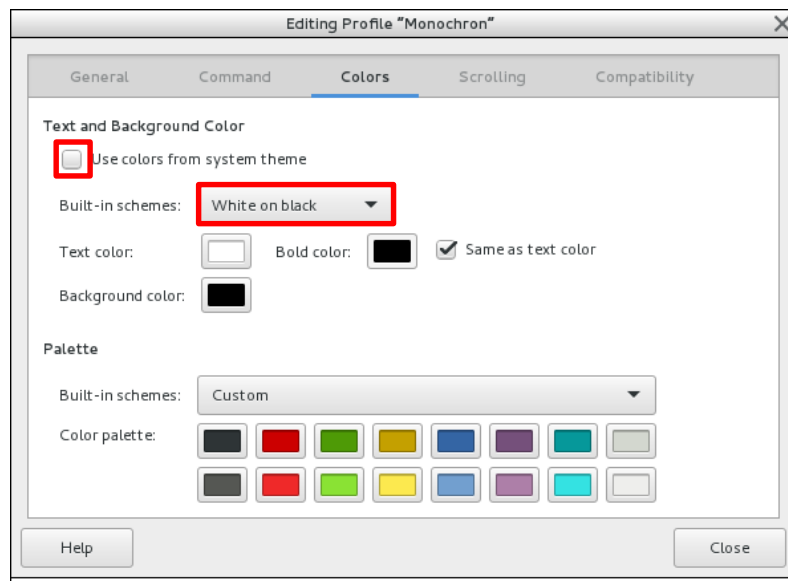


Figure 26: Terminal profile tab 'Colors'

- Tab 'Scrolling'.

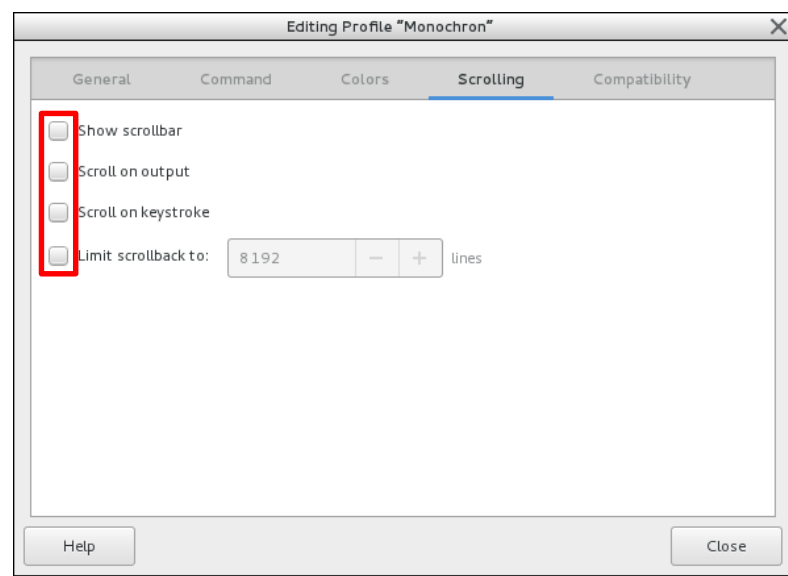


Figure 27: Terminal profile tab 'Scrolling'

- Tab 'Compatibility'.

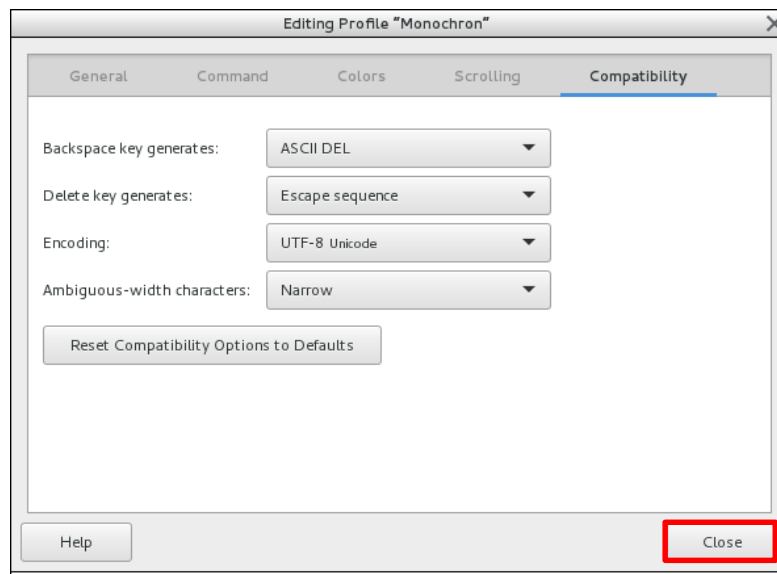


Figure 28: Terminal profile tab 'Compatibility'

As a final step click 'Close' to complete the setup of the Monochron terminal profile.