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

# Erase lcd display
le

# Set variables for 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*factor hor*factor 0
  pl f hor ver*factor hor-(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: v4.1
Date: 29 June 2018

This page is intentionally left blank

Disclaimer

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 qrduino.
  https://github.com/tz1/qrduino

Version history

<table>
<tr><th>Version (date)
Author</th><th>Description</th></tr>
<tr><td>v4.1
(2018-07-01)
T. Ceulemans</td><td>Emuchron emulator code base:
– Alarm audio for Mario or two-tone alarm is now created at mchron build time, using genalarm.c [firmware/emulator], instead of mchron runtime. This makes alarm audio start playing instantly when the alarm is triggered or is resumed from snoozing.
– The command argument scanning, parsing and publishing workflow is restructured and optimized, resulting in cleaner code and improved command list execution performance.
– The command dictionary functions are moved to dictutil.c [firmware/emulator].
– Command 'hm' now accepts a blank message.
– Improved OpenGL2/GLUT stability when VM 3D acceleration is enabled.
– Improved audio stability for short audio pulses.

Monochron firmware code base:
– Optimized <code>glcdPutStr3v()</code>.

Relevant bug fixes:
– None.

Generic:
– Refinements in Emuchron and minor bug fixing.
– Latest documentation.</td></tr>
<tr><td>v4.0.2
(2017-12-23)
T. Ceulemans</td><td>Relevant bug fixes:
– Corrected mchron emulator build error when configured for two-tone alarm.</td></tr>
<tr><td>v4.0.1
(2017-12-13)
T. Ceulemans</td><td>Relevant bug fixes:
– Incorrect definition of string length define <code>AD_AREA_AD_WIDTH</code> for function <code>animDTAreaUpdate()</code> causing most clocks to fail. As such, discard v4.0 and migrate to v4.0.1.</td></tr>
<tr><td>v4.0
(2017-12-10)
T. Ceulemans</td><td>Emuchron emulator code base:
– Dropped support for Debian 7 (Wheezy).
– Commands 'hc' and 'vp' now use a regular expression as argument to show commands and variables. For consistency reasons, command 'vr' now uses argument value '.' to reset all variables.
– In command 'vr', resetting a variable that does not exist is considered an error.
– Added command 'lgg' to set graphics options in the OpenGL2/GLUT LCD device. For consistency reasons, command 'lnb' is renamed to 'lng' to set graphics options in the ncurses LCD device.
– The OpenGL2/GLUT LCD device will temporarily display window pixel size information after a window resize action.
– Added function rand() in the expression evaluator.
– Improved command list execution performance when running Debian as a VM.</td></tr>
</table>

Version (date) ***Author***	***Description***
	Monochron firmware code base: – Monochron variable `mcUpdAlarmSwitch` is no longer triggered upon a change in date (undocumented feature). – Added Monochron variable `mcClockDateEvent` that is triggered upon a change in date. – `animAlarmAreaUpdate()` is renamed to `animDTAreaUpdate()` and now also supports a date-only area. – `glcdFillCircle2()` now supports fill option `FILL_INVERSE`. In fact, due to v2.0 function optimizations it worked as of that version but until now it was not realized it would. – Added one QuintusVisuals and four Spotfire clocks. – For digital HM clock added select-to-build option to blink the bottom dot of the ':' time separator per second. This option is enabled by default. – Added glcd performance test suite for `glcdRectangle()`. Relevant bug fixes: – Command 'lr' erroneously accepts '*' as input for parameter <variable>. When using 'vp' as next command it may crash mchron. – `glcdFillCircle2()` draws one vertical pixel short in the bottom half of the circle. Generic: – Minor bug fixes, optimizations and refinements in Emuchron, Monochron and clock code. – Latest documentation.

Summary

Emuchron is a lightweight Monochron emulator for Debian Linux 8 and 9.
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 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 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 in early 2014.

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 therefore 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 90 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                          Press '<ctrl>d'
$
```

Table of contents

This page is intentionally left blank

1 Introduction

1.1 About this manual

The purpose of this manual is to provide background information on Adafruit's 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 mostly 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 connecting to one of the LCD controllers with built-in delays to compensate for hardware response times. In our emulator case, Emuchron has a module that implements the controller hardware as a finite state machine. It processes the data byte by storing it in a data structure representing local LCD display memory. When the data byte actually leads to a change in the LCD display, it is 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 controllers and LCD devices there are others that emulate all other hardware elements, being the real time clock, clock buttons, alarm on/off switch, piezo speaker and LCD backlight. 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. Reading the real time clock, sounding the alarm, snoozing, and scanning the buttons and the alarm on/off switch is 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 represents 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 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 its graphic layout, 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 command line tool `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, execute command script files, 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 2`**
  Select the second 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 to make sure the alarm will not be tripped by subsequent mchron commands.
- `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 v4.1 is developed in Debian 8 and has been verified to work in Debian 9. The table below provides the details of the several environments in which Emuchron is verified to work.

Debian environment	***VM info***
Host: Windows-7 Professional Hypervisor: VirtualBox 5.2.12 VM OS: Debian 8 64-bit	Linux kernel: 3.16.0-4-amd64 gcc/avr-gcc: 4.9.2/4.8.1 Memory: 1 GB
Host: Windows-7 Professional Hypervisor: VirtualBox 5.2.12 VM OS: Debian 9 64-bit	Linux kernel: 4.9.0-3-amd64 gcc/avr-gcc: 6.3.0/4.9.2 Memory: 1.5 GB
Host: OS-X 10.13 Hypervisor: VMware Fusion 8.5.9 VM OS: Debian 8 64-bit	Linux kernel: 3.16.0-4-amd64 gcc/avr-gcc: 4.9.2/4.8.1 Memory: 1 GB
Host: OS-X 10.13 Hypervisor: VMware Fusion 8.5.9 VM OS: Debian 9 64-bit	Linux kernel: 4.9.0-3-amd64 gcc/avr-gcc: 6.3.0/4.9.2 Memory: 1.5 GB

Table 1: The Emuchron v4.1 runtime environments for Debian and AVR

Please consider the following:
- The information above shows up-to-date version info at the time of releasing this document, and is not actively maintained. In the development stage of Emuchron earlier versions of VM tools, Linux kernels and hosts were used as well.
- VM performance in VirtualBox on a Windows-7 host has decreased considerably as of Q2 2018. It is assumed it is caused by Intel CPU microcode patches due to Spectre and Meltdown.

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
   #ifndef 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 mchron 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
```

1.6 Migrating from Emuchron v2.x to v3.x

Compared to v2.x, no changes were made in existing mchron commands. However, a v2.x clock may not compile in v3.x due to refactoring efforts in v3.0. Also, in v3.0, clock plugin framework functionality is slightly improved, allowing specific code optimizations at clock plugin level.

Refer below for an overview of the changes.

In Emuchron v3.x, upon a clock full init (`DRAW_INIT_FULL`), the LCD display is already cleared prior to entering the clock `init()` method. As such, a clock plugin no longer needs to clear the display by itself.
Also, upon a clock full init (`DRAW_INIT_FULL`), a clock plugin in its `init()` method no longer needs to initiate an alarm area initialization by setting `mcAlarmSwitch` to `ALARM_SWITCH_NONE`.
Both actions are now taken care of by `animClockDraw()` in anim.c [firmware].
For performance and code object size reasons it is highly recommended to update affected clock plugins accordingly.
Refer below for an example for example.c [firmware/clock].

```
   :
   // Get a subset of the global variables representing the Monochron state
   extern volatile uint8_t mcClockNewTS, mcClockNewTM, mcClockNewTH;
   extern volatile uint8_t mcClockOldDD, mcClockOldDM, mcClockOldDY;
   extern volatile uint8_t mcClockNewDD, mcClockNewDM, mcClockNewDY;
   extern volatile uint8_t mcClockInit;
-- extern volatile uint8_t mcAlarmSwitch;
   extern volatile uint8_t mcClockTimeEvent;
-- extern volatile uint8_t mcBgColor, mcFgColor;
++ extern volatile uint8_t mcFgColor;
   :
   //
   // Function: exampleInit
   //
   // Initialize the lcd display of a very simple clock.
   // This function is called once upon clock initialization.
++ // At this point the display has already been cleared.
   //
   void exampleInit(u08 mode)
   {
     DEBUGP("Init Example");
--
--   // Start from scratch by clearing the lcd using the background color
--   glcdClearScreen(mcBgColor);

     // Paint a text on the lcd with 2x horizontal scaling
     glcdPutStr3(11, 2, FONT_5X7N, "-Example-", 2, 1, mcFgColor);
--
--   // Force the alarm info area to init itself in animAlarmAreaUpdate()
--   // upon the first call to exampleCycle()
--   mcAlarmSwitch = ALARM_SWITCH_NONE;
   }
```

In Emuchron v3.0 refactoring efforts were made for, amongst others, coding consistency reasons. Find below an overview of impacted Monochron objects. Note that this list is incomplete and includes only those objects that are likely to be used in clock plugins.

The following relevant `#define` values were renamed and/or relocated:

Description	***API impact***

Old color defines:
`ON`, `OFF`
New color defines:
`GLCD_ON`, `GLCD_OFF`
	Value for `mcFgColor`, `mcBgColor`.
	All `glcd`*`Function`*`()` functions supporting parameter `color`.

Old button defines:
`BTTN_PLUS`, `BTTN_SET`
New button defines:
`BTN_PLUS`, `BTN_SET`
	Value for clock *`clock`*`Button()` function parameter `pressedButton`.
	Also, to obtain the button `#define` values, a clock must now include header buttons.h [firmware].

The following relevant static data variables were renamed:

Variable old name	***Variable new name***
`const char *days[7]`	`const char *animDays[7]`
`const char *months[12]`	`const char *animMonths[12]`

The following relevant utility functions were renamed:

Function old name	***Function new name***
`uint8_t int2bcd()`	`uint8_t bcdEncode()`
`uint8_t dotw()`	`uint8_t rtcDotw()`
`uint8_t leapyear()`	`uint8_t rtcLeapYear()`

1.7 Migrating from Emuchron v3.x to v4.x

In v4.x, a v3.x clock may not compile, may not work correctly or can be optimized due to changes in the Monochron framework variables and generic support functions.
In addition to that, in v4.x changes are made in existing mchron commands that may require changes in v3.x script files.

Refer below for an overview of the changes.

Prior to v4.x, `mcUpdAlarmSwitch` includes an undocumented feature that triggers the variable upon a date change. This allowed to optimize code in generic function `animAlarmAreaUpdate()`.
In v4.x, `mcUpdAlarmSwitch` is no longer triggered upon a date change. Instead, Monochron variable `mcClockDateEvent` is introduced that is triggered upon a date change. Any code relying on this undocumented feature of `mcUpdAlarmSwitch` must be changed to use `mcClockDateEvent`.
Another benefit of this change is that `mcClockDateEvent` allows writing more compact code to detect date changes in functional clock code. Refer below for a code snippet from example.c [firmware/clock].

In v3.x, generic utility function `animAlarmAreaUpdate()` is used to draw and control a date/alarm area in a clock. In v4.x, this function now also supports date-only areas in a clock. Because of this the function has been renamed to `animADAreaUpdate()` while `#define` values for its last argument have been renamed to reflect the new function name. Refer below for a code snippet from example.c [firmware/clock].

```
   :
   // Get a subset of the global variables representing the Monochron state
   extern volatile uint8_t mcClockNewTS, mcClockNewTM, mcClockNewTH;
-- extern volatile uint8_t mcClockOldDD, mcClockOldDM, mcClockOldDY;
   extern volatile uint8_t mcClockNewDD, mcClockNewDM, mcClockNewDY;
   extern volatile uint8_t mcClockInit;
-- extern volatile uint8_t mcClockTimeEvent;
++ extern volatile uint8_t mcClockTimeEvent, mcClockDateEvent;
   extern volatile uint8_t mcFgColor;

   //
   // Function: exampleCycle
   //
   // Update the lcd display of a very simple clock.
   // This function is called every application clock cycle (75 msec).
   //
   void exampleCycle(void)
   {
     char dtInfo[9];

     // Use the generic method to update the alarm info in the clock.
     // This includes showing/hiding the alarm time upon flipping the alarm
     // switch as well as flashing the alarm time while alarming/snoozing.
--   animAlarmAreaUpdate(2, 57, ALARM_AREA_ALM_ONLY);
++   animADAreaUpdate(2, 57, AD_AREA_ALM_ONLY);
     :
     // Only paint the date when it has changed or when initializing the clock
--   if (mcClockNewDD != mcClockOldDD || mcClockNewDM != mcClockOldDM ||
--       mcClockNewDY != mcClockOldDY || mcClockInit == GLCD_TRUE)
++   if (mcClockDateEvent == GLCD_TRUE || mcClockInit == GLCD_TRUE)
     {
       // Put new month, day, year in a string and paint it on the lcd
       :
     }
   }
   :
```

In v4.x commands 'hc' and 'vp' now use a regular expression pattern as argument. As a result, for consistency reasons, command 'vr' now uses argument value '.' instead of '*' to reset all variables.
When any of these commands is used in a v3.x command script it is very likely that the script needs to be corrected for v4.x.
Refer below for a few examples. See also section 5.8.

```
# Emuchron v3.x: commands 'he', 'vp' and 'vr' accept specifying a single command
# or variable name, or a '*' representing all commands and variable names.
hc *
vp abc
vr *

# Emuchron v4.x: commands 'he' and 'vp' now use a regular expression pattern as
# argument for searching commands and variables, where '.' represents all
# commands and variables. Command 'vr' accepts either a single variable name or a
# '.' for all variables.
hc .
vp ^abc$
vr .
```

Command 'lnb' (ncurses backlight support) has been renamed to 'lng' (ncurses graphics options), due to the introduction of command 'lgg' that sets graphics options for the glut LCD device.
Any script that refers to command 'lnb' must be updated to use 'lng' instead.

2 The Emuchron project

2.1 The project folder structure

The Emuchron project uses the following folder structure.

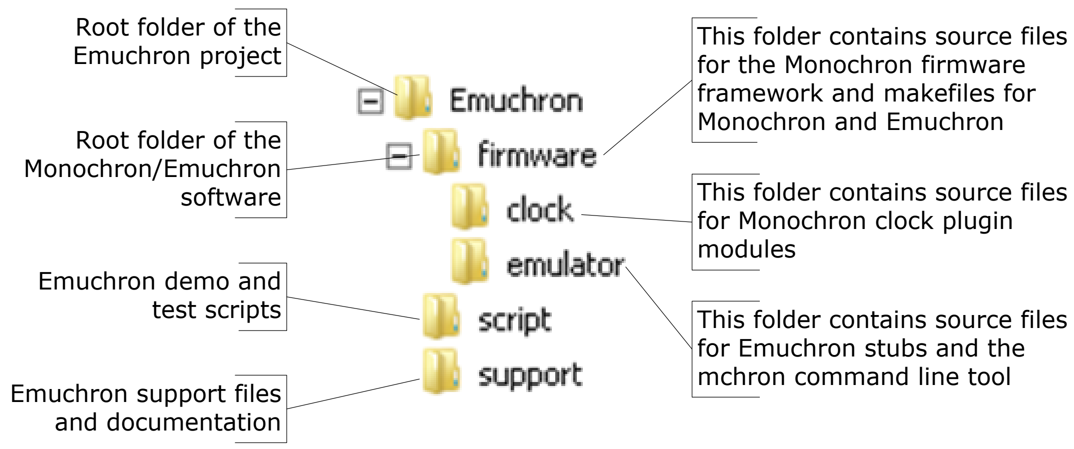


Figure 1: The Emuchron project folder structure

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.

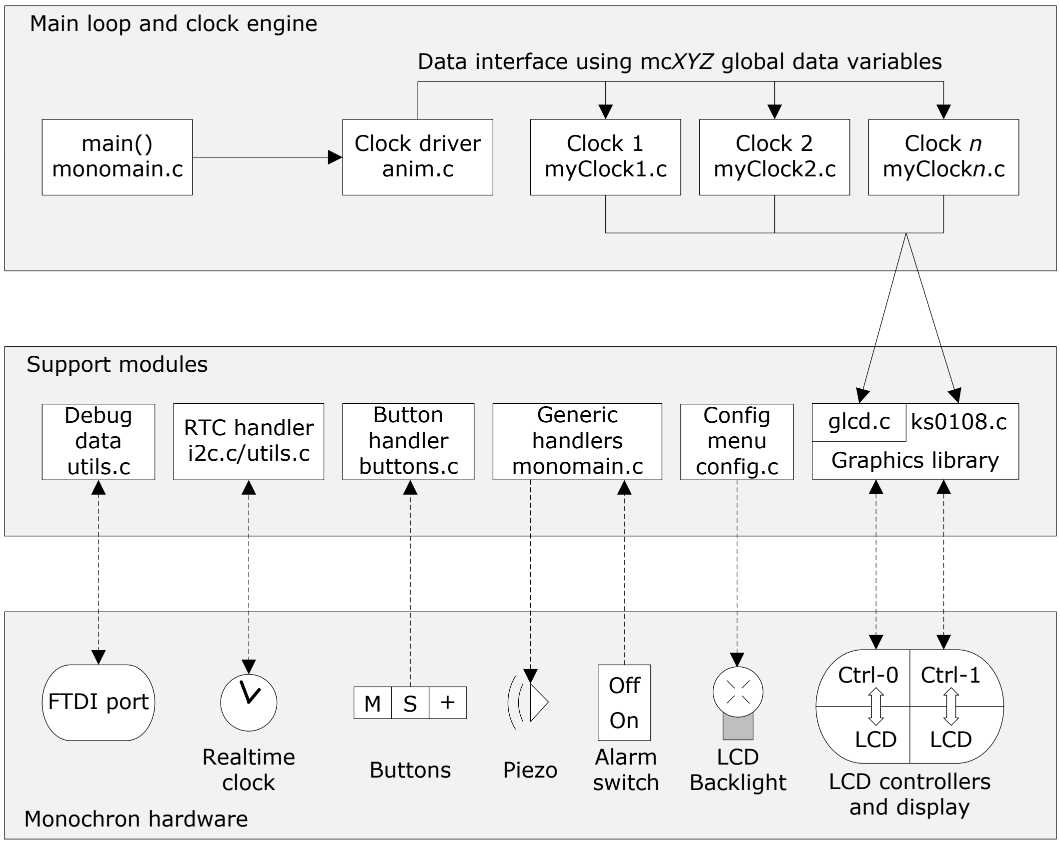


Figure 2: The Monochron runtime environment

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***
anim.c [firmware]	In module anim.c 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 mc*XYZ* data interface to the clock plugins.
buttons.c [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 monomain.c [firmware] and config.c [firmware].
config.c [firmware]	This support module contains the main entry for the configuration menu as used in the Monochron application. It is activated in `main()` by pressing the 'M' button. Via one of the menu items the LCD backlight brightness is changed.
glcd.c [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 ks0108.c [firmware] for the actual interface to the LCD controllers.
i2c.c [firmware]	In this module we find the interface to the real time clock (RTC).
ks0108.c [firmware]	The low-level graphics library. It contains functions to interact with two hardware ks0108 LCD controllers, driving the left and right side of the LCD display. This module initializes the controller hardware, interacts with controller hardware registers, clears the LCD, and writes data to and reads data from the LCD.
monomain.c [firmware]	In module monomain.c we find the `main()` function. Next to `main()`, monomain.c contains much additional functionality related to interrupt handlers as well as handling the real time clock, the alarm and snooze logic, the piezo speaker and the state of the alarm switch. The `main()` function contains an infinite loop and will interact with the clock driver in anim.c [firmware] and the clock configuration menu in config.c [firmware] when appropriate. It is responsible for a subset of the mc*XYZ* data interface to the clock plugins.
myClock*x*.c [firmware/clock]	A Monochron plugin clock. Based on the mc*XYZ* data interface the module is responsible for drawing and updating itself on the LCD. This is where functional clock code resides.
utils.c [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. Prior to Emuchron, this used to be the only method available for debugging a Monochron application.

Table 2: The Monochron modules

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.

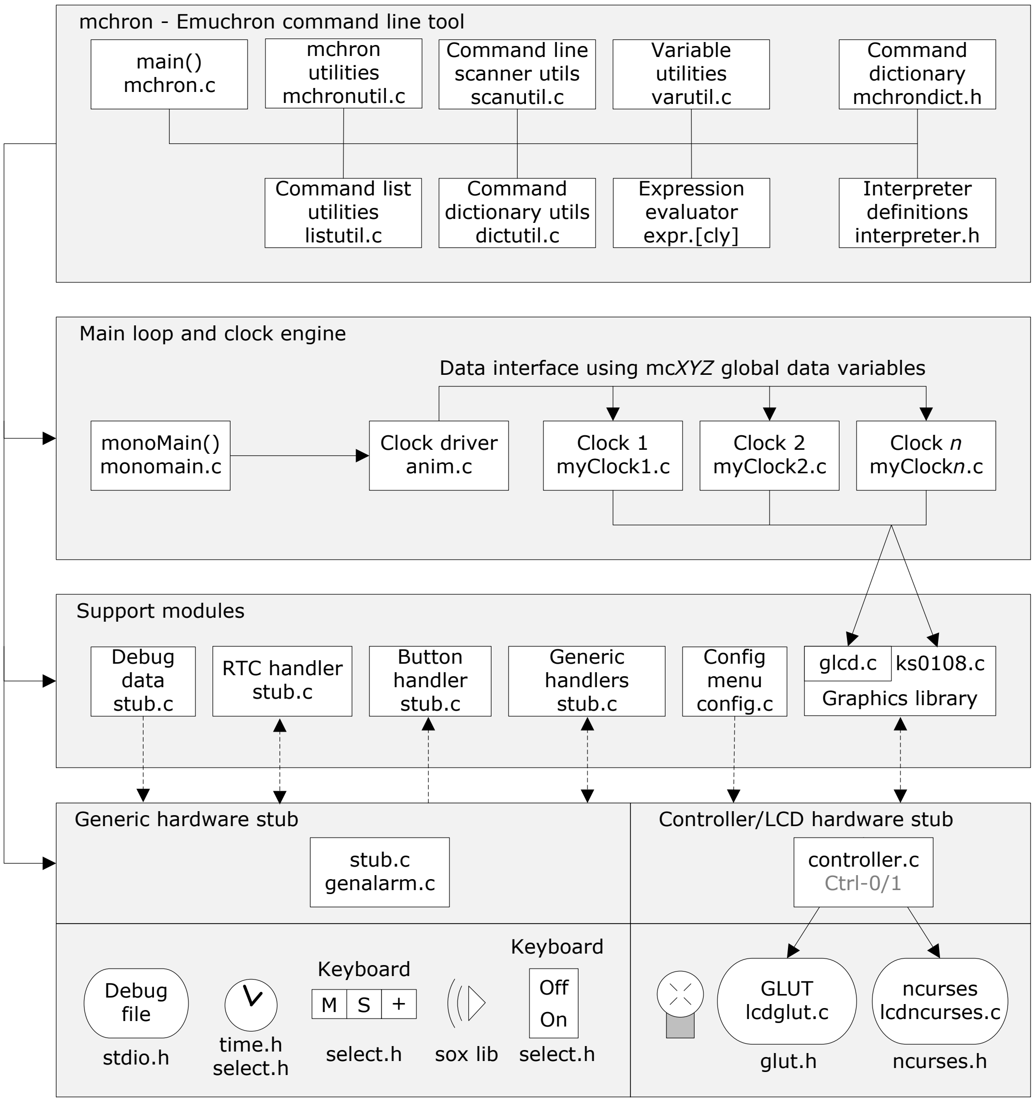


Figure 3: The Emuchron runtime environment

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] – myClock*x*.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 myClock*x*.c [firmware/clock] and glcd.c [firmware] does not require any stub functionality.

- In monomain.c [firmware] the `main()` has been renamed to `monoMain()` but besides that effectively remains the same function.
  Note that the mchron command line tool can start a fully functional Monochron application simply by calling `monoMain()`.
- All hardware has been stubbed by stub.c [firmware/emulator] and controller.c [firmware/emulator] and is emulated using off-the-shelf Linux libraries.
- Monochron modules like i2c.c [firmware], buttons.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***
controller.c [firmware/emulator]	The controller module implements the stubbed ks0108 LCD controllers and 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, including changes in the backlight brightness setting. Note: As controller.c 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 controller.c as long as it publishes functions similar to the GLUT and ncurses modules.
dictutil.c [firmware/emulator]	This module implements the utilities to access the mchron command dictionary.
expr.c/expr.l/expr.y [firmware/emulator]	The flex (expr.l) and bison (expr.y) modules implement an expression evaluator. The code generated by flex and bison is included in the master module (expr.c) and compiled into a separate expression evaluator object. The following elements are supported: – Math operators +, -, *, /, % (modulo), ^ (power), =, ( ) – Bit operators <<, >>, &, |, ~ – Logic operators <, >, <=, >=, ==, !=, &&, ||, ?: ('C'-style ternary operator) – Functions abs(), cos(), frac(), int(), rand()[1], sin() – Constants null, pi, true, false [1] For rand() seeding and number generation options see its actual implementation in expr.y. Refer to script dot1.txt [script] for a functional overview and actual use-case.
genalarm.c [firmware/emulator]	This is the source for utility tool `genalarm` that creates alarm audio file alarm.au [firmware/emulator] containing a Mario melody or two-tone tune. The audio file is played when the alarm is triggered or resumes from snoozing. The `genalarm` tool is built and executed at Emuchron build time.
interpreter.h [firmware/emulator]	This module defines the core structures and constants for the mchron interpreter.

Module	Description
lcdglut.c [firmware/emulator]	This module implements an OpenGL2/GLUT LCD device. The GLUT device is implemented using a separate thread, making the GLUT window update itself asynchronously from the mchron 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 and that the GLUT window can be resized at will while retaining the 2:1 aspect ratio.
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 clock 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, that its window cannot be freely resized (but we can use keyboard shortcuts instead) and that the ncurses library does not play nice with gdb (refer to section 6.3.2).
listutils.c [firmware/emulator]	This module implements the utilities to build and cleanup command lists, as well as functions to execute a single command line and a command list.
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 numeric 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.
mchrondict.h [firmware/emulator]	The mchrondict header module creates the mchron command dictionary. It defines a set of structures of (from low level to high level) domain values, command arguments, commands and command groups. The command dictionary itself 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 publisher of command line arguments to a command handler.
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 and its controllers.
varutils.c [firmware/emulator]	This module implements the administration of the interpreter named variables.

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 described.

Generic functionality in `main()`:

- A single loop application clock cycle is executed every 75 msec.
  This is defined by `#define ANIM_TICK_CYCLE_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 clock 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 config.c [firmware]. After exit of configuration menu: invoke `init()` for clock with `DRAW_INIT_FULL` invoke `cycle()` for clock
Press 'S' button	if `button()` is defined for clock then invoke `button()` for clock end-if invoke `cycle()` for clock
Press '+' button	if `button()` is defined for clock then invoke `button()` for clock end-if invoke `cycle()` for clock
No button pressed	invoke `cycle()` for clock

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

Per application clock 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 config.c [firmware]. After exit of configuration menu: invoke `init()` for clock with `DRAW_INIT_FULL` invoke `cycle()` for clock

Press 'S' button	if `button()` is defined for clock
	then
	   invoke `button()` for clock
	else
	   select next clock in `monochron[]` (round-robin)
	   invoke `init()` for clock with `monochron[].initType`
	end-if
	invoke `cycle()` for clock

Event	*Action*
Press '+' button	select next clock in `monochron[]` (round-robin) invoke `init()` for clock with `monochron[].initType` invoke `cycle()` for clock
No button pressed	invoke `cycle()` for clock

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

Per application clock 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 invoke `cycle()` for clock
Press 'S' button	reset snooze timer timeout invoke `cycle()` for clock
Press '+' button	reset snooze timer timeout invoke `cycle()` for clock
No button pressed	invoke `cycle()` for clock

Table 6: Single clock 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*
`mcAlarmH` `mcAlarmM`	The active alarm time (hour, min), regardless whether the alarm switch position is on or off.
`mcAlarming`	Value: `GLCD_TRUE` / `GLCD_FALSE` Indicates whether the clock is alarming/snoozing (`GLCD_TRUE`) or not (`GLCD_FALSE`).
`mcAlarmSwitch`	Value: `ALARM_SWITCH_NONE` / `ALARM_SWITCH_ON` / `ALARM_SWITCH_OFF` The current on/off state of the alarm switch. – `ALARM_SWITCH_NONE` This value clears the stored value and forces an alarm switch change event in `mcUpdAlarmSwitch`. If needed, set its value in the clock `init()` method. In case of a full clock init, the clock plugin API will take care of this prior to entering the first clock `cycle()`. For a description of clock initialization types refer to section 2.12. Note: The clock `cycle()` will only see values `ALARM_SWITCH_ON` and `ALARM_SWITCH_OFF`. – `ALARM_SWITCH_ON` Indicates that the alarm switch position is switched on. – `ALARM_SWITCH_OFF` Indicates that the alarm switch position is switched off.

Variable	***Description***
`mcBgColor` `mcFgColor`	Value: `GLCD_ON` (white pixel) / `GLCD_OFF` (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 properly implements its drawing graphics with these variables, can freely swap between showing itself white-on-black and black-on-white without any code changes.
`mcClockInit`	Value: `GLCD_TRUE` / `GLCD_FALSE` Indicates that a clock must initialize itself. It is set prior to calling the clock `init()` and is reset after executing a clock `cycle()`.

`mcClockNewTH`
`mcClockNewTM`
`mcClockNewTS`
`mcClockNewDD`
`mcClockNewDM`
`mcClockNewDY`
	The new Monochron clock time (hour, min, sec) and date (day, month, year).

`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, pointing to the active clock. In the Monochron application the clock array being used is `monochron[]` in anim.c [firmware]. In Emuchron the clock array being used is `emuMonochron[]` in mchron.c [firmware/emulator].
`mcClockDateEvent`	Value: `GLCD_TRUE` / `GLCD_FALSE` Indicates that the date has changed. This event must be handled in the clock `cycle()` as it is reset every application clock cycle. Note that `mcClockDateEvent` can be `GLCD_TRUE` only when `mcClockTimeEvent` is `GLCD_TRUE`.
`mcClockTimeEvent`	Value: `GLCD_TRUE` / `GLCD_FALSE` Indicates that the time and/or date has changed. This event must be handled in the clock `cycle()` as it is reset every application clock cycle.
`mcCycleCounter`	A counter that is incremented every application 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 some demo clocks where `mcU8Util1` is used to store the blinking state of the alarm draw area when alarming or snoozing. Note that these variables are under control of the active clock and as such must be initialized, set and processed in clock code.

Variable	Description
mcUpdAlarmSwitch	Value: GLCD_TRUE / GLCD_FALSE Signals a change in the alarm switch position. This event must be handled in the clock cycle() as it is reset every application clock cycle. Use it in combination with mcAlarmSwitch.

Table 7: The Monochron variables for clock plugins

In a clock plugin cycle() function the population of variables mcClockNew*XY* and mcClockOld*XY* are tied to variable mcClockTimeEvent as follows.

Variables	Impact
mcClockTimeEvent = GLCD_FALSE	mcClockOld*XY* = the last created timestamp mcClockNew*XY* = the last created timestamp
mcClockTimeEvent = GLCD_TRUE	mcClockOld*XY* = the previous timestamp mcClockNew*XY* = the new timestamp

Table 8: The Monochron time event and time variables

2.6 The glcd graphics library enhancements

This project is based on the original Monochron pong firmware. To enhance the graphic 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 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.

Function	Description
`glcdPutStr2()`	Proxy function for `glcdPutStr3()` with a reduced interface regarding font scaling, allowing to optimize code on object size.
`glcdPutStr3()`	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 `glcdPutStr3()` function draws horizontal text at any (x,y) pixel location, allows independent font scaling on the x and y axis, 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].
`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 `glcdBuffer[]` buffer

The interface to the LCD controllers can be optimized by preventing frequent switching between LCD read and LCD write operations.
For this, most graphics functions have implemented a method to read all relevant LCD bytes from a single LCD byte row in buffer `glcdBuffer[]` first, next apply changes to the buffered data and then write the modified data back to the LCD.
This method significantly improves the speed of the graphics interface to the LCD. The downside is that 128 bytes of stack RAM (out of 2 KB) is constantly allocated for this purpose, which is considered acceptable.

2.6.3 Text fonts and font scaling

Most 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].
`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:

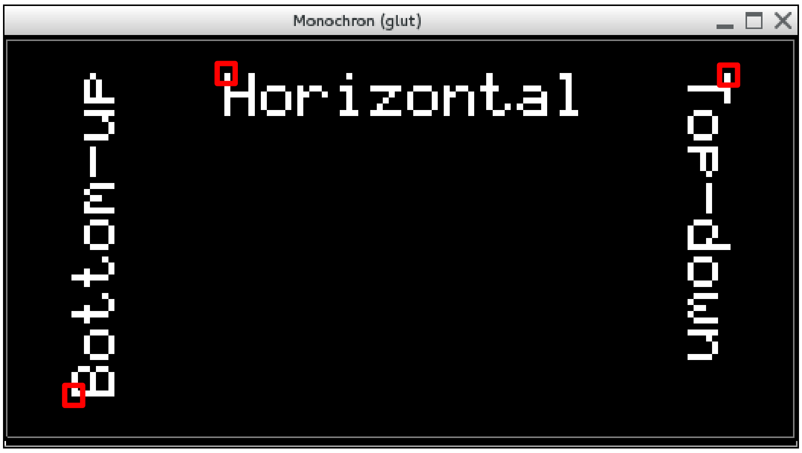
Text orientation	***Description***
`ORI_HORIZONTAL`	Paint the text horizontally.
`ORI_VERTICAL_BU`	Paint the text vertically in a bottom-up direction.
`ORI_VERTICAL_TD`	Paint the text vertically in a top-down direction.

Table 11: Text orientation overview

Enter the following mchron commands.

```
mchron> pa f 35 5 5x7n h 1 1 Horizontal
mchron> pa f 10 57 5x7n b 1 1 Bottom-up
mchron> 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. Vertical text is painted using function `glcdPutStr3v()`.



2.6.5 Fill patterns

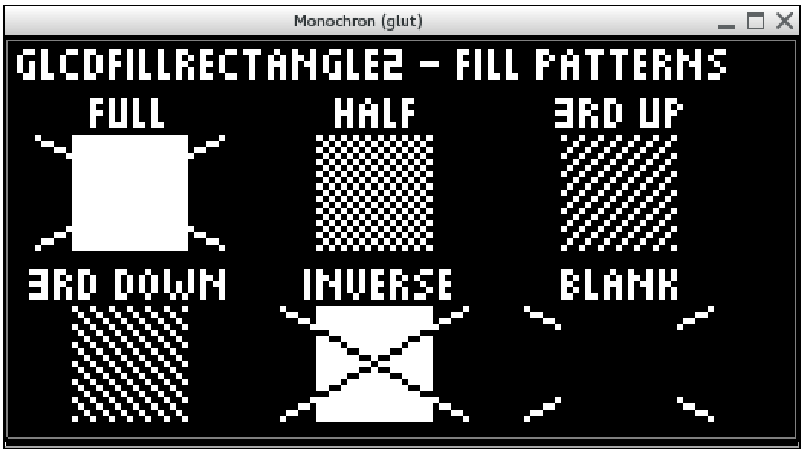
The `glcdFillRectangle2()` and `glcdFillCircle2()` functions provide a method to fill an area with several fill patterns.

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_THIRDUP`	The area is filled with a 1/3rd pattern using the given paint color giving an upward illusion.
`FILL_THIRDDOWN`	The area is filled with a 1/3rd pattern using the given paint color giving a downward illusion.
`FILL_INVERSE`	The area is inverted.
`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

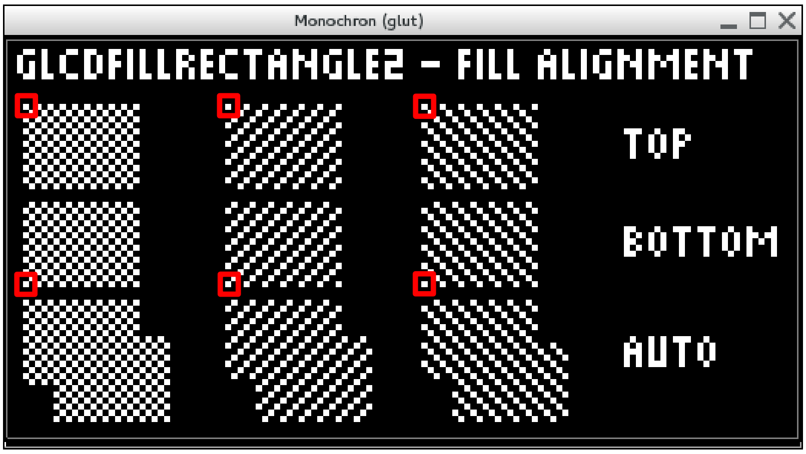
The `glcdFillRectangle2()` function supports a fill alignment option for fill patterns `FILL_HALF`, `FILL_THIRDUP` and `FILL_THIRDDOWN`.

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

Alignment	***Description***
`ALIGN_TOP`	The top-left pixel of the fill area is filled with the given paint color.
`ALIGN_BOTTOM`	The bottom-left pixel of the fill area is filled with the given paint color.
`ALIGN_AUTO`	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 properly overlap one another.

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

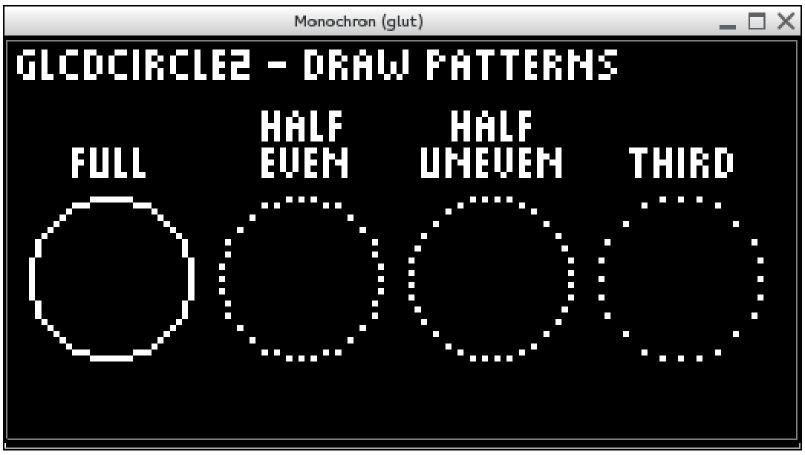
The `glcdCircle2()` function provides a method to draw a circle using several patterns.
Note that the method to determine which pixels are drawn is rather crude. The quality of the non-full draw patterns will vary depending on the radius and center of the circle being drawn.

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 1/3rd 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.

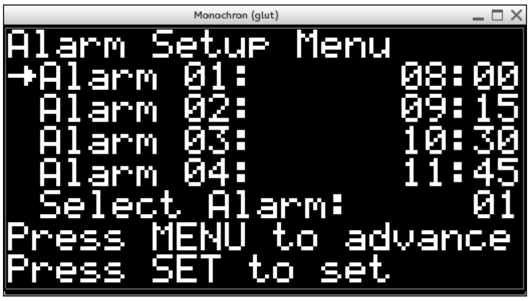


2.7 Monochron configuration screens

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

The two alarm melodies are mutually exclusive. Switching between the two is done by (un)defining `MARIO` in alarm.h [firmware]. In the same file specify the two-tone alarm tones and tone duration. See below an excerpt where is chosen to use the Mario alarm.

```
// Uncomment this if you want a Mario tune alarm instead of a two-tone alarm.
// Note: This will cost you ~536 bytes of Monochron program and data space.
#define MARIO

// Configure two-tone alarm
#define ALARM_FREQ_1          4000
#define ALARM_FREQ_2          3750
#define SND_TICK_TONE_MS      325
```

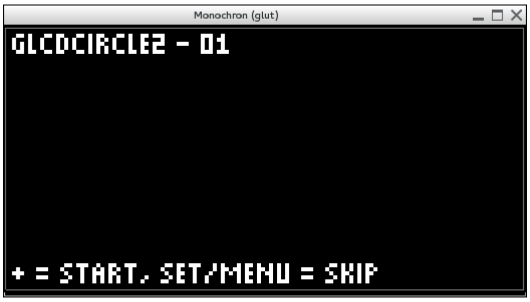
Alarming and snoozing timeouts are controlled by the following defines in alarm.h [firmware]. Note that for the emulator reduced timeouts are specified.

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

2.9 Performance tests for high-level glcd functions

Modifying a 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 high-level 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 performance test runs.
For code refer to perftest.c [firmware/clock].



2.10 Demo and test mchron command scripts

In node [script] mchron 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.
circle*X*.txt [script]	A total of six scripts for testing high-level glcd graphics. The first three scripts verify the correctness of the circle functions.
controller*X*.txt [script]	A total of 4 scripts for testing the LCD controller state machine. It verifies the correctness of the controller command set and its impact on the LCD devices.
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.
dot*X*.txt [script]	Two scripts for testing high-level glcd graphics. The first one is also a showcase for the built-in rand() function.
line*X*.txt [script]	A total of seven scripts for testing high-level glcd graphics. It verifies the correctness of the line function.
rectangle*X*.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.
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 pre-built monochron.hex [firmware] firmware using avr-gcc 4.8.1 (Debian 8).
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 alarm.h [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 that contain 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***
`clockId`	This is the unique clock Id assigned to a clock, as defined in anim.h [firmware].
`initType`	The initialization mode that is forwarded to the `init()` function of a clock. It has two distinctive values as defined in anim.h [firmware]. – `DRAW_INIT_FULL` The clock begins from scratch. The LCD display has already been cleared prior to entering the `init()` function, so the clock plugin can start with a full graphic build-up of the static clock layout. – `DRAW_INIT_PARTIAL` 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 analog.c [firmware/clock] and digital.c [firmware/clock].
`init()`	This is the published initialization function for a clock. It is invoked via anim.c [firmware] when the clock needs to initialize itself.
`cycle()`	This is the published cycle function for a clock. It is invoked via anim.c [firmware] every main loop application clock cycle of 75 msec, thus giving the clock the opportunity to update itself. This function must handle changes in time, changes in the position of the alarm on/off switch, and changes in the alarming/snoozing state of the clock.
`button()`	This is the optional published function for a clock. When published, it is invoked via anim.c [firmware] in a main loop application clock 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.
This overview is based on the `CHRON_EXAMPLE` clock as found in example.c and example.h [firmware/clock]. It is a bare bone yet fully functional Monochron clock plugin with proper date, time and alarm area handling.

File	***Description***
anim.h [firmware]	– Create a new unique id with unique number for the clock. `#define CHRON_EXAMPLE 19`

File	**Description**
anim.c [firmware]	– Include the new clock header. `#include "clock/example.h"` – When you want to test or upload your new clock to the actual Monochron clock, add the clock id and public `init()`, `cycle()` and (optional) `button()` functions for the clock in static array `monochron[]`. The example clock does not have a `button()` function.
example.c [firmware/clock]	– Create a new clock source file that implements the public and private functions for the clock.
example.h [firmware/clock]	– Create a new clock header file that publishes the public `init()`, `cycle()` and (optional) `button()` functions for the clock. The example clock does not have a `button()` function.
help.txt [firmware/emulator]	– Modify the help text for command 'cs' by adding the clock index in `emuMonochron[]` and a short description. See also the changes for mchron.c [firmware/emulator].
Makefile [firmware]	– When appropriate add the new example.c file in variable `SRC`. This is required for building Monochron firmware that includes the new clock. See also changes for anim.c [firmware].
MakefileEmu [firmware]	– Add the new example.c file in variable `CSRC`. This is required for building Emuchron and the mchron command line tool.
mchron.c [firmware/emulator]	– Include the new clock header. `#include "../clock/example.h"` – Add the clock id and public `init()`, `cycle()` and (optional) `button()` functions for the clock in static array `emuMonochron[]`. The example clock does not have a `button()` function. – Verify if the clock requires special handling in `doAlarmSet()`. For the example clock this is not the case.
mchronutil.c [firmware/emulator]	– Verify if the clock requires special handling in `emuClockUpdate()`. For the example clock this is not the case.

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 64-bit versions of Debian 8 and 9. 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 64-bit Debian in a VM it is required to enable Intel (VT-x) or AMD (AMD-V) CPU Virtualization Technology in the BIOS of the host machine. On Intel Macs this is enabled by default.
Also, verify that the VM accepts USB devices. In general, when the contents of a plugged-in USB flash drive can be seen in the VM, the VM is also able to successfully attach to the FTDI USB device.
In section 1.4 an overview is provided of hypervisors used and observed issues.

3.2.2 Configuring a Debian VM in VirtualBox

Debian Linux clients in VirtualBox seem to have erratic support for 3D acceleration, and when enabled may result in vastly degraded OpenGL2/GLUT performance or mchron crashes occurring upon exiting the tool. It is therefore recommended to disable 3D acceleration. Only enable and keep enabled when it has proven to work properly.

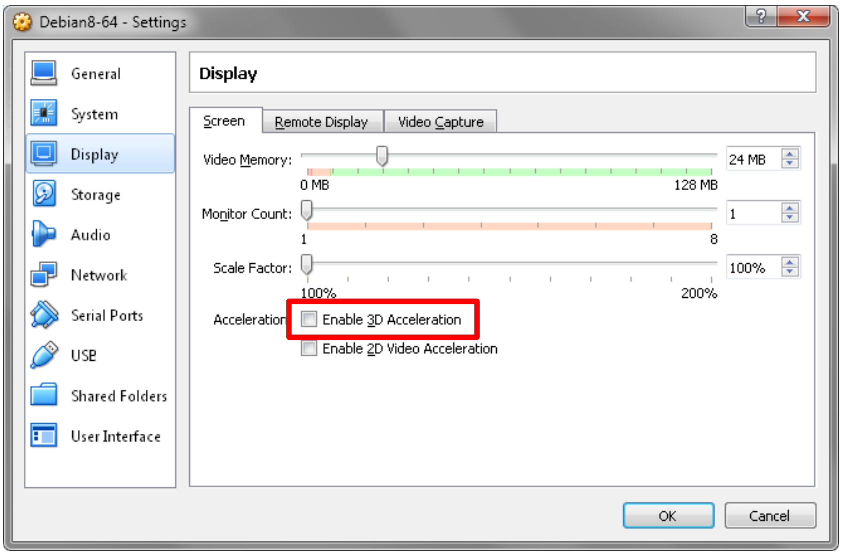


Figure 4: Disable 3D acceleration for a Debian VM in VirtualBox

3.2.3 Configuring a Debian VM in VMware Fusion

When using 3D acceleration in combination with the open source version of VMware tools (Debian packages 'open-vm-tools', 'open-vm-tools-desktop' and 'open-vm-tools-dkms'), it may result in erratic display update behavior in, oddly enough, the Monochron ncurses LCD device.

For this configuration it is therefore recommended to disable 3D acceleration. Only enable and keep enabled when it has proven to work correctly.

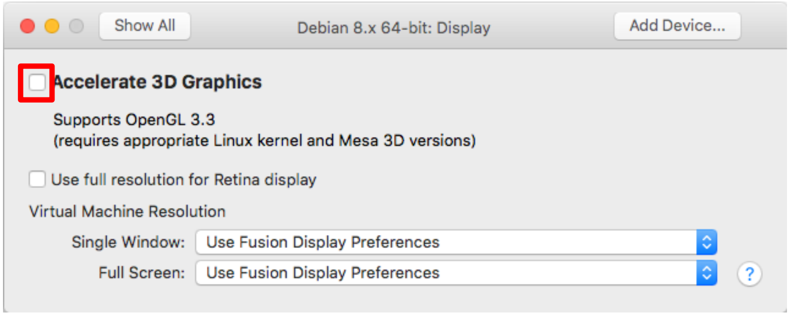


Figure 5: Disable 3D acceleration for a Debian VM in VMware

3.3 Unpacking the project software

The Emuchron project package can be downloaded from github location https://github.com/tceulema/Emuchron and can be unpacked in any location. Make sure that full read and write access is granted to the project root and its folder 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 glibc source files.

In the Emuchron node the shell script packages.txt [support] 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.

Note: For background information regarding downloading and installing glibc source files refer to section 3.7.2.

3.5 Copying configuration file for minicom

The `minicom` application is used for debugging the actual Monochron clock. It allows making a connection to Monochron via the FTDI port and, when proper firmware is uploaded to Monochron, extract runtime debug messages from the port.
The `minicom` application 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 minirc.Monochron [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 an OpenGL2/GLUT device and an ncurses device. The OpenGL2/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 are found in appendix C.

3.6.2 Starting a Monochron ncurses terminal

Once a terminal profile is created a Monochron ncurses terminal is started by executing the proper shell command.
For this, a command shortcut named Monochron [support] or 'gnome-terminal.desktop' [support] is available. 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; echo -ne '\033]0\073Monochron (ncurses)\007';
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 is fed with a number of commands.
6. The first command 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 information on mchron being able to link to an ncurses tty refer to section 5.2.
7. The second command will echo appropriate escape characters and text, resulting in the terminal header being set to "Monochron (ncurses)".
8. The last command will start a new bash, keeping the window alive and ready to be taken over by ncurses.

When the Monochron terminal profile is properly setup, double-clicking the Monochron shortcut will create a blank black Monochron ncurses terminal as shown below. Note the small command prompt at the top left of the window, caused by the very small font point size.

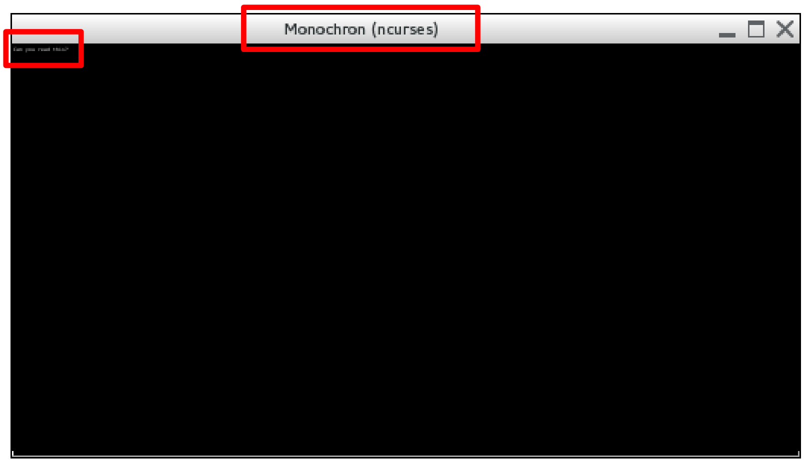


Figure 6: A blank Monochron terminal

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

When running the emulator in an ncurses terminal, its size may not be increased or decreased 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 produce square 'pixels'.

3.7 Debian 8 and 9 issues and regression in functionality

3.7.1 ALSA audio is getting ever less responsive in a VM

This issue only applies to Debian installations that run as a VM. To illustrate the problem, in mchron, execute script beep.txt [script] in either Debian 8 or 9. There is a time lag between individual beeps that has gotten worse over time when compared to older Debian releases (6 and 7). As of Debian 8 however not only the time lag between individual beeps increased, but in-between beeps random underflow buffer errors occur. Also, short audio pulses may get clipped and are not played at al.
In general, in VM's, the ALSA audio interface gets less and less responsive with each Debian release.

In Emuchron measures are taken to suppress buffer underrun error messages from being displayed when using the mchron beep command. In addition to that, a configurable workaround for clipping short audio pulses is found in stub.c [firmware/emulator]. This workaround is currently disabled as specific sox audio options appear to prevent audio clipping behavior, but can be enabled by setting a value for define `ALSA_PREFIX_PULSE_MS`.

```
// ALSA_PREFIX_PULSE_MS: Set the blank audio pulse cutoff (msec).
:
#define ALSA_PREFIX_PULSE_MS  95
```

3.7.2 Debugger is missing "syscall-template.S" or "pselect.c"

In Debian 8 and 9, 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" or "pselect.c" cannot be found. Even worse, in some cases it is seen that DDD fails to debug mchron or attach to a running mchron process because of the missing file.
The issue originates from how Linux run-time libraries are built. It causes gdb to forward the missing file to its graphical front-end, eventually resulting in the pop-ups. In short, 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 [support] script will do this specifically for Debian 8 and 9 systems.
The main problem however is that the location where gdb expects to find the sources is not static and may differ per Debian Linux kernel release.
Note that only Nemiver will provide the full path of the file not being found. Refer below for an example.

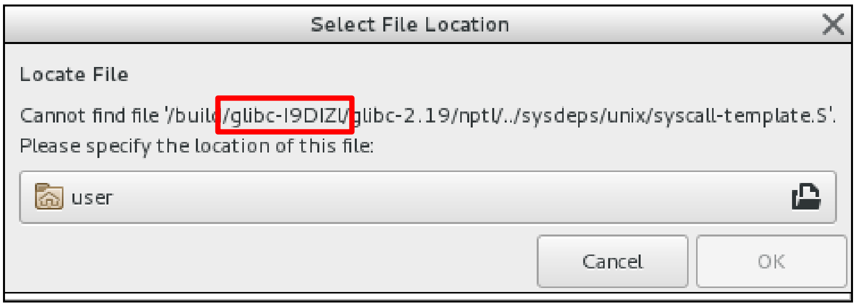


Figure 7: The gdb debugger cannot find file syscall-template.S

The part in the path that may vary per Debian kernel release is highlighted, in this case "glibc-I9DIZl" for Debian 8.
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] script that installs the glibc sources already creates a number of symbolic links that appear to be 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 creates 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 /build/glibc-ABC
$ su - root
$ cd /build
$ ln -s /build/glibc /build/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.

3.7.3 Debian 9 Nemiver has erratic multi-thread debug behavior

When using Nemiver in Debian 9 and debugging a multi-threaded application (as is the case for mchron when using the OpenGL2/GLUT LCD device), thread context switching is erratic.
This erratic behavior shows itself when stopping in the program, after which Nemiver may switch to the wrong thread context.
For example, when stopping or stepping through the main mchron thread, after each stop Nemiver makes the OpenGL2/GLUT thread the active thread.
This is annoying and considered a Nemiver or gdb bug. It requires the end-user to manually switch the context back to the proper thread prior to using any debugger step functionality. In Nemiver the thread context is selected in the most bottom left window section of the context tab.
Note that this behavior does not apply when only using the ncurses LCD device, as ncurses does not have its own thread.

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 contains the proper list of clock plugin modules.
- 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 before:
monochron.elf  :
section      size      addr
.data         984   8388864
.text       25510         0
.bss          313   8389848
.stab       56844         0
.stabstr    16227         0
.comment       17         0
Total       99895
```

The Monochron Atmel CPU contains 32 KB flash memory, of which 30 KB is available for Monochron firmware. Verify that the sum of .data and .text does not exceed 30720 bytes (=30 KB). 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.
Please note that even though the program and data size is below the 30 KB firmware size limit, a Monochron application may still unexpectedly crash at runtime due to exceeding the 2 KB RAM limit. This behavior cannot be detected using the memory map provided by avr-gcc. In the example clocks provided by this project, the QR clocks use large amounts of stack RAM that in combination with other clocks in a Monochron application may cause it to crash due to exceeding that 2 KB RAM limit at runtime.

When a 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. See below.

```
Linking: monochron.elf
avr-gcc -mmcu=atmega328p -I. -g -Os -MMD -funsigned-char -funsigned-bitfields -
fpack-struct -fshort-enums -Wall -Wstrict-prototypes -DF_CPU=8000000 -Wa,-
adhlns=anim.o  -std=gnu99 anim.o buttons.o config.o glcd.o i2c.o ks0108.o
monomain.o util.o clock/digital.o clock/analog.o clock/puzzle.o clock/spotfire.o
clock/cascade.o clock/speeddial.o clock/spiderplot.o clock/trafficlight.o  --
output monochron.elf -Wl,-Map=monochron.map,--cref -lm
anim.o: file not recognized: File format not recognized
collect2: error: ld returned 1 exit status
Makefile:293: recipe for target 'monochron.elf' failed
make: *** [monochron.elf] Error 1
$
```

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

4.2 Building Emuchron and the mchron command line tool

Emuchron and its mchron command line tool use a dedicated make file, being MakefileEmu [firmware]. The Emuchron build does not require any configuration.

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

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. See below.

```
:
gcc -o mchron emulator/stub.o emulator/controller.o emulator/lcdglut.o
emulator/lcdncurses.o emulator/scanutil.o emulator/mchronutil.o
:
clock/cascade.o clock/speeddial.o clock/spiderplot.o clock/trafficlight.o -lm -
lncurses -lreadline -lglut -lGLU -lGL -lpthread emulator/expr.o -lfl
/usr/bin/ld: monomain.o: Relocations in generic ELF (EM: 83)
monomain.o: error adding symbols: File in wrong format
collect2: error: ld returned 1 exit status
MakefileEmu:95: recipe for target 'mchron' failed
make: *** [mchron] Error 1
$
```

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. Verify USB support by attaching a USB flash drive prior to attaching the USB FTDI device.
- When plugged in, the USB FTDI device will appear as logical device /dev/ttyUSB*x*.
  To prevent confusion on which hardware USB device is which logical /dev/ttyUSB*x* device, it is recommended to unplug all other USB devices first.
- Plugin the FTDI device in Monochron with the controller chip and USB port facing down, and the settings jumpers facing up. When plugged in and looked at from above you will NOT see the controller chip or the USB port.
- When the USB FTDI device is the only USB terminal device connected to your machine it will map to logical device /dev/ttyUSB0.
  If you do need other USB terminal devices you need to verify which logical /dev/ttyUSB*x* 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 appear.
  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 similar commands for a toolchain when installed on 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 30 KB, a firmware verification error is reported at the 30 KB memory address. See the example below. Don’t be surprised when your clock hangs soon after being 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, execute command script files, and run a stubbed Monochron application ahead of building and uploading actual firmware.

5.2 Starting mchron

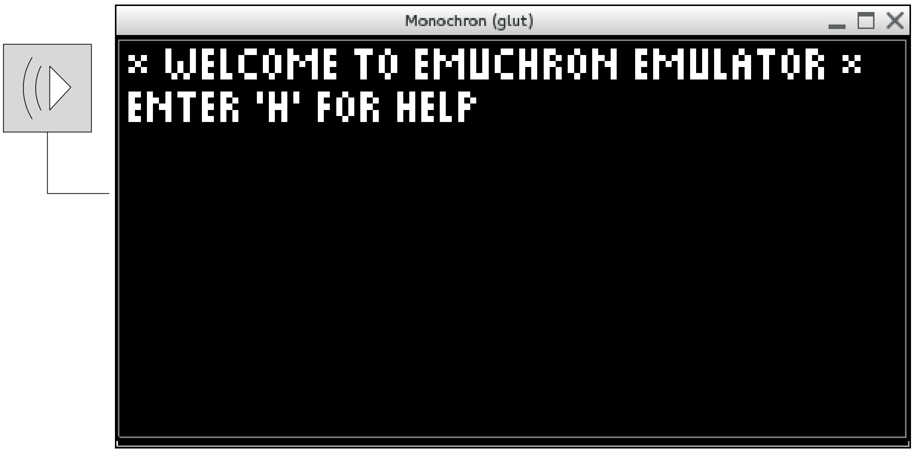
For building `mchron` refer to section 4.2. Find below an excerpt from the help file as found in help.txt [support].

```
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 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 emulator command line tool mchron v4.1 ***

process id  : 3824
ncurses tty : /dev/pts/1

time   : 16:41:35 (hh:mm:ss)
date   : 22/05/2018 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off

enter 'h' for help
mchron>
```

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

Note: In the event that the LCD stub display remains empty for a very long time while also lacking an audible beep and mchron command prompt refer to section 3.2.2.

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 operations requested by end-user commands or incorrect 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 mchron at command prompt level using command 'x' or '<ctrl>d' on an empty line.
  Example:

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

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

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

- Stop mchron 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 mchron to coredump.

5.4 Pre-emptive coredump of mchron

Mchron coredumps itself when it detects an invalid operation on glcd graphics, LCD controller or eeprom level.

Taking glcd graphics as an example, it is very well possible to enter an mchron 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 such an attempt is made, mchron 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 mchron 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 mchron: `ulimit -c unlimited`

Refer below for an example.

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

*** invalid graphics api request in glcdBufferRead()
api info (controller:x:y:data) = (0:121:7:48)
*** registered variables
registered variables: 0
*** debug by loading coredump file (when created) in a debugger
Aborted (core dumped)
$ ls -l core
-rw------- 1 user user 1912832 Oct 28 09:41 core
$
```

5.5 The mchron stack trace

When executing commands from a command file or multi-command input, mchron 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.

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

Press 'q'

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

5.6 Recovering from command syntax and parse errors

Whenever mchron 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.

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

5.7 The mchron command line history log

Standard readline library functionality is used by mchron for command line input and caching, and flushing the cache to a command line history file.

The default command history log file is $HOME/.mchron_history that is created by mchron when not present. Clear the history by stopping mchron and then remove 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      "/.mchron_history"
#define READLINE_MAXHISTORY   250
```

Some examples of functionality provided by the readline library:
Browse the command log using the up/down arrows. Navigate in a command line using the left/right arrows, or '<ctrl>a' and '<ctrl>e' for respectively the start and end of a command line. Start a reverse-order search using '<ctrl>r'.

5.8 The mchron command groups

The syntax structure of an mchron command is simple.

<command> <arg$_1$> <arg$_2$> .. <arg$_n$>

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' and underscores '_'.
- 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 command arguments. Command handlers are responsible for casting numeric expression evaluator results, being of type `double`, into an integer type. For this, casting macros are provided 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.
- Commands 'hc' and 'vp' use a regular expression pattern argument. For more information on regular expressions refer to the many web resources available.
  A rough and incomplete description of useful regular expression meta characters are:

<table>
<tr><th>Meta Character</th><th>Description</th></tr>
<tr><td>'^', '$', '|'</td><td>Start and end of string, and logical 'or'.</td></tr>
<tr><td>'[' with ']' and '-'</td><td>Start and end of a list. The '-' within a list denotes a range.</td></tr>
<tr><td>'(' with ')'</td><td>Combine start and end.</td></tr>
<tr><td>'.', '*', '+'</td><td>Any single character, repeat 0..n, and repeat 1..n.</td></tr>
</table>

  Some regular expression examples for searching mchron variable names:

Regexp	*Description*
ab	All variables containing the sequence 'ab'.
^ab	All variables starting with 'ab'.
^ab$	Variable 'ab'.

Regexp	***Description***
[a-fv].$	All variables where the last but one character is either between 'a' and 'f' or a 'v'.
^ab|v	All variables that start with 'ab' or contain a 'v'.
^(ab|vr.)$	Variable 'ab' or any three character variable starting with 'vr'.

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
          Argument: <comments>
              comments: optional ascii text
```

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  : 28/10/2017 (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  : 28/10/2017 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : on
mchron> # Toggle alarm switch
mchron> at
time  : 17:03:55 (hh:mm:ss)
date  : 28/10/2017 (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.
- As of Debian 8, and when used as a VM, short audio pulses from the Linux ALSA audio system are clipped. Emuchron provides a configurable workaround for this, as described in section 3.7.1.
- The quality of the actual piezo speaker is worse than miserable. It has only a 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 = example, 2 = analogHMS, 3 = analogHM,
                     4 = digitalHMS, 5 = digitalHM, 6 = mosquito, 7 = nerd,
                     8 = pong, 9 = puzzle, 10 = slider, 11 = cascade,
                     12 = speed, 13 = spider, 14 = thermometer, 15 = traffic,
                     16 = bar, 17 = cross, 18 = line, 19 = pie, 20 = bigdigOne,
                     21 = bigdigTwo, 22 = qrHMS, 23 = qrHM, 24 = 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 generated timestamp during execution of the 'cf' and 'm' emulator commands. This is per design and allows the user to switch between clocks displaying the same time, making it easier to compare them. 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 clock cycle mode using keypress 'c', audible alarm is temporarily stopped. Audible alarm resumes upon switching back to normal mode.
- After entering single cycle mode the user can use keypress 'p' to execute a single application clock cycle after which its glcd and controller statistics are printed. This allows quantifying the graphics interface impact of a clock.
- 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 19, 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 2
mchron> # Start this clock in a testbed environment
mchron> cf n
emuchron clock emulator:
  c = execute single application clock 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'.

Single application clock cycle 'c' keypress specifics:

- Keypress 'c' results in executing the next application clock cycle.
- Keypress 'p' results in executing the next application clock cycle after which the glcd and controller statistics for the cycle are printed. For details on glcd and controller statistics refer to section 5.8.13.
- Any other key will resume normal application clock cycle execution.

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 20*xx*.
- 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.
- The 'ds' command verifies whether the requested date is valid. For example, date April 31st will be rejected.

Example:

```
mchron> # Set our own date to Jan 27 2018
mchron> ds 27 1 18
time  : 17:08:10 (hh:mm:ss)
date  : 27/01/2018 (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  : 28/10/2017 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : off
mchron> # September 31 does not exist
mchron> ds 31 9 17
date? invalid
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 for containing a valid command in order to have it linked to an mchron command dictionary entry. However, at loading time no check is made whether command arguments are complete and valid. Command argument validation is performed at execution time.
- The maximum depth level of nested command files is set by `#define CMD_FILE_DEPTH_MAX` in interpreter.h [firmware/emulator].
- The echo argument value 'e' indicates that all commands, accompanied by its 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 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 3
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 utilities.

```
Commands:
  'h'   - Help
  'hc'  - Show command details
          Argument: <pattern>
              pattern: mchron command name regexp pattern, '.' = all
  'he'  - Show expression result
          Argument: <number>
              number: expression
  'hm'  - Show help message
          Argument: <message>
              message: optional ascii text
```

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 mchrondict.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()
------------------------
registered commands: 1
mchron> # Print the result of an expression
mchron> he sin(pi/6)
0.500000
mchron> hm Provide help message to end user while executing script
Provide help message to end user while executing script
mchron>
```

5.8.8 'i' – If

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

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

Usage specifics:

- An if-then-else construct start with an 'iif' (if) 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 'l' – LCD

The LCD commands allow interacting with the LCD controllers, erase or inverse the LCD contents, set the LCD backlight brightness and set graphics options for the ncurses and glut LCD devices.

```
Commands:
  'lbs' - Set lcd backlight brightness
          Argument: <backlight>
              backlight: 0..16 (0 = dim .. 16 = bright)
  'lcr' - Reset controller lcd cursors
  'lcs' - Set controller lcd cursor
          Arguments: <controller> <x> <yline>
              controller: 0, 1
              x: 0..63
              yline: 0..7
  'lds' - Switch controller lcd display on/off
          Arguments: <controller-0> <controller-1>
              controller-0: 0 = off, 1 = on
              controller-1: 0 = off, 1 = on
  'le'  - Erase lcd display
  'lgg' - Set glut graphics options
          Argument: <pixelbezel> <gridlines>
              pixelbezel: 0 = off, 1 = on
              gridlines: 0 = off, 1 = on
  'li'  - Inverse lcd display
  'lng' - Set ncurses graphics options
          Argument: <backlight>
              backlight: 0 = off, 1 = on
  'lp'  - Print controller state/registers
  'lr'  - Read controller lcd data in variable
          Arguments: <controller> <variable>
              controller: 0, 1
              variable: word of [a-zA-Z_] characters
  'lss' - Set controller lcd start line
          Arguments: <controller-0> <controller-1>
              controller-0: 0..63
              controller-1: 0..63
  'lw'  - Write data to controller lcd
          Arguments: <controller> <data>
              controller: 0, 1
```

Usage specifics:

- The 'li' command will, next to inversing the contents of the LCD display, also swap the LCD foreground and background colors. This will make clocks and graphics functions automatically swap their painting behavior.
- In case fast switching of backlight brightness causes display update performance issues in an ncurses LCD device, the 'lng' command allows to disable variable backlight. By default, variable backlight is enabled. Upon disabling variable backlight, the ncurses LCD device will default to full backlight brightness.
- The 'lgg' command enables drawing pixel bezels in the OpenGL2/GLUT LCD device, but will only do so after a certain minimum Monochron window pixel width to avoid blurred pixels.
  The minimum width is set by `#define GLUT_PIXBEZEL_WIDTH_PX` in lcdglut.c [firmware/emulator].
- The ks0108.c [firmware] module keeps a software administration of the controller y cursor for graphics speed optimization purposes. However, controller command 'lcs' make changes to the actual controller cursor that bypasses this software cursor administration. This means that after using the 'lcs' command, subsequent high-level graphics commands may position graphics on the wrong y position. This is not a bug.
  If needed, use command 'lcr' to reset the controller cursors to the top-left position while resyncing with the software controller cursor administration.

Example:

```
mchron> # Paint a clock so we have something on the LCD display
mchron> cs 11
mchron> # Set LCD backlight brightness to a medium setting
mchron> lbs 8
mchron> # Disable variable backlight in the ncurses LCD device
mchron> lng 0
ncurses backlight: off
mchron> # Enable pixel bezels and gridlines in the glut LCD device
mchron> lgg 1 1
glut pixel bezel: on
glut gridlines: on
mchron> # Inverse LCD display and inverse back
mchron> li
mchron> li
mchron> # Set display offset only for right side and switch back to normal
mchron> lss 0 25
mchron. lss 0 0
mchron> # Switch left side of the LCD display off and switch it on again
mchron> lds 0 1
mchron> lds 1 1
mchron> # Set cursor in controller 0 and write a byte to the LCD at bottom left
mchron> lcs 0 0 7
mchron> lw 0 0x55
mchron> # Read the contents of that location. Note that excuting a sequence
mchron> # of controller reads requires two reads for obtaining the first byte.
mchron> lcs 0 0 7
mchron> lr 0 myLcdByte
myLcdByte=254
mchron> lr 0 myLcdByte
myLcdByte=85
mchron> # Reset and resync hardware and software controller cursors
mchron> lcr
mchron> # Erase the display
mchron> le
mchron>
```

Command 'lp' prints the state and the stubbed hardware registers for each of the controllers.

```
mchron> # Print controller state and stubbed hardware registers
mchron> lp
ctrl-0 : state=write, display=1, startline=0
       : x=31, y=5, write=138 (0x8a), read=142 (0x92)
ctrl-1 : state=write, display=1, startline=0
       : x=8, y=4, write=54 (0x36), read=200 (0xc8)
mchron>
```

The content of a controller report is as follows:

Item	***Description***
display	Indicates whether the display is on or off. Note that even when the display is off, LCD contents are refreshed when writing to the LCD. Use command 'lds' to set its value.
read	The data result of the last LCD read operation on the controller. Use command 'lr' to read directly from the LCD controller.
startline	The current value of the LCD display line offset. Use command 'lss' to set its value.
state	The current machine state of the controller. For more information on the implemented controller finite state machine refer to controller.c [firmware/emulator].
write	The data of the last LCD write operation on the controller. Use command 'lw' to directly write data to the LCD controller.
x	The current cursor x position. Use command 'lcs' to set its value.

Item	*Description*
y	The current cursor y line (containing 8 vertical pixels) position. Use command 'lcs' to set its value.

Table 18: Controller state and register values

5.8.10 'm' – Monochron

The Monochron command will run an emulated Monochron application.

```
Command:
  'm'   - Run Monochron application
          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 mchron 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 preceding 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.
– After entering single application clock cycle mode the user can use keypress 'p' to execute a single application clock cycle after which its glcd and controller statistics are printed. This allows to quantify the graphics interface impact of a clock.

Example:

```
mchron> # Run the emulated Monochron application
mchron> m n k
emuchron monochron emulator:
  c = execute single application clock 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'.

Single application clock cycle 'c' keypress specifics:

– Keypress 'c' results in executing the next application clock cycle.
– Keypress 'p' results in executing the next application clock cycle after which the glcd and controller statistics for the cycle are printed. For details on glcd and controller statistics refer to section 5.8.13
– Any other key will resume normal application clock cycle execution.

5.8.11 'p' – Paint

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

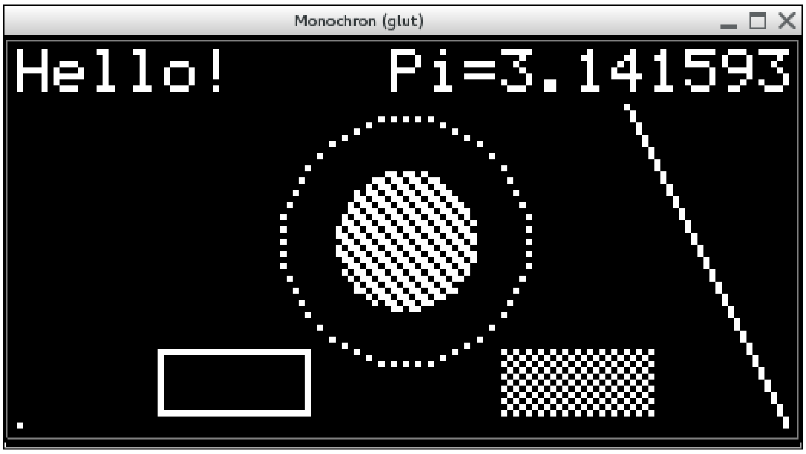
```
Commands:
  'pa'  - Paint ascii
          Arguments: <color> <x> <y> <font> <orientation> <xscale> <yscale>
                     <text>
              color: 'f' = foreground, 'b' = background
              x: 0..127
              y: 0..63
              font: '5x5p' = 5x5 proportional, '5x7n' = 5x7 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: 0..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: 0..31
              pattern: 0 = full, 1 = half, 2 = 3rd up, 3 = 3rd down
                       4 = inverse, 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' and '%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 11 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 62 1 5x7n h 1 1 pi Pi=%f
hor px=66
mchron> # Paint rectangle
mchron> pr f 24 50 25 11
mchron> # Paint filled rectangle
mchron> prf f 80 50 25 11 0 1
mchron>
```

These commands will produce the following output.



5.8.12 'r' – Repeat

The repeat commands implement a command block 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 3
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, the stubbed controller and Emuchron LCD stub device(s).

```
Commands:
  'sp'  - Print application statistics
  'sr'  - Reset application statistics
```

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=4132, outTime=6, singleCycle=0
         avgSleep=73 msec, minSleep=72 msec
glcd   : dataWrite=81087, dataRead=57411, setAddress=18797
ctrl-0 : write=32025 (41%), read=19707 (79%), display=14 (0%)
       : x=8465 (93%), y=3488 (100%), startline=14 (0%)
ctrl-1 : write=49062 (40%), read=37704 (83%), display=14 (0%)
       : x=12820 (96%), y=5522 (100%), startline=14 (0%)
glut   : lcdByteRx=33040, bitEff=24%
         msgTx=33041, msgRx=33041, maxQLen=1638, avgQLen=96
         redraws=343, cycles=9242, updates=343, fps=29.4
ncurses: lcdByteRx=33040, 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 an application clock cycle as defined by `#define ANIM_TICK_CYCLE_MS` in monomain.h [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 shortest clock cycle sleep based on the duration of the cycle that took most time to complete. Only cycles that are completed as being inTime are taken into account for calculating its value.

KPI	***Description***
outTime	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 will 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. Note: As the ncurses LCD interface runs in the same thread as mchron, flushing the ncurses display will have a negative impact on the clock cycle performance. Note: Emulator cycles that are administered under KPI singleCycle are not taken into account for calculating the outTime KPI. 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 while a clock plugin is perfectly able to complete its clock cycle well within the given timeframe.
singleCycle	The number of executed single application clock cycles as invoked by emulator command argument <mode> or emulator keypress 'c'.

Table 19: Emuchron stub statistics

The statistics for the glcd interface are as follows:

KPI	***Description***
dataWrite	The number of pixel bytes written to the LCD. It is administered by counting the number of calls to `glcdDataWrite()`.
dataRead	The number of pixel byte read operations from the LCD. It is administered by counting the number of calls to `glcdDataRead()`. Note: This number does not fully represent the actual number of LCD pixel bytes read. After a write operation or when modifying an LCD controller cursor, the hardware requires two sequential read operations of which the first is a dummy read.
setAddress	The number of explicit requests to set the LCD display cursor. It is administered by counting the number of calls to `glcdSetAddress()`. Note: Upon calling a `glcdDataWrite()` or a non-dummy `glcdDataRead()` the internal controller LCD display cursor is automatically incremented to the next LCD byte. The automatic hardware increment action is not included in this KPI.

Table 20: Monochron glcd interface statistics

Monochron has two ks0108 LCD controllers. The statistics for each of the stubbed controllers are as follows:

KPI	***Description***
display (%)	The number of commands to switch on/off the LCD. The percentage indicates the number of commands that actually lead to a change.
read (%)	The number of LCD read operations. The percentage indicates the number of read operations that actually return a proper value. Executing a sequence of read operations requires two read operations for obtaining the first byte. The first read operation in a sequence of reads is a dummy read and will lower the percentage value. In essence, the lower the percentage, the higher the number of read sequence operations are executed.

KPI	***Description***
startline (%)	The number of commands to set the LCD display start line. The percentage indicates the number of commands that actually lead to a change.
write (%)	The number of LCD write operations. The percentage indicates the number of write operations that actually lead to a change in the LCD.
x (%)	The number of commands to set the x cursor in the LCD. The percentage indicates the number of commands that actually lead to a change of the x cursor.
y (%)	The number of commands to set the y cursor in the LCD. The percentage indicates the number of commands that actually lead to a change of the y cursor.

Table 21: Monochron controller statistics

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

KPI	***Description***
avgQLen	This KPI is calculated by dividing KPI msgRx by updates. It gives the average length of the GLUT message queue to be processed.
bitEff	The percentage of bits in a processed LCD byte that will lead to a change in the LCD display. In the example above, out of 8 bits/pixels per byte, on average about 2 pixels per LCD byte will lead to a change in 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 ~30.1 fps. In practice this will be lower due to the processing power needed to process the GLUT message queue and redraw its window, in combination with latency caused by VM hypervisors and the Linux thread and process scheduler.
lcdByteRx	The number of LCD bytes (with 8 pixel bits) that are received in the interface. The interface will, via the controller, only receive LCD bytes that lead to a change on the LCD.
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 3 higher than the lcdByteRx KPI. This is explained by a number of controller and backlight commands 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, change the display and startline registers and shutting down the GLUT interface. In the example above notice that msgTx and msgRx are identical, which is normally the case. They may differ when statistics are reset while GLUT messages are still waiting to be processed.

KPI	Description
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 GLUT decides a window refresh is required, for example when the window is resized, an internal GLUT redraw event is signaled.
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 22: 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 implement statistics administration.

KPI	Description
bitEff	The percentage of bits in a processed LCD byte that will lead to a change in the LCD display. In the example above, out of 8 bits/pixels per byte, on average about 2 pixels per LCD byte will lead to a change in the LCD display.
lcdByteRx	The number of LCD bytes (with 8 pixel bits) that are received in the interface. The interface will, via the controller, only receive LCD bytes that lead to a change on the LCD.

Table 23: 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 4
mchron> # Print the current time/date/alarm (clock layout is not updated)
mchron> tp
time  : 11:10:55 (hh:mm:ss)
date  : 30/10/2017 (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  : 30/10/2017 (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  : 30/10/2017 (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  : 30/10/2017 (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: <pattern>
              pattern: variable name regexp pattern, '.' = 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' and '_', excluding reserved function and constant keywords.
  Examples: x (=ok), radius (=ok), my_Local_Var (=ok), a1 (=bad), abc$ (=bad), true (=bad)
- Variables must explicitly have been 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
registered variables: 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
registered variables: 2
mchron> # Reset all variables
mchron> vr .
mchron> vp .
registered variables: 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 (msec)
          When waiting, a 'q' keypress will return control back to the mchron
          command prompt
  'wte' - Wait for wait timer expiry
          Argument: <expiry>
              expiry: 1..1000000 (msec)
          When waiting, a 'q' keypress will return control back to the mchron
          command prompt
  'wts' - Start wait timer
```

Usage specifics:

- The wait 'w' command supports two flavors. One flavor will wait a dedicated amount of time and another waits for an end-user keyboard keypress.
- For waiting time within loops, such as in script time-hm.txt [script], wait timer commands are a more reliable means than the 'w' command.
  The 'wts' command starts a new timer. Next, when executing the 'wte' command, the expiry argument defines the amount of time to wait after 'wts' was invoked, making it independent of the time required to execute other mchron commands prior to executing the 'wte' command. This mechanism also negates display update time differences between the OpenGL2/GLUT and ncurses LCD devices, resulting in identical graphic update-per-second behavior between these LCD devices.
- When 'wte' completes waiting for a timer to expire, it will restart the wait timer for the next 'wte' iteration based on adding the expiry argument to the timer start timestamp. This results in very reliable timer behavior over a longer period of time.
- When the expiry timer in 'wte' has already expired, 'wte' will complete immediately, but will restart the timer based on 'now' for the next 'wte' iteration.
- When 'wte' is interrupted using a 'q' keypress while waiting for the timer to expire, the timer is not restarted.
- The wait commands are used in many scripts to temporarily halt script execution or wait a while after updating the LCD display with new information.

Example:

```
mchron> # Wait one second
mchron> w 1000
mchron> # Wait for keypress (don't forget to press a button)
mchron> w 0
<wait: press key to continue>
mchron> # Start a timer
mchron> wts
mchron> # Wait a few seconds...
mchron> # Then wait the remaining time of 10 seconds after starting the timer
mchron> wte 10000
mchron> # Then wait a few more seconds...
mchron> # Then wait the remaining time of 7.5 seconds after restarting the timer
mchron> wte 7500
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 mchron 'hello world!' command

Mchron 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 mchron purposes.

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

```
1 mchron> pa f 2 8+7 5x5p h 3 2 hello world!
                                        // cmdArgValidateNum(): x <= 127
                                        cmdArgDomain_t argNumPosX =
                                        { DOM_NUM_MAX, "", 0,
                                          GLCD_XPIXELS - 1, NULL};
2 cmdLineExecute():                     // cmdArgValidateChar(): 'b' or 'f'
  Execute a single command line         cmdArgDomain_t argCharColor =
                                        { DOM_CHAR, "bf", 0,
                                          0, "b = background, f = foreground" };
3 cmdArgInit(): Init scan + get command and associated command dictionary

cmdArg_t argCmd[] =             cmdCommand_t cmdGroupPaint[] =
{ { ARG_WORD, "command",        { { "pa", PC_CONTINUE, CMDARGS(argPaintAscii),
    &argInfoCommand } };            CMDHANDLER(doPaintAscii),  "paint ascii" },
                                    .. };
4 cmdArgGet(): Get command arguments using command dictionary + verify
               domain non-numeric arguments
cmdLine_t cmdLine.args[] =      cmdArg_t argPaintAscii[] =
{ "f",                          { { ARG_CHAR, "color",        &argCharColor },
  "2\n",                          { ARG_UNUM, "x",            &argNumPosX },
  "8+7\n",                        { ARG_UNUM, "y",            &argNumPosY },
  "5x5p",                         { ARG_WORD, "font",         &argWordFont },
  "h",                            { ARG_CHAR, "orientation",  &argCharOrient },
  "3\n",                          { ARG_UNUM, "xscale",       &argNumScaleX },
  "2\n",                          { ARG_UNUM, "yscale",       &argNumScaleY },
  "hello world!" };               { ARG_STRING, "text",       &argInfoText } };
5 cmdArgPublish(): Publish arguments + verify domain numeric arguments

  argWord[0] = "pa"             cmdLine_t cmdLine.args[] =
  argWord[1] = "5x5p"           { "f",
                                  "2\n",
  argChar[0] = 'f'                "8+7\n",
  argChar[1] = 'h'                "5x5p",
                                  "h",
  argDouble[0] = 2                "3\n",
  argDouble[1] = 15               "2\n",
  argDouble[2] = 3                "hello world!" };
  argDouble[3] = 2

  argString = "hello world!"
6 Command "pa" handler = doPaintAscii(): Paint text string

 len = glcdPutStr3(TO_U08(argDouble[0]), TO_U08(argDouble[1]), font,
         argString, TO_U08(argDouble[2]), TO_U08(argDouble[3]), color);
```

Figure 8: Processing an mchron 'hello world!' command

Step 1:
The user enters a 'pa' (paint ascii) command using the keyboard, or has it prepared in an mchron command file. The command is copied into a command line structure `cmdLine_t` as defined in interpreter.h [firmware/emulator.]

Step 2:
Main command processing takes place in `cmdLineExecute()` in listutil.c [firmware/emulator]. All remaining steps are initiated from within this function.

Step 3:
The command scanning process is started by calling `cmdArgInit()` in scanutil.c [firmware/emulator]. This function will scan the command name based on an `cmdArg_t` structure. For this, 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. When the command is scanned its command dictionary entry is retrieved.
From this point on all scanning and parsing functionality will be based on information accessed from the command dictionary entry.

Step 4:
In `cmdArgGet()` in scanutil.c [firmware/emulator] the remaining part of the command will be scanned, parsed and processed, based on the command dictionary for the 'pa' command. Each argument is copied into a dynamic array within the `cmdLine_t` structure. For character and word arguments 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'. 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 6 below.
Note that for numeric arguments only the expression is copied, including a newline character, which is per expression evaluator requirement. Evaluation and domain checking for this type of arguments is done in the next step.

Step 5:
In `cmdArgPublish()` in scanutil.c [firmware/emulator] all the command arguments are published into dedicated argument arrays for words, characters, doubles and a string. They are respectively `argWord[]`, `argChar[]`, `argDouble[]` and `argString`. In the example above, the `ARG_WORD` font argument is added in `argWord[]`.
Note that numeric arguments and now run through the expression evaluator and have their domain value checked, when configured. In this case the 'x' unsigned number argument may not exceed the maximum value 127.

Step 6:
When the command line has been fully scanned, parsed and published, 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 `cmdLineExecute()`.

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 almost identical to the time-hm.txt [script] script.

```
# Demo script
cs 3
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 or enter a non-existing command name.

```
mchron> # Demo multi-line command entry via 'rf' to paint all minutes in a day
mchron> cs 3
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 as depicted in figure 9 below.

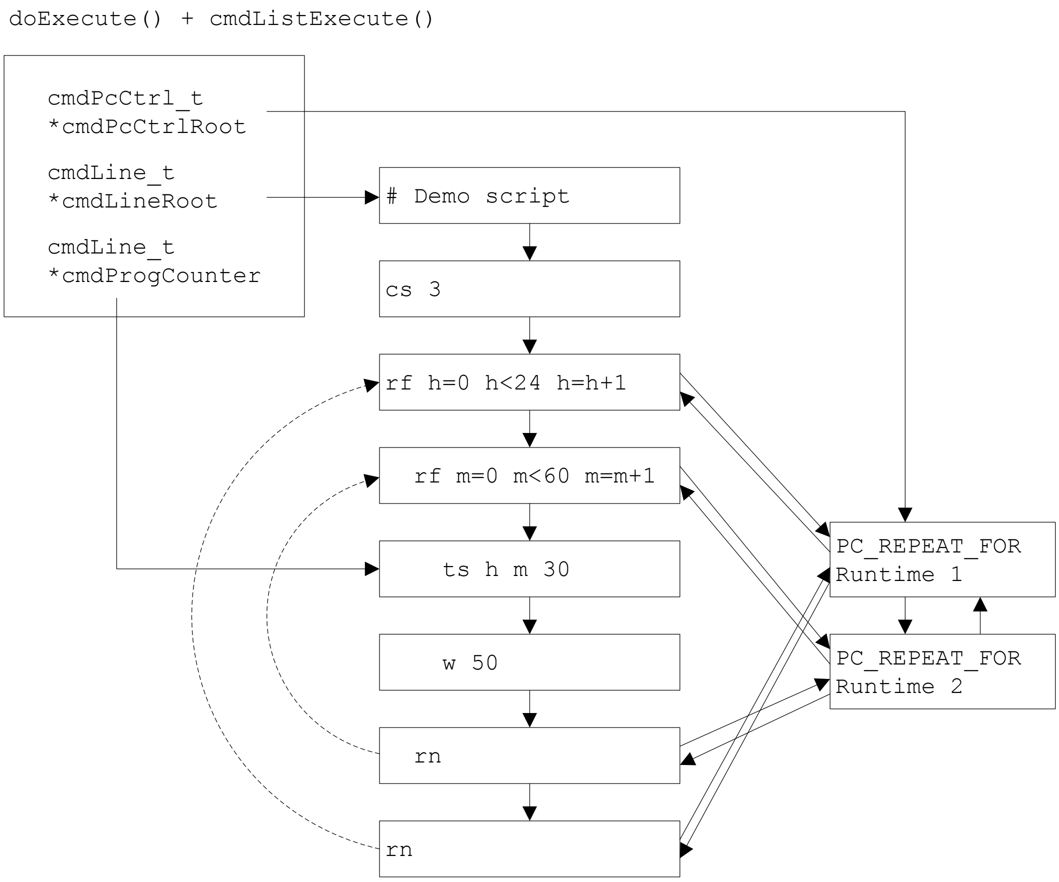


Figure 9: Creating and executing an mchron command list

Step 1: Load the file contents in linked lists.

- The 'e' command is interpreted in `cmdLineExecute()`. 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 9 above. 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`.

Step 2: Execute the commands in the linked list.

- When loaded, in `doExecute()` the linked list structure is then executed via function `cmdListExecute()`. In this function another pointer named `cmdProgCounter` is available that will serve as a list execution program counter.

- In `cmdListExecute()` 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.
- 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 process the repeat condition via its `cmdLine_t` 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 supports 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 small. 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 (re)build 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/ttyUSB*x* 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/ttyUSB*x* 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 3.5)
$ 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.

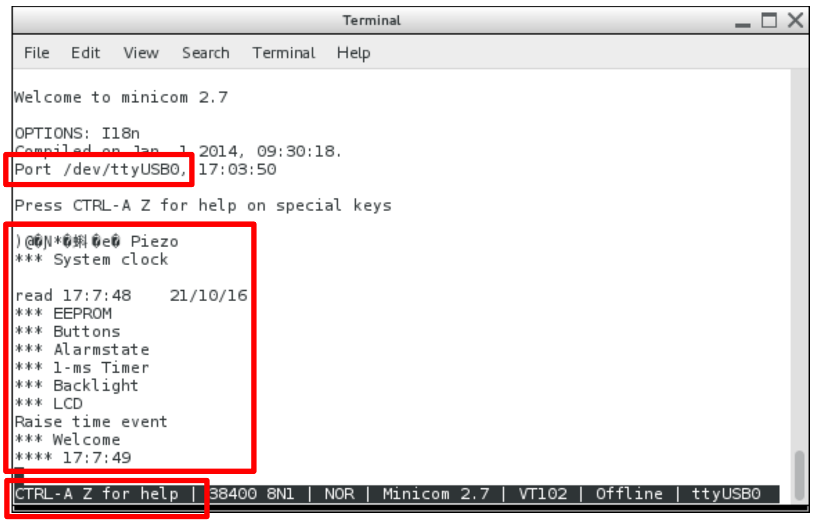


Figure 10: Minicom receiving Monochron debug string

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

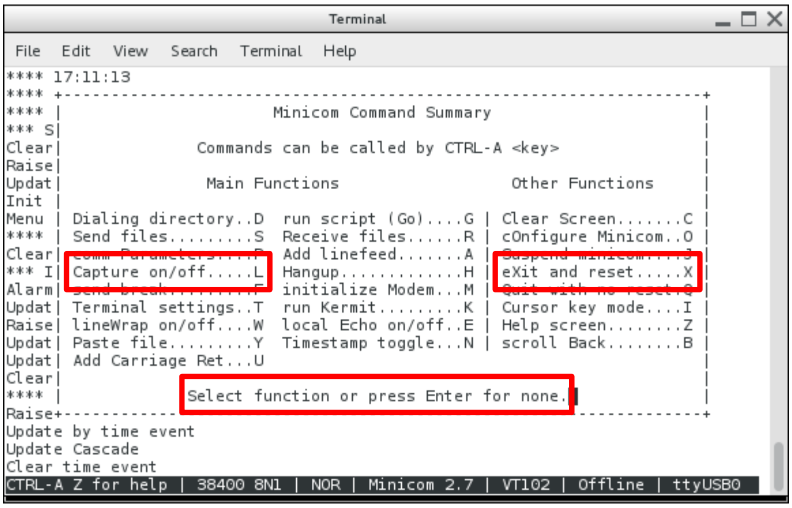


Figure 11: Minicom command summary via '<ctrl>az'

– 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, mchron must be started with the -d flag to specify the debug log output file. See below.

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

Note that if mchron is built with the master switch set to 0, mchron 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 emulator command line tool mchron v4.1 ***

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

process id  : 3201

time   : 20:18:35 (hh:mm:ss)
date   : 14/03/2018 (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

read 16:55:50  21/10/16
Clear time event
Raise time event
*** UART
*** Piezo
*** System clock

read 16:55:59  21/10/16
*** EEPROM
*** Buttons
*** Alarmstate
*** 1-ms Timer
*** Backlight
*** LCD
*** Welcome
*** Start initial clock
Clear time event
**** 16:56:0
Raise time event
Update by time event
Init Cascade
(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 application 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 and 9 when using gdb

As of Debian 8, there are gdb requirements with respect to referencing glibc sources. These requirements are described in section 3.7.2.

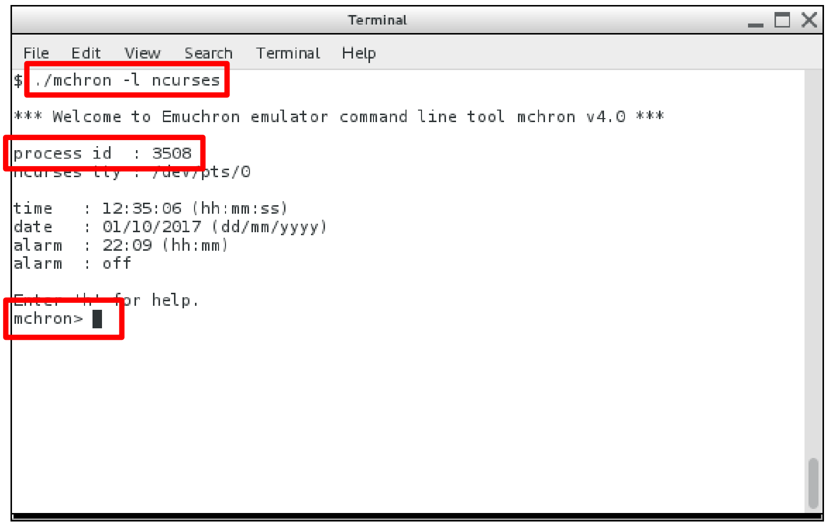
6.3.2 Limitations on using ncurses

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 is to start mchron first, 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.

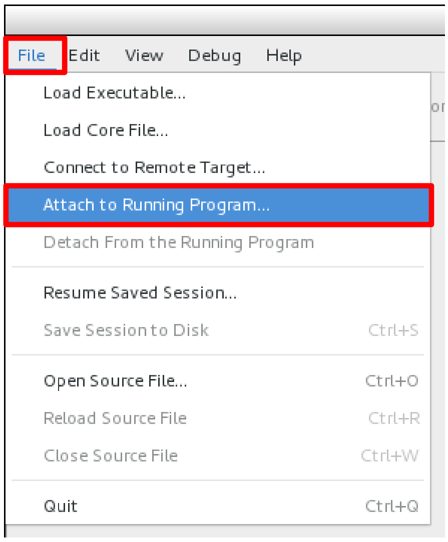
However, depending on the GUI front-end being used, different steps need to be taken. In the sections below is explained on a step-by-step basis how to get an ncurses LCD display to work in a gdb debugging session.

6.3.3 Debugging Emuchron with ncurses device using Nemiver

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

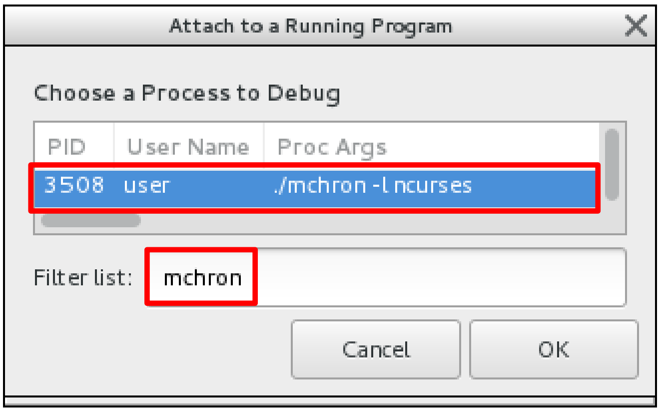


Then, start Nemiver and select "File→Attach to Running Program…" to attach to a running process.



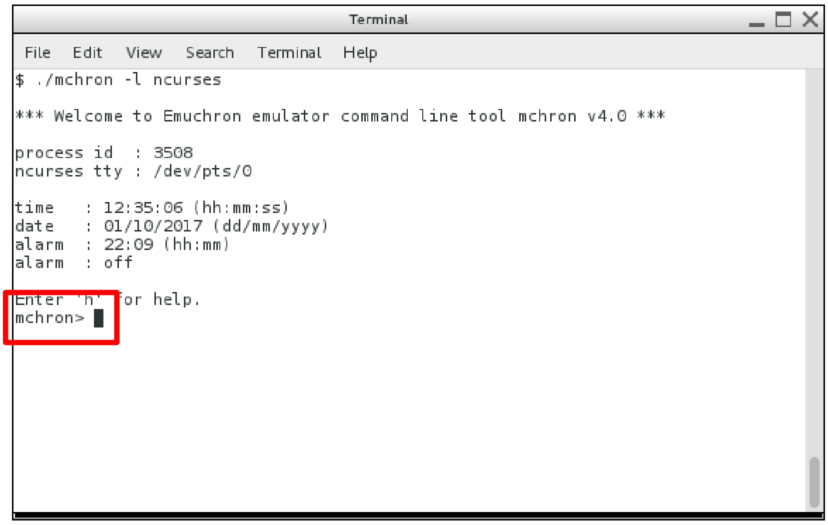
In the popup list search for the mchron command and click 'OK'.

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

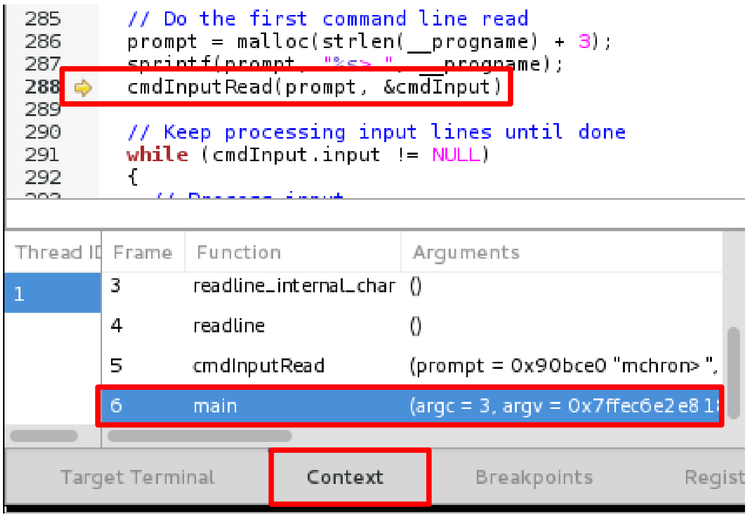


Nemiver now tries to attach to the process, but for now it cannot. The reason for this is that the mchron process is not active at this time as it waits for a command on the command line.
So, what needs to be done is to enter a blank command by hitting the return key in the mchron console. When hit, mchron now seems to hang as the mchron process is brought under control of Nemiver.

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.



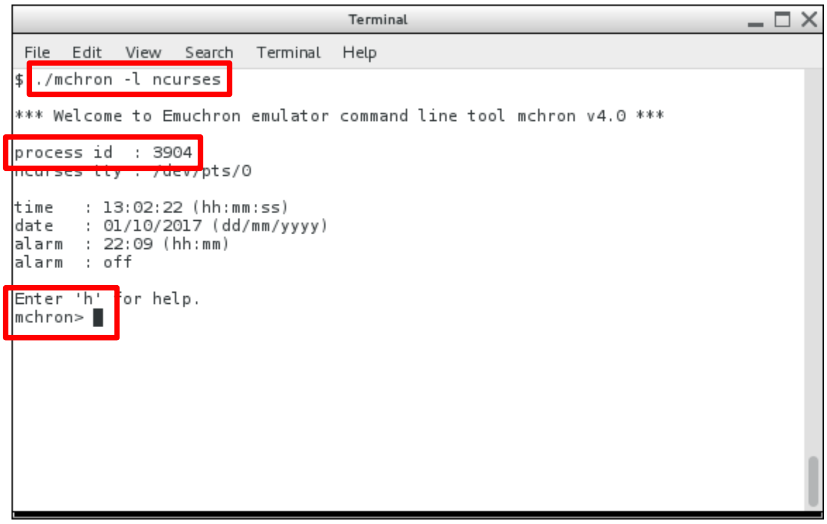
In Nemiver we are now be able to browse the application sources. The easiest way to open the first source is to go to tab 'Context' that provides the runtime call stack, and select the lowest call stack level available, which is `main()` in the mchron.c [firmware/emulator] source. See below.



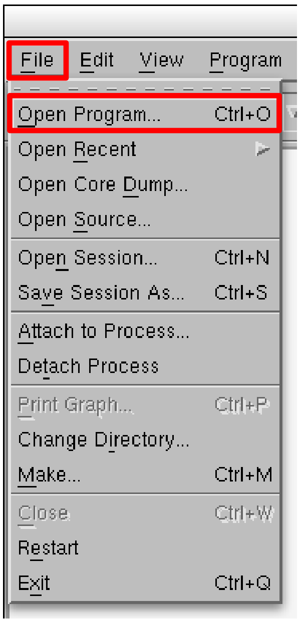
From this point on you are able to open any Emuchron source, set, disable and re-enable breakpoints, and verify local and global data.
For more information on using Nemiver use the 'Help' menu.

6.3.4 Debugging Emuchron with ncurses device using DDD

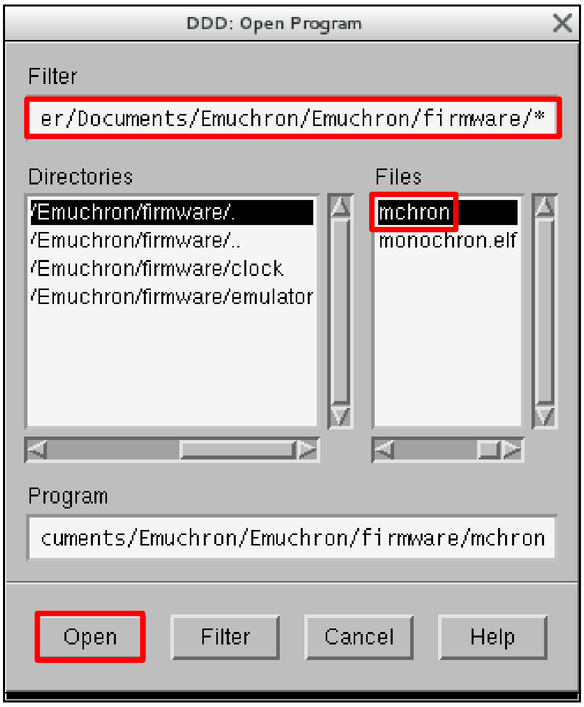
First startup mchron and make sure there is a command prompt.



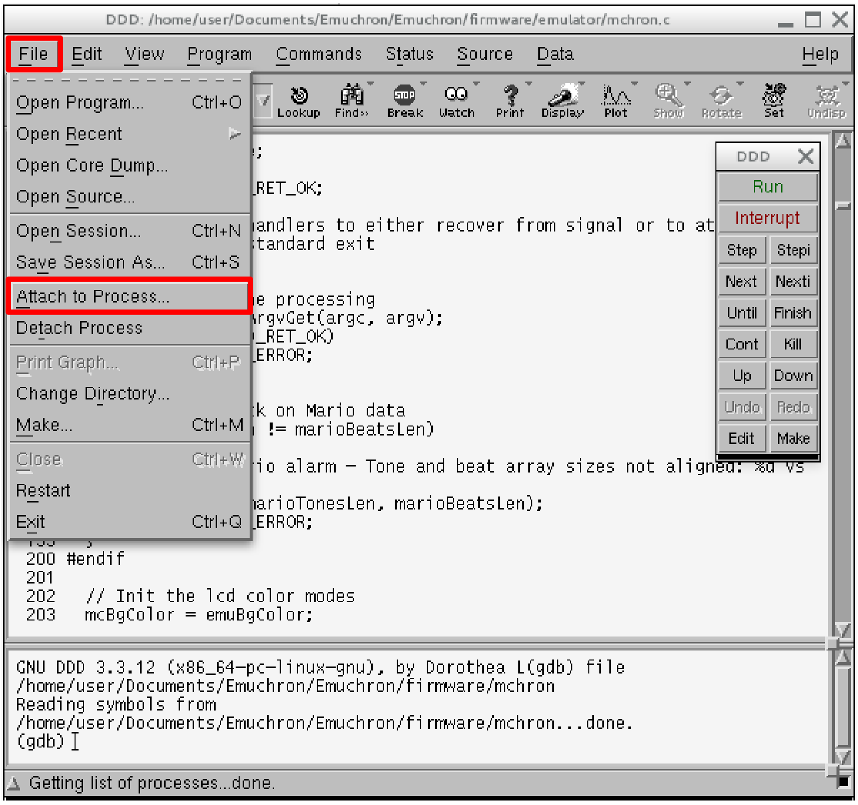
Then, start DDD and select "File→Open Program..." to open an executable program.



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

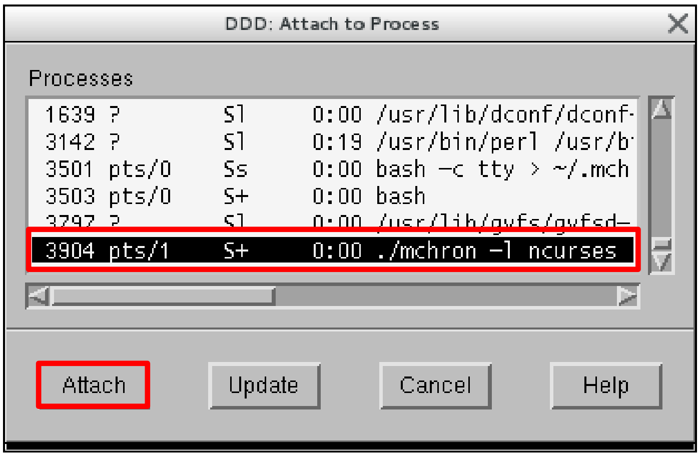


DDD should now display the mchron.c [firmware/emulator] source file, but we're not running an actual debug session yet. For this, attach to the running mchron process using "File→Attach to Process…".



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

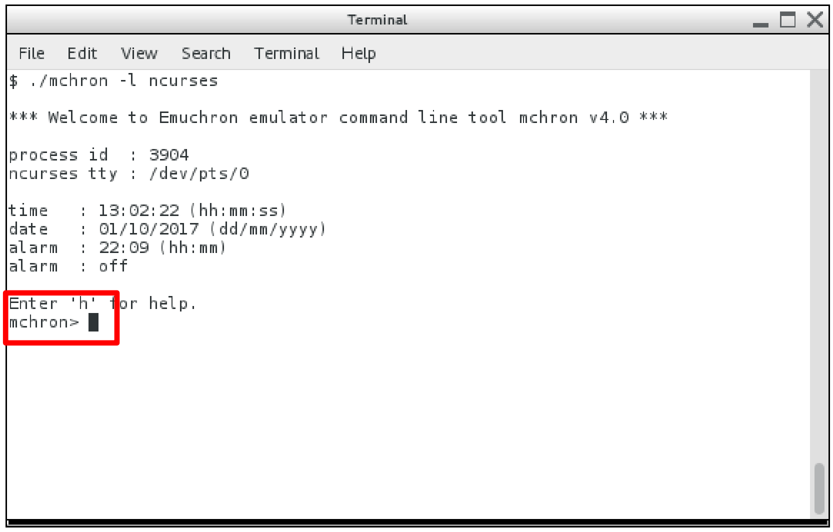
Note: You can use the reported mchron PID in the mchron shell as a cross reference in the list below.



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.

Note: When DDD fails to connect to mchron and/or kills the mchron process, refer to section 7.12.

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 debugging a coredump of another application. 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 different runtime behavior is found in `rtcMchronTimeInit()` in monomain.c [firmware]. This function requires fully dedicated code sections for Monochron and Emuchron.

7.2 Linux mathlib accuracy vs. AVR mathlib accurary

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 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 only 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` except for bit operators. As bit operators require an integer type, numbers are temporarily cast to type `unsigned int` and are cast back to type `double` upon completing the operation.
The expression evaluator 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 application clock cycle, being 75 msec that equals to a ~13.3 Hz scan frequency. This results in the smoothest possible seconds indicator behavior in a clock.

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 ~5.7 Hz, or every 175 msec. Taking into account that a single application clock cycle is 75 msec it means that it may take up to three cycles for obtaining new time information, resulting in a time update delay of 225 msec. This delay is acceptable for the original Monochron Pong clock as it 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 time to switch from one seconds value to the next.

As this was deemed unacceptable, in Emuchron the timer interrupt handler firmware was reconfigured such that the RTC scan frequency increased to ~8.5 Hz. This reduced the time update delay back to two application clock cycles, being 150 msec, which was considered acceptable.

However, starting Emuchron v3.0 the RTC scan frequency is increased to ~13.6 Hz, or about every 74 msec. This scan frequency guarantees that every application clock cycle will always include an RTC scan, leading to the lowest possible time update delay of 75 msec and therefore the smoothest possible seconds indicator behavior in a clock.
The RTC scan frequency is controlled using the following defines in monomain.h [firmware].

```
// Uncomment to implement RTC readout @ ~5.7Hz
//#define TIMER2_RETURN_1     80
//#define TIMER2_RETURN_2     6
// Uncomment to implement RTC readout @ ~8.5Hz
//#define TIMER2_RETURN_1     53
//#define TIMER2_RETURN_2     9
// Uncomment to implement RTC readout @ ~13.6Hz
#define TIMER2_RETURN_1       33
#define TIMER2_RETURN_2       14
```

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 mchron is likely to report an error on startup as the destination terminal will not meet the minimum size requirements set by the mchron application.
Note that the shell to receive ncurses output may even be the one in which mchron is started.

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

7.6 When experiencing Debian audio or graphics issues

When using a VM in VirtualBox refer to section 3.2.2 for configuring VM graphics and audio that may prevent the issues observed.
When using a VM in VMware Fusion refer to section 3.2.3 for configuring VM graphics that may prevent the issues observed. Graphics issues may include erratic ncurses screen update behavior.
Also refer to section 3.7.1 for configuring an ALSA workaround for a Debian 8 and 9 VM when playing short audio pulses.

7.7 Controller behavior and controller stub compatibility

Emuchron supports stubbed ks0108 LCD controllers using a finite state machine implemented in controller.c [firmware/emulator]. The state machine has proven to be compatible with actual hardware behavior when using the controllers as intended. This means the following:

- A sequence of read or write operations consists of first setting the cursor in the LCD controller after which a series of read or write operations are executed on the controller.
- Setting the LCD controller cursor requires setting the x position register or the y position register, or both.
  Note: In ks0108.c [firmware], the Monochron firmware sets the LCD controller cursor position by always setting the x position register, and by setting the y position register only when its position changes.
- When reading from the LCD controller, a hardware limitation requires reading the first byte with two sequential read operations. After that, each subsequent read operation will retrieve the next LCD byte. Also, the LCD controller will automatically increase the x cursor position.
- When writing to the LCD, the first write will write to the cursor location. Each subsequent write operation will write to the next LCD byte. Also, the LCD controller will automatically increase the x cursor position.
- After reading from or writing to the last x position for a controller, the controller will reset the x position to 0. The y position remains unchanged.

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 8 million times
mchron> rf x=0 x<8000000 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 on average about 13 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 5.4 for an example.

7.10 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 with generic support functions, 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 smaller, despite its enhancements.

In general, 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.

7.11 Is Debian Linux required for building firmware

No.
Only the Emuchron emulator and mchron command line tool requires Debian Linux to build and run. For building the Monochron firmware 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.

7.12 Debugger is missing "syscall-template.S" or "pselect.c"

This is an annoying Debian 8 and 9 'feature'. Apart from being annoying it may also be the root cause to DDD not being able to debug the mchron process. Although an attempt is made to fix this during the installation of required Debian packages, for more recent Debian 8 or 9 releases it may be needed to create a specific symbolic link resolving a missing link in the glibc source path. Instructions for this are found in section 3.7.2.

8 Known bugs

8.1 The mchron terminal no longer echoes characters

When mchron executes a command list or a wait command, it switches the terminal input behavior from using a readline input method, where text input is completed using 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 for interrupting command or script execution. When command or script execution has completed, mchron 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 mchron terminal.

When mchron 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 mchron 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 mchron 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 blindly typed terminal `reset` command method turns out to be very effective.

8.2 Pending characters in the mchron terminal input buffer

As explained in section 8.1, mchron 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 mchron command prompt.
This may not always be successful, especially when the end-user press-holds a key, thereby generating multiple 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 mchron command.
In case this occurs, the next mchron command is likely to fail as the remaining input buffer characters are not expected to make up a correct mchron 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

All LCD device output screendumps in this document are taken using screen capture tool Shutter. 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? For this see the mchron command session below.

- First, we start mchron using either the ncurses or glut LCD device.
- Then, five mchron commands are used to respectively (1) select the digital HMS clock, (2) set the position of the alarm switch to 'on' to make the clock display the alarm time, (3) set the date to Sep 14th 2013, (4) set the alarm to 06:45, and (5) to 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 screen capture tool Shutter. The resulting clock layout can be seen in appendix A.3.
- For other screendumps using the same date and time, select another clock using command 'cs'.

```
$ ./mchron
:
mchron> cs 4
mchron> ap 1
time  : 19:20:15 (hh:mm:ss)
date  : 04/12/2017 (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>
```

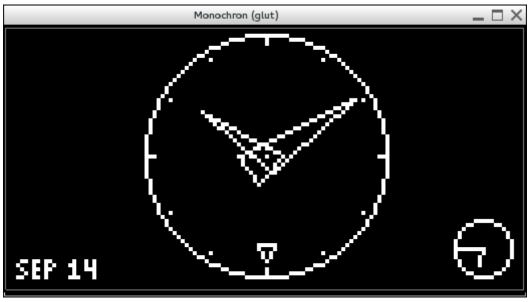
A.1 Analog clocks

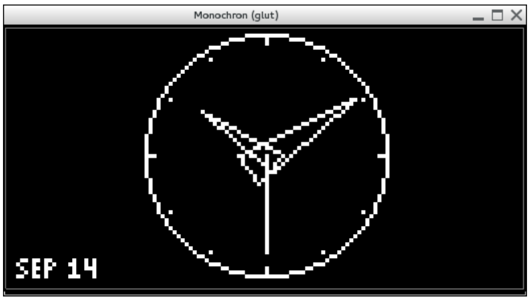
Clock Ids: `CHRON_ANALOG_HMS` and `CHRON_ANALOG_HM`

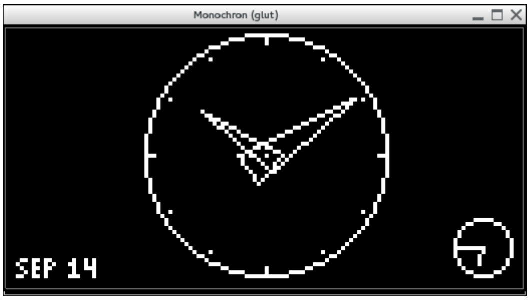
These are basic analog clocks with h/m/s or h/m time notification.
When the alarm switch is on, the alarm time will appear at the bottom right in a small analog clock. When alarming or snoozing, the alarm time will blink.
There are several build options for an analog clock, allowing eight different versions 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].



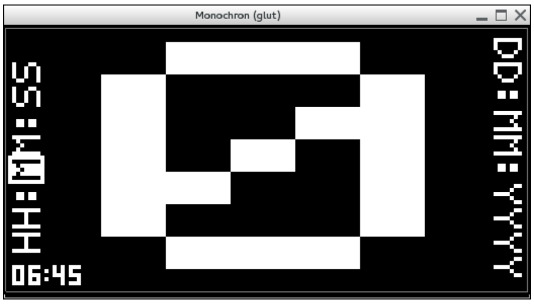


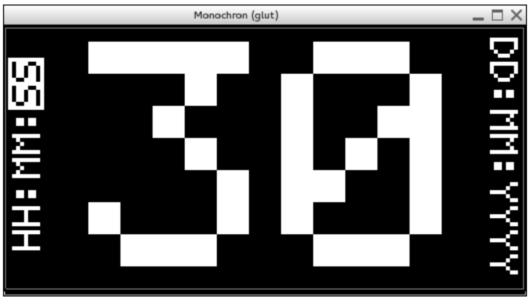


A.2 Big Digit clocks

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 are, at its lowest level, generated using a single glcd graphics function only, being `glcdPutStr3()`.
The clock has two build options. The first is to apply a 'glitch' every once in a while by randomly setting the LCD controller start line and display on/off registers. The second is to blink the bottom dot of the ':' separator in the hh:mm clock. See below.
For code refer to digital.c [firmware/clock].

```
// Uncomment if you want to apply a 'glitch' mode to the clock.
// Refer to digiPeriodSet() for setting glitch delay and duration.
//#define DIGI_GLITCH

// For the CRHON_DIGITAL_HM clock you can make the bottom dot ":" separator
// blink on a per second basis. Set the blink bezel size between 0 (no bezel)
// and 3 (thick bezel).
// Uncomment if you want to enable a blinking separator in the CHRON_DIGITAL_HM
// clock.
#define DIGI_HM_BLINK
#define DIGI_HM_BLINK_BEZEL   2
```

A.4 Example clock

Clock Id: `CHRON_EXAMPLE`

This is a very basic clock that serves as an example for those new to the Emuchron clock plugin framework. The entire clock requires about 80 lines of code, including blank lines and comments. 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 example.c [firmware/clock].

A.5 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 right.
When alarming or snoozing, the alarm time will blink.
For code refer to mosquito.c [firmware/clock].

A.6 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.7 Pong clock

Clock Id: `CHRON_PONG`

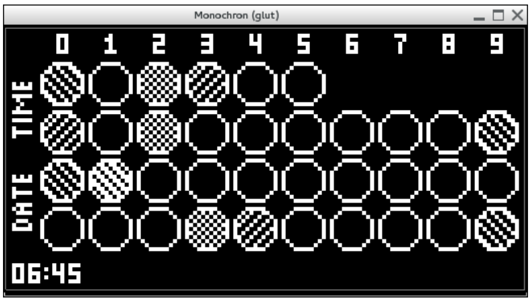
This clock is the original Monochron pong clock, but is migrated to be used in the Emuchron framework. Functionality to process time, date and alarm has been re-implemented to use the Emuchron data environment. The basic migration of the clock code took about one day of efforts.
A number of functional 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 while also allowing shallow angles. Next, 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.8 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.

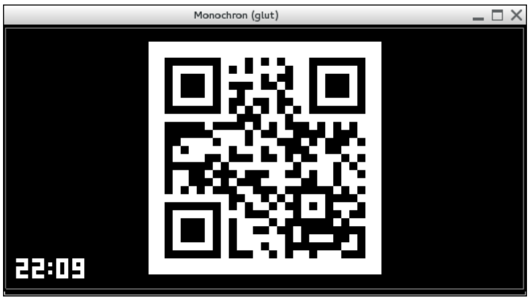
For code refer to puzzle.c [firmware/clock].

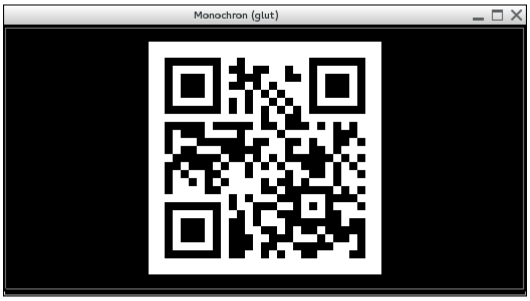


A.9 QR clocks

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.10 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.11 Spotfire and QuintusVisuals clocks

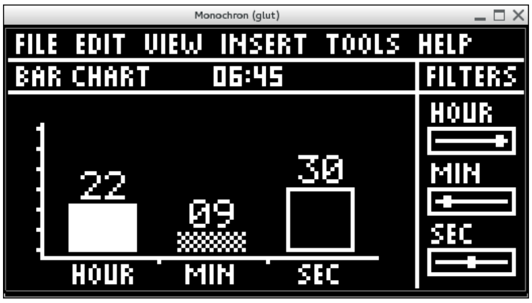
Clock Ids: `CHRON_BARCHART`, `CHRON_CASCADE`, `CHRON_CROSSTABLE`, `CHRON_LINECHART`, `CHRON_PIECHART`, `CHRON_SPEEDDIAL`, `CHRON_SPIDERPLOT`, `CHRON_THERMOMETER` and `CHRON_TRAFLIGHT`

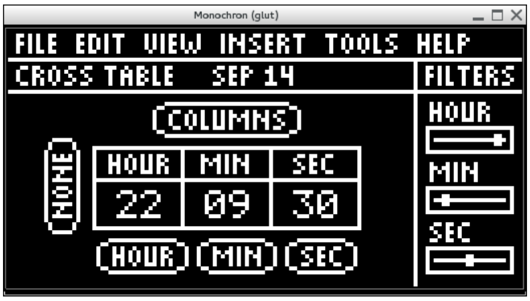
TIBCO Spotfire (http://spotfire.tibco.com) is a professional business analytics tool that provides insight in very large amounts of data using visualizations. QuintusVisuals (http://www.quintusvisuals.com/en/home) is an extension to TIBCO Spotfire and provides additional visualization types. The clocks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

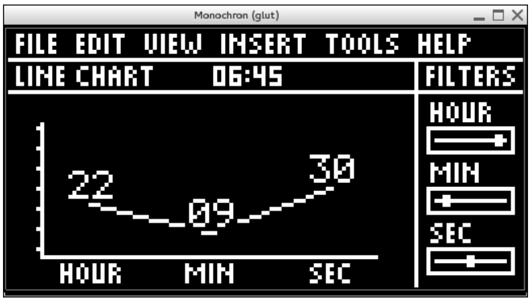
The (non-functional) header of a visualization represents the header of TIBCO Spotfire. However, the clocks include a hard-coded calendar that will change the header on specific dates to a dedicated message. See the spider plot example for March 14th below.
The filter panel on the right side contains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

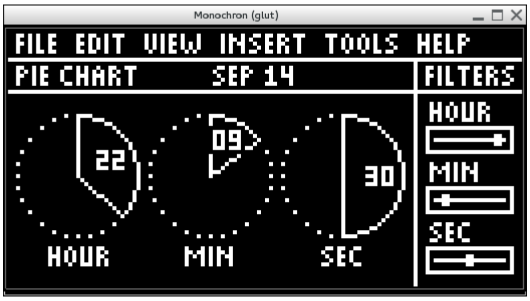
For code refer to spotfire.c (generic module for all TIBCO Spotfire and QuintusVisuals clocks, including the calendar), barchart.c, cascade.c, crosstable.c, linechart.c, piechart.c, speeddial.c, spiderplot.c, thermometer.c and trafficlight.c [firmware/clock].

The TIBCO Spotfire clocks are as follows:

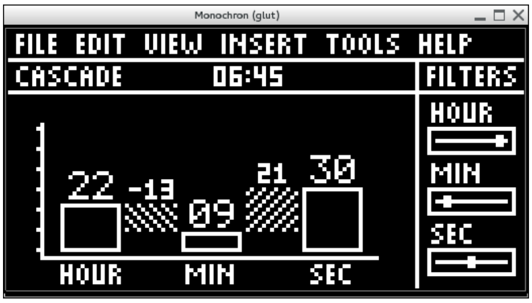


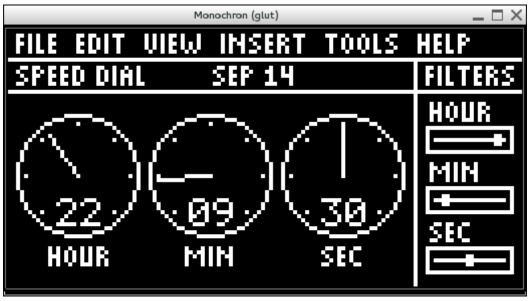


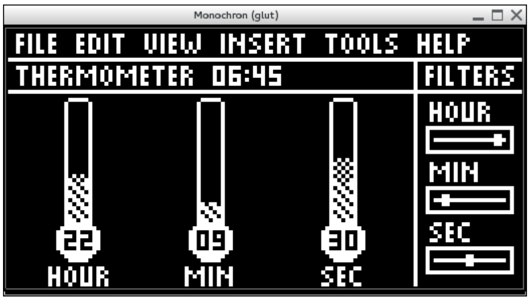


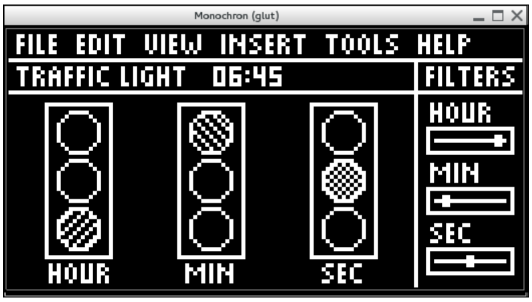


The QuintusVisuals clocks are as follows:









B High-level glcd performance tests

Over time, in Emuchron many modifications are made in its entire code base to increase glcd draw performance and/or reduce firmware size. In order to find out how modifications impact the 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/modified in subsequent releases. Source code can be found in perftest.c [firmware/clock].

Below is a table with an overview of the average draw performance achieved per high-level graphics function over time. For this, the performance test module and analog clock code of Emuchron v4.1, as well as bugfixes in glcd graphics code, were ported back to Emuchron v1.0.

High-level glcd graphics function name	***Draw performance Emuchron v1.0***	***Draw performance Emuchron v4.1***
`glcdDot()`	1.0x	2.5x
`glcdLine()`	1.0x	4.2x
`glcdCircle2()`	1.0x	4.6x
`glcdFillCircle2()`	1.0x	4.3x
`glcdRectangle()`	1.0x	4.0x
`glcdFillRectangle2()`	1.0x	4.5x
`glcdPutStr3()`	1.0x	3.1x
`glcdPutStr3v()`	1.0x	6.7x
`glcdPutStr()`	1.0x	1.6x

Table 24: Draw performance Emuchron v1.0 vs v4.1 (avr-gcc 4.8.1)

B.1 Test results Emuchron v3.1 vs v4.0

Find below an overview of version and build sizes of the firmware when built with default settings.

Version avr-gcc	***Emuchron v3.1 Object size (bytes)***	***Emuchron v4.0 Object size (bytes)***
4.8.1 (Debian 8)	.data: 988 .text: 25446 **Total: 26434**	.data: 984 .text: 25446 **Total: 26430**
4.9.2 (Debian 9)	.data: 988 .text: 25614 **Total: 26602**	.data: 984 .text: 25602 **Total: 26586**

Table 25: Default firmware size Emuchron v3.1 vs Emuchron v4.0

Some remarks on the build statistics:

– The difference between firmware size of the v3.1 and v4.0 builds is negligible. Optimized code versus new and modified code negate one another in the final build.

In Emuchron v4.0, when compared to v3.1, a test suite is added for `glcdRectangle()` that includes two tests.

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.

The graphics performance in Emuchron v4.0 is impacted by the following implemented features:
- An optimization is applied in `glcdRectangle()` for drawing single-height rectangles, which is essentially drawing a horizontal line.
- A bugfix is applied in `glcdFillCircle()`, correcting one missing vertical pixel in the bottom area of the circle. For this, the number of drawing loops in a test for this function is recalibrated to run in 2 minutes. The corrected function shows a decrease in draw performance of approx. 2%.

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 more information regarding the glcd dataWrite, dataRead and setAddress indicators refer to section 5.8.13. The indicator values listed below are those from the test run on avr-gcc 4.8.1.

Test	***Test Name***	***Emuchron v3.1***		***Emuchron v4.0***	
1	glcdDot-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:01 01:56 1048576 2097152 2097152	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:01 01:56 1048576 2097152 2097152
2	glcdDot-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:01 01:56 630784 2523136 1892352	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:01 01:56 630784 2523136 1892352
3	glcdLine-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 335819 1694889 388101	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 335818 1694888 388101
4	glcdLine-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:59 01:57 1286488 1505000 390992	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:59 01:57 1286488 1505000 390992
5	glcdCircle2-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1616058 1867044 501972	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1616058 1867044 501972
6	glcdCircle2-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1646280 2033640 774720	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1646280 2033640 774720
7	glcdFillCircle2-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:55 1513345 863471 1113986	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:55 1513345 863471 1113986
8	glcdFillCircle2-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1065120 1483560 1160220	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:57 1065120 1483560 1160220

Test	***Test Name***	***Emuchron v3.1***		***Emuchron v4.0***	
9	glcdRectangle-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:15 02:02 1592165 2146540 1076818	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:56 1511448 2042761 1032468
10	glcdRectangle-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:58 2482400 2503280 317840	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:58 2482400 2503280 317840
11	glcdFillRectangle2-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 1899240 2170560 506464	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 1899240 2170560 506464
12	glcdFillRectangle2-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3674000 1376080 100200	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3674000 1376080 100200
13	glcdFillRectangle2-03	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 4065264 1032448 40330	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 4065264 1032448 40330
14	glcdFillRectangle2-04	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3976560 1009920 39450	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3976560 1009920 39450
15	glcdPutStr3-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2197944 2232832 34888	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2197944 2232832 34888
16	glcdPutStr3-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 3030300 1539200 36075	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 3030300 1539200 36075
17	glcdPutStr3-03	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2142720 2177280 34560	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2142720 2177280 34560
18	glcdPutStr3-04	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 2887200 1761192 38496	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 2887200 1761192 38496
19	glcdPutStr3-05	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2283120 2319360 36240	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2283120 2319360 36240

Test	Test Name	Emuchron v3.1	Emuchron v4.0
20	glcdPutStr3v-01	Time 4.8.1: 02:00 4.9.2: 01:53 dataWrite: 714000 dataRead: 215900 setAddress: 178500	Time 4.8.1: 02:00 4.9.2: 01:53 dataWrite: 714000 dataRead: 215900 setAddress: 178500
21	glcdPutStr3v-02	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 2078496 dataRead: 544368 setAddress: 123720	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 2078496 dataRead: 544368 setAddress: 123720
22	glcdPutStr3v-03	Time 4.8.1: 02:00 4.9.2: 01:52 dataWrite: 772240 dataRead: 220640 setAddress: 137900	Time 4.8.1: 02:00 4.9.2: 01:52 dataWrite: 772240 dataRead: 220640 setAddress: 137900
23	glcdPutStr3v-04	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 2016000 dataRead: 542400 setAddress: 168000	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 2016000 dataRead: 542400 setAddress: 168000
24	glcdPutStr-01	Time 4.8.1: 02:00 4.9.2: 02:00 dataWrite: 6058080 dataRead: 0 setAddress: 48080	Time 4.8.1: 02:00 4.9.2: 02:00 dataWrite: 6058080 dataRead: 0 setAddress: 48080

Table 26: Performance test run Emuchron v3.1 vs Emuchron v4.0

B.2 Test results Emuchron v4.0 vs v4.1

Find below an overview of version and build sizes of the firmware when built with default settings.

Version avr-gcc	Emuchron v4.0 Object size (bytes)	Emuchron v4.1 Object size (bytes)
4.8.1 (Debian 8)	.data: 984 .text: 25446 **Total: 26430**	.data: 984 .text: 25428 **Total: 26412**
4.9.2 (Debian 9)	.data: 984 .text: 25602 **Total: 26586**	.data: 984 .text: 25568 **Total: 26552**

Table 27: Default firmware size Emuchron v4.0 vs Emuchron v4.1

Some remarks on the build statistics:

- The difference between firmware size of the v4.0 and v4.1 builds is very small and is solely the result of code optimizations in `glcdPutStr3v()`.

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.

The graphics performance in Emuchron v4.1 is impacted by the following implemented features:

- Code optimizations are applied in `glcdPutStr3v()` resulting in improved drawing performance in all `glcdPutStr3v()` tests.

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 more information regarding the glcd dataWrite, dataRead and setAddress indicators refer to section 5.8.13. The indicator values listed below are those from the test run on avr-gcc 4.8.1.

Test	***Test Name***	***Emuchron v4.0***	***Emuchron v4.1***
1	glcdDot-01	Time 4.8.1: 02:01 4.9.2: 01:56 dataWrite: 1048576 dataRead: 2097152 setAddress: 2097152	Time 4.8.1: 02:01 4.9.2: 01:56 dataWrite: 1048576 dataRead: 2097152 setAddress: 2097152
2	glcdDot-02	Time 4.8.1: 02:01 4.9.2: 01:56 dataWrite: 630784 dataRead: 2523136 setAddress: 1892352	Time 4.8.1: 02:01 4.9.2: 01:56 dataWrite: 630784 dataRead: 2523136 setAddress: 1892352
3	glcdLine-01	Time 4.8.1: 02:00 4.9.2: 01:59 dataWrite: 335819 dataRead: 1694889 setAddress: 388101	Time 4.8.1: 02:00 4.9.2: 01:59 dataWrite: 335818 dataRead: 1694888 setAddress: 388101
4	glcdLine-02	Time 4.8.1: 01:59 4.9.2: 01:57 dataWrite: 1286488 dataRead: 1505000 setAddress: 390992	Time 4.8.1: 01:59 4.9.2: 01:57 dataWrite: 1286488 dataRead: 1505000 setAddress: 390992
5	glcdCircle2-01	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1616058 dataRead: 1867044 setAddress: 501972	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1616058 dataRead: 1867044 setAddress: 501972
6	glcdCircle2-02	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1646280 dataRead: 2033640 setAddress: 774720	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1646280 dataRead: 2033640 setAddress: 774720
7	glcdFillCircle2-01	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 1513345 dataRead: 863471 setAddress: 1113986	Time 4.8.1: 02:00 4.9.2: 01:55 dataWrite: 1513345 dataRead: 863471 setAddress: 1113986
8	glcdFillCircle2-02	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1065120 dataRead: 1483560 setAddress: 1160220	Time 4.8.1: 02:00 4.9.2: 01:57 dataWrite: 1065120 dataRead: 1483560 setAddress: 1160220
9	glcdRectangle-01	Time 4.8.1: 02:00 4.9.2: 01:56 dataWrite: 1511448 dataRead: 2042761 setAddress: 1032468	Time 4.8.1: 02:00 4.9.2: 01:56 dataWrite: 1511448 dataRead: 2042761 setAddress: 1032468
10	glcdRectangle-02	Time 4.8.1: 02:00 4.9.2: 01:58 dataWrite: 2482400 dataRead: 2503280 setAddress: 317840	Time 4.8.1: 02:00 4.9.2: 01:58 dataWrite: 2482400 dataRead: 2503280 setAddress: 317840

Test	***Test Name***	***Emuchron v4.0***		***Emuchron v4.1***	
11	glcdFillRectangle2-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 1899240 2170560 506464	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 1899240 2170560 506464
12	glcdFillRectangle2-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3674000 1376080 100200	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3674000 1376080 100200
13	glcdFillRectangle2-03	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 4065264 1032448 40330	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 4065264 1032448 40330
14	glcdFillRectangle2-04	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3976560 1009920 39450	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:59 3976560 1009920 39450
15	glcdPutStr3-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2197944 2232832 34888	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2197944 2232832 34888
16	glcdPutStr3-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 3030300 1539200 36075	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 3030300 1539200 36075
17	glcdPutStr3-03	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2142720 2177280 34560	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2142720 2177280 34560
18	glcdPutStr3-04	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 2887200 1761192 38496	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 2887200 1761192 38496
19	glcdPutStr3-05	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2283120 2319360 36240	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:01 2283120 2319360 36240
20	glcdPutStr3v-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:53 714000 215900 178500	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:51 01:50 714000 215900 178500
21	glcdPutStr3v-02	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:55 2078496 544368 123720	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:49 01:48 2078496 544368 123720

Test	Test Name	Emuchron v4.0		Emuchron v4.1	
22	glcdPutStr3v-03	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:52 772240 220640 137900	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:49 01:49 772240 220640 137900
23	glcdPutStr3v-04	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 01:55 2016000 542400 168000	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	01:50 01:49 2016000 542400 168000
24	glcdPutStr-01	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 6058080 0 48080	Time 4.8.1: 4.9.2: dataWrite: dataRead: setAddress:	02:00 02:00 6058080 0 48080

Table 28: Performance test run Emuchron v4.0 vs Emuchron v4.1

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 Debian 8 and 9.

C.1 Setting up a terminal profile in Debian 8 and 9

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

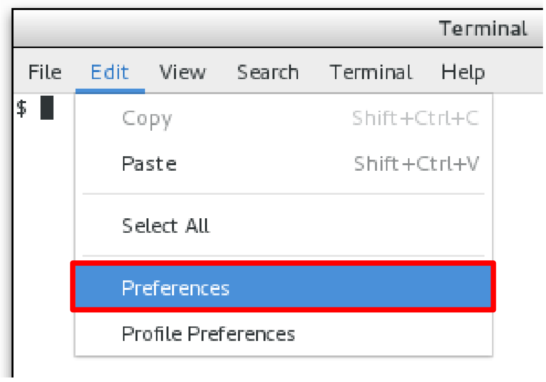


Figure 12: Access terminal profiles

– 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.

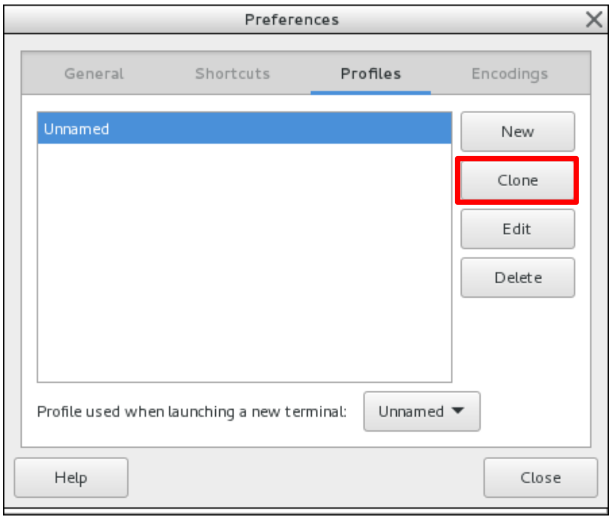


Figure 13: Create new terminal profile

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'.
  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.

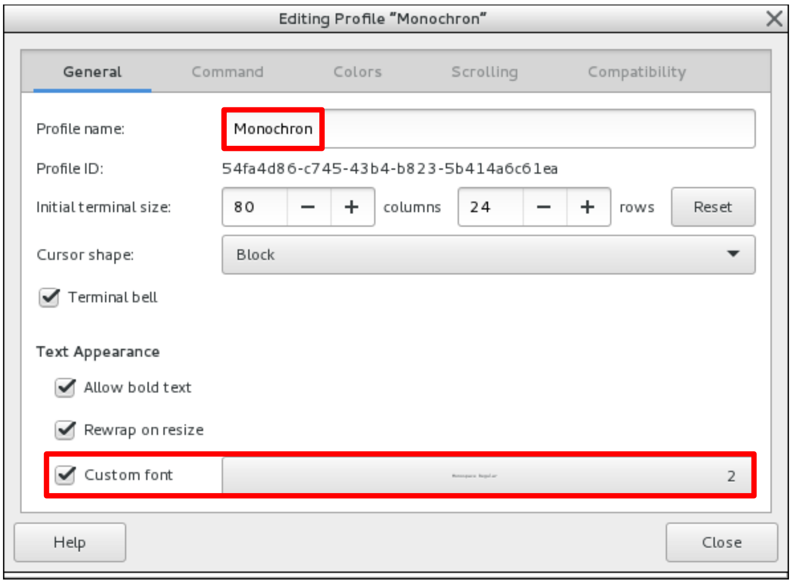


Figure 14: Terminal profile tab 'General'

- Tab 'Command'.
  Note: In Debian 9 the "Update login records" option is no longer present.

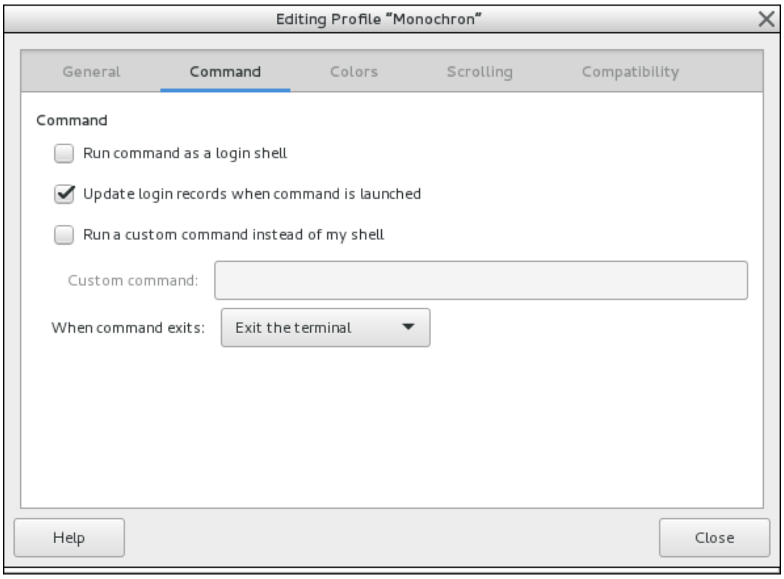


Figure 15: Terminal profile tab ' Command'

– Tab 'Colors'.
  Set the built-in schema to 'White on black'. See below.

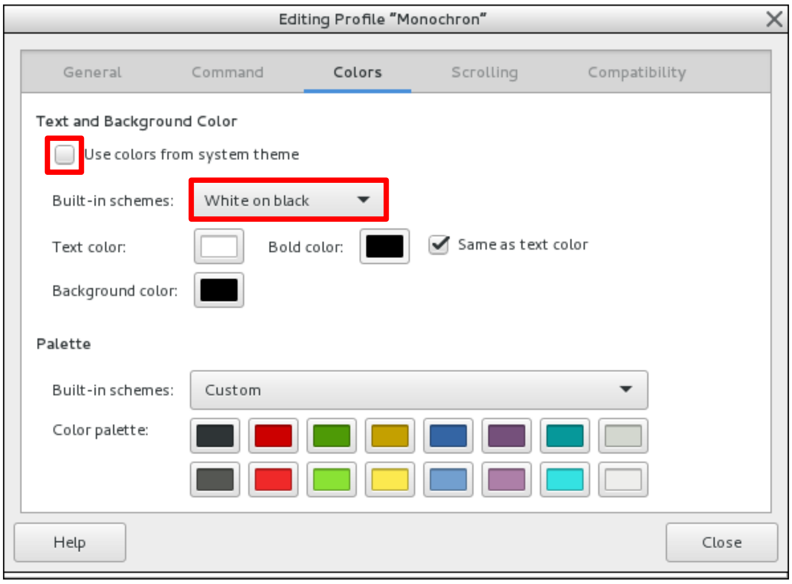


Figure 16: Terminal profile tab 'Colors'

– Tab 'Scrolling'.
  Disable the scrollbar and scrolling. See below.

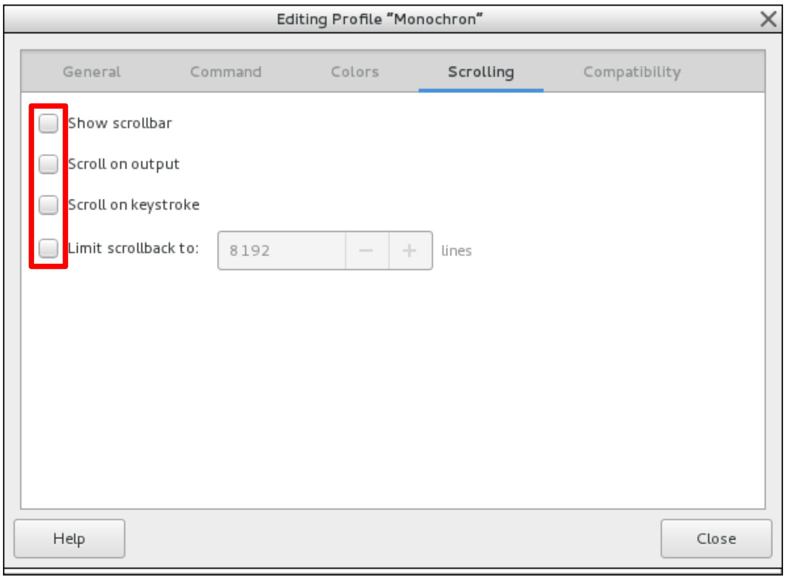


Figure 17: Terminal profile tab 'Scrolling'

– Tab 'Compatibility'.

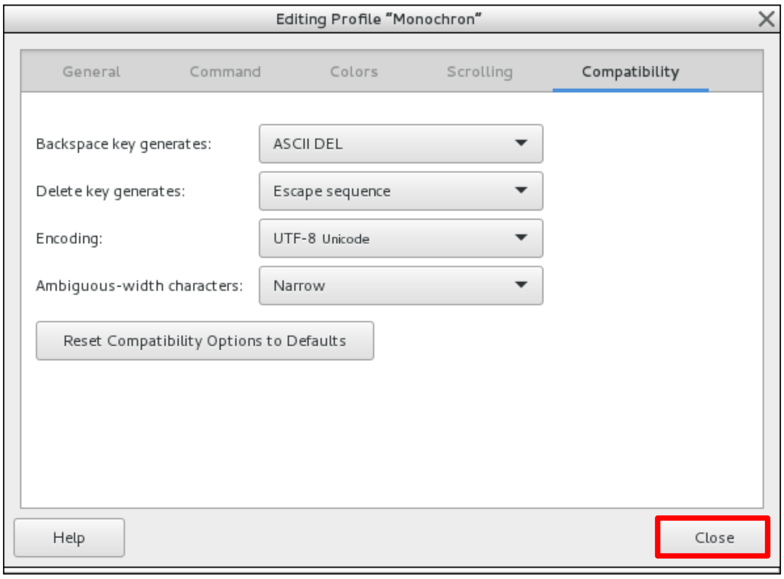


Figure 18: Terminal profile tab 'Compatibility'

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