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

# Erase lcd display and set paint color
le
psf

# 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 0 ver*factor hor-(hor*factor) 0
  pl 0 ver*factor hor*factor ver
  # From right to top and right to bottom
  pl hor ver*factor hor*factor 0
  pl hor ver*factor hor-(hor*factor) ver
rn

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

- EMUCHRON -
A Monochron emulator for Debian Linux

Author: Toine Ceulemans
Version: v8.0
Date: 20 March 2024

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 in the Emuchron QR clocks uses code from project qrduino.
  https://github.com/tz1/qrduino
– Techninja (James T) and Super-Awesome Sylvia
  The Emuchron Marioworld clock is based on the original Mariochron clock code.
  https://github.com/techninja/MarioChron
– Jamie Zawinski and CaitSith2
  The Emuchron Dali clock is based on xdaliclock code that is later integrated in Monochron Multichron code.
  https://www.jwz.org/xdaliclock
  https://github.com/CaitSith2/monochron/tree/MultiChron/firmware

Version history

Version (date) Author	***Description***
v8.0 (2024-03-20) T. Ceulemans	Emuchron emulator code base: – Added repeat commands 'rb' (repeat-break) and 'rc' (repeat-continue). – Added command 'tg' (time get) to load the current date/time in mchron variables. – Command 'lr' (lcd read) allows the optional use of the target variable name. – Single clock cycle statistics are now non-destructive for aggregated statistics and also include a date and timestamp. – Improved keypress user interface when running the emulator for a clock, a Monochron application, or the Monochron configuration pages. – Improved command list execution stability by making (wait)timers no longer interfere with one and another. – Added a data load timestamp in the metadata of a graphics buffer. Monochron firmware code base: – Refactored `mcUpdAlarmSwitch` to `mcAlarmSwitchEvent` for consistency reasons. – The Mario clock includes two Mario sprite sets to choose from. Relevant bug fixes: – None. Generic: – Minor bug fixes and refinements in Emuchron/Monochron code. – Latest documentation.

Summary

Emuchron is a lightweight Monochron emulator for Debian Linux 11 and 12. 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 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

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.
As a product, Monochron has been discontinued as of mid-2019.

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 clock and graphics functionality is to upload firmware to Monochron that generates debug output strings, and have these strings sent back via the Monochron FTDI bus to a terminal application on the connected computer. Although this debug method is still very 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, bitmap and support routines, and is draw speed optimized.

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, Emuchron has a module that implements the controller hardware as a software finite state machine. It processes the data byte, read from stubbed Atmel data ports, 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 front-end gui 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>` **`tap 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 r`**
  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 constantly update itself.
  We will now be able to see on the stubbed LCD device whether the clock correctly processes a day change in its date area.

1.4 Debian Linux and AVR

Emuchron is developed in Debian 12 and has been verified to work in Debian 11. The table below provides the details of the several environments in which Emuchron is verified to work.

Host environment	***VM info***
Host: macOS Monterey 12.7.4 (Intel) Hypervisor: VMware Fusion Player 13.5.1	VM OS: Debian 11 64-bit Kernel: 5.10.0-28-amd64 gcc/avr-gcc: 10.2.1-6/5.4.0 Memory: 2 GB
	VM OS: Debian 12 64-bit Kernel: 6.1.0-18-amd64 gcc/avr-gcc: 12.2.0-14/5.4.0 Memory: 3 GB
Host: Win 10 Pro 64-bit version 22H2 (AMD) Hypervisor: VirtualBox 7.0.14	VM OS: Debian 11 64-bit Kernel: 5.10.0-28-amd64 gcc/avr-gcc: 10.2.1-6/5.4.0 Memory: 2 GB
	VM OS: Debian 12 64-bit Kernel: 6.1.0-18-amd64 gcc/avr-gcc: 12.2.0-14/5.4.0 Memory: 3 GB
Host: macOS Sonoma 14.4 (Apple silicon) Hypervisor: VMware Fusion Player 13.5.1	VM OS: Debian 12 64-bit Kernel: 6.1.0-18-arm64 gcc/avr-gcc: 12.2.0-14/5.4.0 Memory: 3 GB

Table 1: The Emuchron runtime environments for Debian/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.
- The arm64 isa is supported on Debian 12. No installation instructions or code changes were needed for this, including building firmware for and uploading it to Monochron.

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.

1.8 Migrating from Emuchron pre-v4.2 to v4.2 and later

From a functional point of view there are no changes.

However, in v4.2 the Emuchron data stored in eeprom is reorganized for optimization purposes, requiring to use a different Emuchron eeprom data validity identifier. See below the code snippet from monomain.h [firmware].

```
   // Warning: Do not set EE_INITIALIZED to 0xff/255, as that is the state the
   //          eeprom will be in when totally erased.
   #define EE_SIZE              1024
   #define EE_OFFSET            0
-- #define EE_INITIALIZED       0xC3
++ #define EE_INITIALIZED       0x5A
```

The impact of this change is that when v4.2 (or later) firmware is uploaded to a Monochron clock running pre-v4.2 or Adafruit's original Monochron firmware, the Emuchron eeprom data will be reset to default settings as defined in `eepDefault[]` in monomain.c [firmware]. This means that the display brightness, foreground/background color, the four alarm times and selected alarm are reset. Upon reverting back to pre-v4.2 Emuchron firmware or the original Monochron firmware the eeprom data will be reset again according its built-in eeprom reset functionality.

1.9 Migrating from Emuchron v4.x to v5.x

In v5.0 mchron data files are now stored in folder $HOME/.config/mchron instead of $HOME.
The following legacy mchron configuration files are no longer used and can be removed.

```
$ rm ~/.mchron
$ rm ~/.mchron_history
```

The settings for the Monochron ncurses terminal profile has changed in v5.x. Refer to appendix C.1 to update the setting of any existing Monochron terminal profile.
Also, the Monochron ncurses terminal shortcut as supplied in v4.x will no longer work in Debian 10. Refer to appendix D.1 to create a Monochron ncurses terminal main menu launcher instead. Creating a main menu launcher requires Debian package alacarte. For this, see packages.txt [support].

In v5.x, upon building Monochron firmware, a v4.x clock may result in link failures.
This is caused by the avr-gcc 5.4.0 compiler that is not able to properly resolve (static) data references within a module function. The issue was seen upon attempting to compile v4.2 glcd.c [firmware] as well as pong.c [firmware/clock] code in avr-gcc 5.4.0 (see below).

```
:
Linking: monochron.elf
avr-gcc --output monochron.elf 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  -
mmcu=atmega328p -Wl,-Map=monochron.map,--cref -lm
glcd.o: In function `glcdFillRectangle2':
/home/user/Documents/Emuchron/Emuchron/firmware/glcd.c:374:(.text+0x64e):
relocation truncated to fit: R_AVR_7_PCREL against `no symbol'
/home/user/Documents/Emuchron/Emuchron/firmware/glcd.c:374:(.text+0x6ca):
relocation truncated to fit: R_AVR_7_PCREL against `no symbol'
collect2: error: ld returned 1 exit status
make: *** [Makefile:292: monochron.elf] Error 1
$
```

The issue in glcd.c [firmware] is circumvented by moving static data arrays `pattern3Up[]` and `pattern3Down[]` to `progmem` attributes (glcd.c:40-44 [firmware]). The issue in pong.c [firmware/clock] is circumvented by a major revision of the clock code that was planned for v5.0 anyway.
This avr-gcc behavior is considered a linker deficiency.

1.10 Migrating from Emuchron v5.x to v6.x

In v6.x a clock may not compile due to a rename of a text font in glcd.h [firmware]. Apart from that no functional changes are made in Monochron code, and all v5.x clock code is expected to run without any issue.

Description	***API impact***
Old font define: `FONT_5X7N` (N = non-proportional) New font define: `FONT_5X7M` (M = monospace)	Functions `glcdPutStr2()`, `glcdPutStr3()` and `glcdPutStr3v()` for parameter `font`.

Due to this change, and for consistency reasons, mchron commands 'pa' and 'pn' have a modified value for argument font. Scripts that use command 'pa' or 'pn' must be modified to reflect this change (see below).

```
# Emuchron v5.x: command 'pa' (paint ascii) and 'pn' (paint number) using
# non-proportional fontname 5x7n.
pa f 3 10 5x7n h 1 1 Hello!
pd f 3 20 5x7n h 1 1 pi %f

# Emuchron v6.x: fontname 5x7n (non-proportional) is renamed to 5x7m (monospace).
pa f 3 10 5x7m h 1 1 Hello!
pd f 3 20 5x7m h 1 1 pi %f
```

In v6.x three commands are added to interact with the Monochron eeprom. This impacts existing command 'm' (run Monochron application). Scripts that use command 'm' must be modified to reflect this change (see below).

```
# Emuchron v5.x: command 'm' (run Monochron application) has two arguments where
# the second one <eeprom> is used to keep or reset eeprom contents.
m n k

# Emuchron v6.x: as there is now a separate command 'mer' to reset the eeprom,
# command 'm' no longer supports argument <eeprom>.
m n
```

1.11 Migrating from Emuchron v6.x to v7.x

In v7.x a clock will most likely fail to compile due to refactoring efforts, changes in the high-level glcd interface, and simplification of the `#include` section.

First, the project true/false defines have been refactored.

Description	***Code impact***
Old values: `GLCD_TRUE / GLCD_FALSE` (via ks0108.h [firmware]) New values: `MC_TRUE / MC_FALSE` (via global.h [firmware])	Review `#include` section and refactor clock code for modified define values. For changes in the `#include` section, see example for example.c [firmware] below.

The following changes affect the api of several modules.

Description	***Code impact***

The high-level glcd draw functions in glcd.c [firmware] no longer include argument color. Also, dedicated color proxy functions such as `glcdWriteCharFg()` are removed except for `glcdPrintNumberBg()`.
Controlling the draw color is now achieved using the following new functions in glcd.c [firmware]:
`glcdColorGet()`
`glcdColorSet()`
`glcdColorSetBg()`
`glcdColotSetFg()`
	Modify calls to glcd draw functions in clock code, and add calls to the new glcd draw color functions.
	Note: In a clock, in most cases, it is no longer needed to externally refer to Monochron variables `mcFgColor` and `mcBgColor`.
	For examples refer to the many clock modules in [firmware/clock], starting with example.c.

The following ks0108.c [firmware] functions are moved to glcd.c [firmware]:
`glcdClearScreen()`
`glcdResetScreen()`
	Review `#include` section of clock.

The following util.c [firmware] functions are reused or new: `uart_put_dec()` (re-used) `uart_put_sdec()` (new)	Modify calls to `uart_put_dec()` to `uart_put_sdec()` to put a signed `int8_t` or `s08` value. Re-used function `uart_put_dec()` now puts an unsigned `uint8_t` or `u08` value. As a result, existing calls to `uart_putw_dec()` may be changed to `uart_put_dec()`.

The following monomain.c [firmware] functions are renamed and relocated to anim.c [firmware]:
`rtcDotw()` => `calDotw()`
`rtcLeapYear()` => `calLeapYear()`
	Review `#include` section and modify calls.

The structure to include Monochron and Emuchron definitions has been simplified. The main project definitions are now to be loaded from global.h [firmware] instead of monomain.h [firmware] and ks0108.h [firmware]. And, since util.h [firmware] is now also in scope for the emulator code base and is already included in global.h, there's no longer the need to include it in a clock module separately.

Find below a snippet of the modified include section of example.c [firmware/clock].

```
   // We need the following includes to build for Atmel Monochron/Linux Emuchron
-- #ifdef EMULIN
-- #include "../emulator/stub.h"
-- #else
-- #include "../util.h"
-- #endif
-- #include "../ks0108.h"
-- #include "../monomain.h"
++ #include "../global.h"     // Main Monochron/Emuchron defs
   #include "../glcd.h"       // High-level graphics function defs
   #include "../anim.h"       // Animation util function defs
   #include "example.h"       // Our clock init/cycle function defs
```

There are quite a few changes in the Emuchron command set that may require modifying scripts.

– Removed argument `color` from all paint draw commands.
  Commands 'ps', 'psb' and 'psf' are added to manage the glcd draw color.

- Command 'lcs' (set x/y cursor) is replaced by 'lxs' (set controller x cursor) and 'lys' (set controller y cursor).
  Command name 'lcs' is re-used to select the active ks0108 controller.
  As a result, commands 'lr' (controller read) and 'lw' (controller write) no longer include argument `controller`.
- Moved all commands from command groups alarm and date to command group time. Existing alarm and date commands are renamed by prefixing it with a 't'. For example, command 'at' (toggle alarm switch) is changed to 'tat'.

1.12 Migrating from Emuchron v7.x to v8.x

In v8.x a clock may fail to compile due to refactoring efforts.

Description	***Code impact***
The following Monochron variable has been renamed for consistency reasons. Old name: `mcUpdAlarmSwitch` New name: `mcAlarmSwitchEvent`	Refactor clock code for modified variable name.

In v8.x three commands are impacted due to a consistent approach towards emulator execution modes. Emulator start mode 'n' (normal emulator mode) is renamed to 'r' (emulator run mode).
Scripts that use command 'cf', 'm' or 'mc' with start mode argument 'n' must be modified to reflect this change (see below).

```
# Emuchron v7.x: clock/monochron emulator commands 'cf', 'm' and 'mc' have
# argument <mode>. Value 'n' indicates a 'normal' startmode with continuous
# clock events.
cf n
m n
mc n 0 1

# Emuchron v8.x: clock/monochron emulator commands 'cf', 'm' and 'mc' have
# argument <startmode>. Value 'r' indicates a 'run' startmode with continuous
# clock events.
cf r
m r
mc r 0 1
```

2 The Emuchron project

2.1 The project folder structure

The Emuchron project uses the following folder structure.

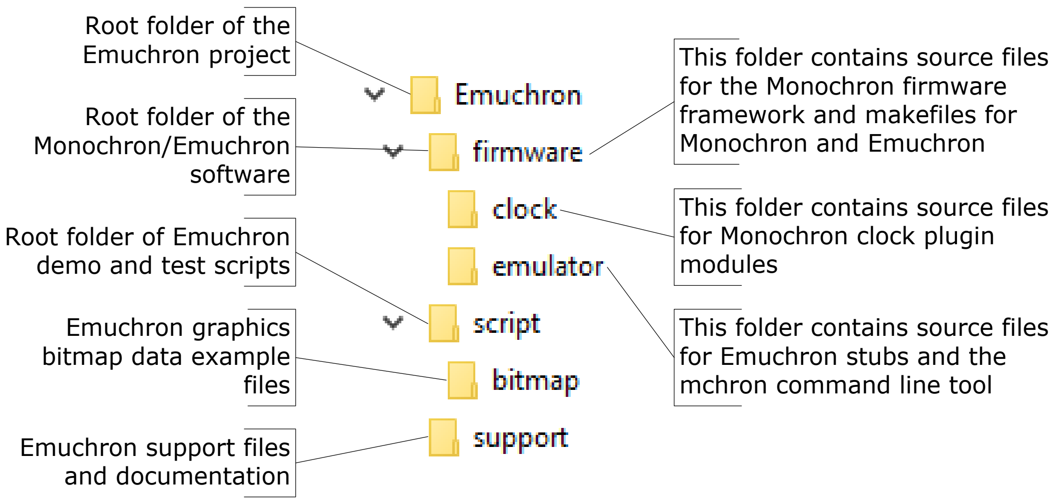


Figure 1: The Emuchron project folder structure

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.

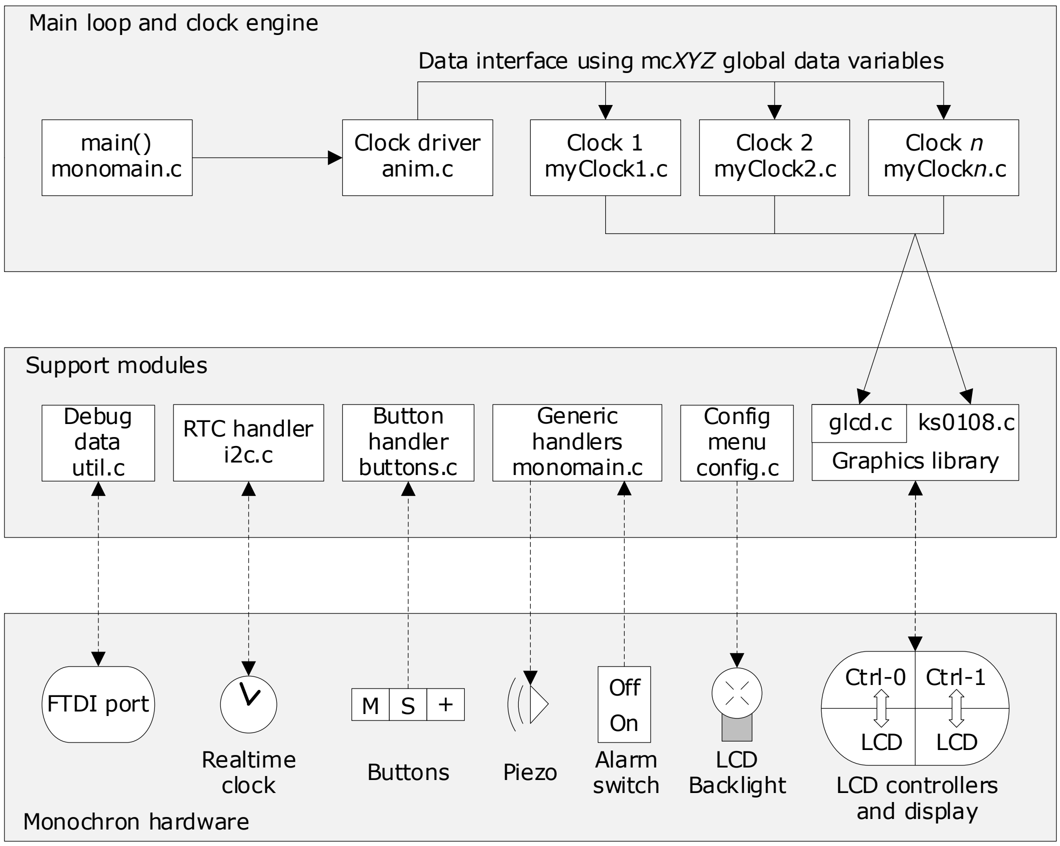


Figure 2: The Monochron runtime environment

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 and most of the mc*XYZ* data interface towards the clock plugins. It also contains several generic clock utility functions.
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, (filled) rectangles and bitmaps. 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, 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.
util.c [firmware]	This support module 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, such as `minicom`, 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.

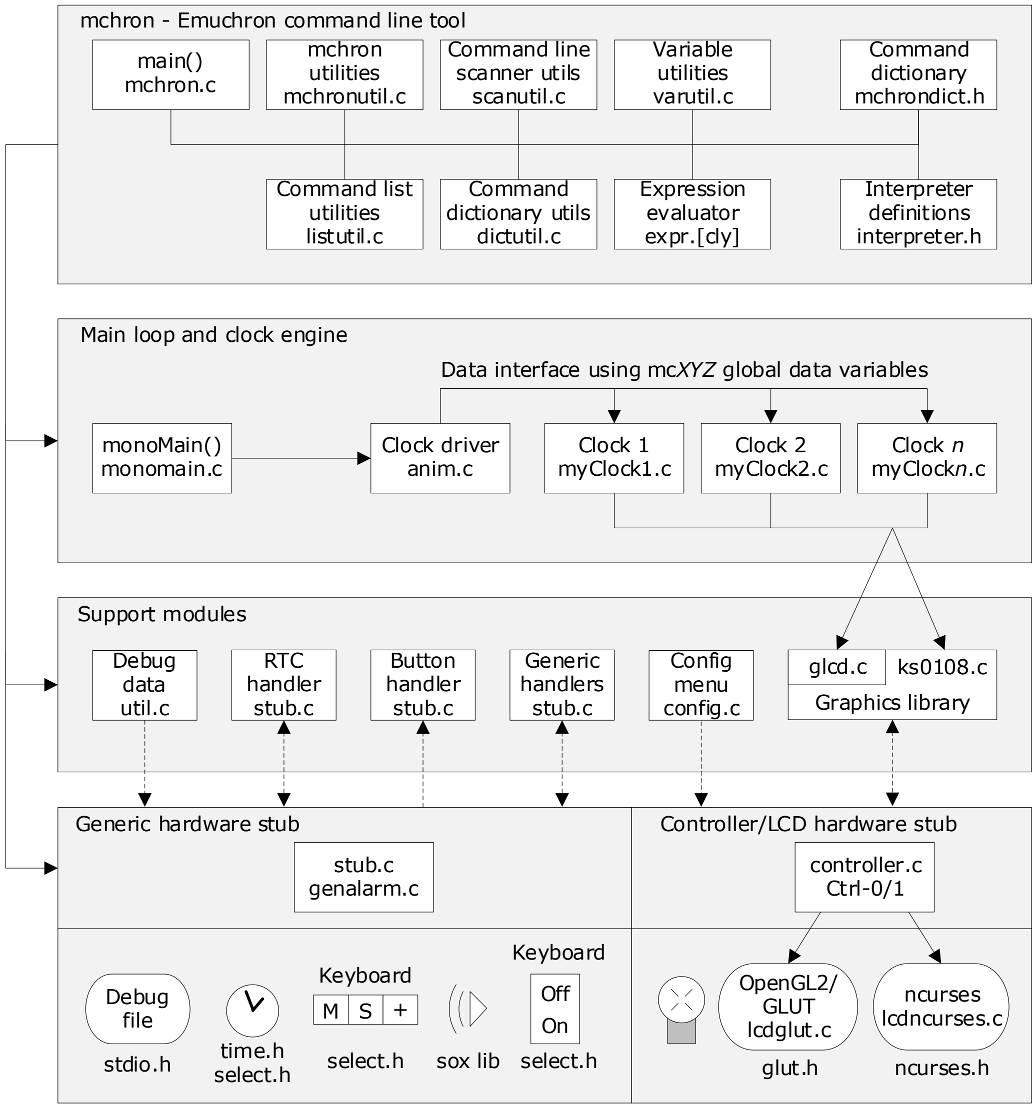


Figure 3: The Emuchron runtime environment

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 stubbed 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 i2c.c [firmware] and buttons.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) – Numbers Exponential number notation is supported. Hexadecimal numbers require a `0x` prefix. – Functions abs(), cos(), frac(), int(), rand()[1], sin() – Constants null, pi, true, false – Variables Named variables representing a numeric value, excluding the constant names and function names listed above. See also varutil.c [firmware/emulator]. [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.

Module	***Description***
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.
lcdglut.c [firmware/emulator]	This module implements an OpenGL2/GLUT LCD device. The GLUT device is run in 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 where up-to-date LCD output is essential. The upside however is that the GLUT interface does not require end-user setup and provides more runtime flexibility. The GLUT window can be resized at will while retaining the 2:1 aspect ratio, supports a right-click event that highlights a glcd pixel and show its glcd coordinates, supports a rudimentary tool to edit the LCD contents using a mouse, and can show pixels bezels and gridlines.
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 where up-to-date 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 6.3.1).
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.

Module	***Description***
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

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	(re)start snoozing and snooze timeout timer invoke `cycle()` for clock
Press '+' button	(re)start snoozing and snooze timeout timer 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

The Monochron variables are defined in anim.c [firmware] and represent a stable software representation of events in and the current state of the Monochron clock.

Note: All variables below are supported in the clock plugin `cycle()` function. The clock plugin `init()` and `button()` functions do not support time, alarming or snoozing state/event, or alarm switch related variables.

Variable	*Description*
`mcAlarmH` `mcAlarmM`	The active alarm time (hour, min), regardless whether the alarm switch position is on or off.
`mcAlarming`	Value: `MC_TRUE` / `MC_FALSE` Indicates whether the clock is alarming or snoozing (`MC_TRUE`) or not (`MC_FALSE`).

Variable	Description
mcAlarmEvent	Value: MC_TRUE / MC_FALSE Signals whether the clock start or stops to alarm. Use it in combination with mcAlarming. It will be triggered when the alarm switch is enabled and the alarm time is met, when the alarm duration times out or when pressing the 'M' button. This event must be handled in the clock cycle() as it is reset every application clock cycle.
mcAlarmSwitch	Value: ALARM_SWITCH_NONE / ALARM_SWITCH_ON / ALARM_SWITCH_OFF The current on/off state of the alarm switch. – ALARM_SWITCH_NONE Internal value only. This value will not be seen in the clock cycle() function. Its purpose is to force an alarm switch change event in mcAlarmSwitchEvent. – ALARM_SWITCH_ON Indicates that the alarm switch position is switched on. – ALARM_SWITCH_OFF Indicates that the alarm switch position is switched off.
mcAlarmSwitchEvent	Value: MC_TRUE / MC_FALSE Signals a change in the alarm switch position. Use it in combination with mcAlarmSwitch. When the alarm switch is switched off, it will also signal the end of active alarming or snoozing. Note that when mcClockInit is MC_TRUE then mcAlarmSwitchEvent is also MC_TRUE. This event must be handled in the clock cycle() as it is reset every application clock cycle.
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 is 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 and the glcd color functions, can freely swap between showing itself white-on-black and black-on-white without any code changes.
mcClockInit	Value: MC_TRUE / MC_FALSE Signals 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 a clock pool 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 pool array being used is emuClockPool in mchron.c [firmware/emulator].

Variable	***Description***
mcClockDateEvent	Value: MC_TRUE / MC_FALSE Signals 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 MC_TRUE only when mcClockTimeEvent is MC_TRUE.
mcClockTimeEvent	Value: MC_TRUE / MC_FALSE Signals that the time or time+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.
mcSnoozing	Value: MC_TRUE / MC_FALSE Indicates whether the clock is snoozing (MC_TRUE) or not (MC_FALSE). Note that mcSnoozing can be MC_TRUE only when mcAlarming is MC_TRUE.
mcSnoozeEvent	Value: MC_TRUE / MC_FALSE Signals whether the clock (re)starts or stops snoozing. Use it in combination with mcSnoozing. It will be triggered when during alarming or snoozing the '+' or 'S' button is pressed, or when the snooze duration times out. This event must be handled in the clock cycle() as it is reset every application clock cycle.
mcTickerSnooze	Value: >= 0 The snooze countdown timer in seconds. It can only be greater than 0 when mcSnoozing is MC_TRUE. When snoozing, its value is decreased when an mcClockTimeEvent is triggered.
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.

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 = MC_FALSE	mcClockOld*XY* = the last created timestamp mcClockNew*XY* = the last created timestamp
mcClockTimeEvent = MC_TRUE	mcClockOld*XY* = the previous timestamp mcClockNew*XY* = the new timestamp

Table 8: The Monochron time event and time variables

In a clock plugin cycle() function quite a number of events can happen related to alarming and snoozing that require checking several Monochron variables. Find below an overview of these events and their representation in Monochron variable data.

Alarm event	***Representation in Monochron variables***
Alarm is triggered. Clock starts audible alarm.	**mcAlarmEvent = MC_TRUE** mcAlarming = MC_TRUE
Pressing the '+' or 'S' button while alarming or snoozing. The clock (re)starts snoozing without audible alarm.	**mcSnoozeEvent = MC_TRUE** mcSnoozing = MC_TRUE mcTickerSnooze = remaining snooze time
Decrease snooze countdown timer while snoozing. Continue snoozing without audible alarm.	**mcClockTimeEvent = MC_TRUE** mcSnoozing = MC_TRUE mcTickerSnooze = remaining snooze time
Snooze countdown timer times out. Restart audible alarm.	**mcSnoozeEvent = MC_TRUE** mcSnoozing = MC_FALSE mcTickerSnooze = 0
Alarm countdown timer times out. End of audible alarm. Ready to trigger alarm again.	**mcAlarmEvent = MC_TRUE** mcAlarming = MC_FALSE
Pressing the 'M' button while alarming or snoozing. End of (audible) alarm. Ready to trigger alarm again.	**mcAlarmEvent = MC_TRUE** mcAlarming = MC_FALSE
Flipping the alarm switch to off regardless of alarming/snoozing or not. End of (audible) alarm. Alarm cannot be triggered.	**mcAlarmSwitchEvent = MC_TRUE** mcAlarmSwitch = ALARM_SWITCH_OFF mcAlarming = MC_FALSE
Flipping the alarm switch to on. Ready to trigger alarm.	**mcAlarmSwitchEvent = MC_TRUE** mcAlarmSwitch = ALARM_SWITCH_ON

Table 9: Alarm events and their representation in Monochron 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, and are draw speed optimized. 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 remove parameter `color` or `inverted`. Functions are added to control the glcd draw color.
`glcdBitmap()`	Draw graphics from progmem or RAM bitmap data.

`glcdBitmap08Pm()`
`glcdBitmap08Ra()`
`glcdBitmap16Pm()`
`glcdBitmap16Ra()`
`glcdBitmap32Pm()`
`glcdBitmap32Ra()`
	A total of six proxy functions for `glcdBitmap()` with a reduced interface for simplicity and allowing to optimize code on object size.

`glcdCircle()`	Superseded by `glcdCircle2()`.

Function	Description
`glcdCircle2()`	Similar to `glcdCircle()` but in addition supports drawing a dotted (1:2 and 1:3) circle outline.
`glcdClearDot()` `glcdSetDot()`	Superseded by `glcdDot()`.

`glcdColorGet()`
`glcdColorSet()`
`glcdColorSetBg()`
`glcdColorSetFg()`
	A set of four functions to get or set the glcd draw color.

`glcdDot()`	Draw a dot.
`glcdFillCircle2()`	Draw a filled circle with several fill patterns. Note that for non-full filled patterns this function does not draw the circle outline. Use an additional call to `glcdCircle2()` to draw the circle outline.
`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()`	Proxy function for legacy function `glcdPrintNumber()` allowing to optimize code on object size.
`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 supporting a monospace 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 10: 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_5X7M`	A 5x7 monospace 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 11: 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:

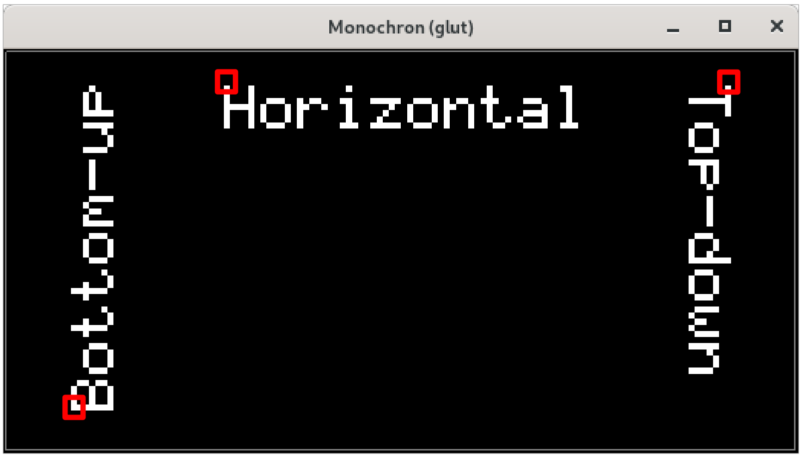
Text orientation	Description
`ORI_HORIZONTAL`	Paint the text horizontally.
`ORI_VERTICAL_BU`	Paint the text vertically in a bottom-up direction.
`ORI_VERTICAL_TD`	Paint the text vertically in a top-down direction.

Table 12: Text orientation overview

Enter the following mchron commands.

```
mchron> pa 35 5 5x7m h 1 1 Horizontal
mchron> pa 10 57 5x7m b 1 1 Bottom-up
mchron> pa 117 5 5x7m 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

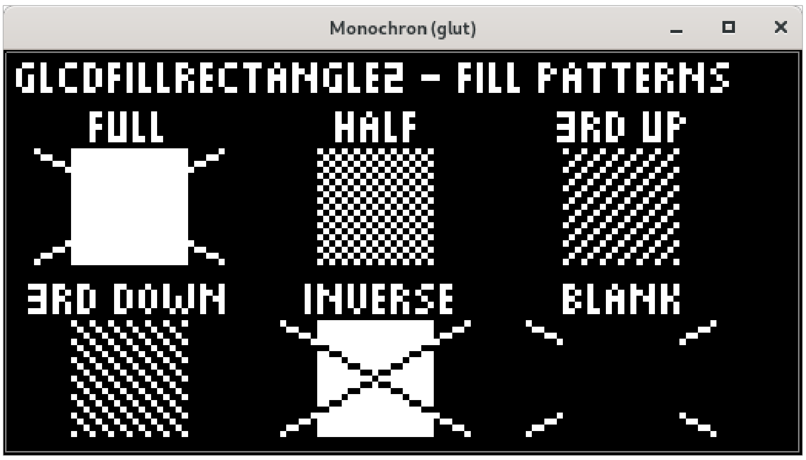
The `glcdFillRectangle2()` and `glcdFillCircle2()` functions provide a method to fill an area with several fill patterns.

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 13: Fill pattern overview

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



2.6.6 Fill alignment

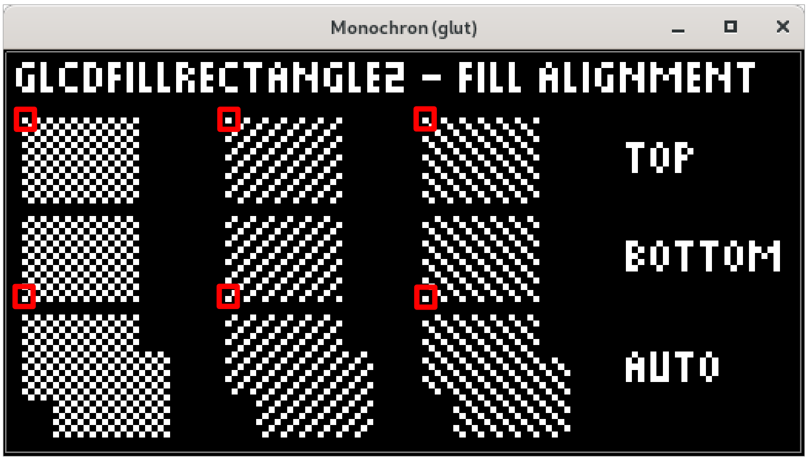
The `glcdFillRectangle2()` function supports a fill alignment option for fill patterns `FILL_HALF`, `FILL_THIRDUP` and `FILL_THIRDDOWN`.

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 and another.

Table 14: 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

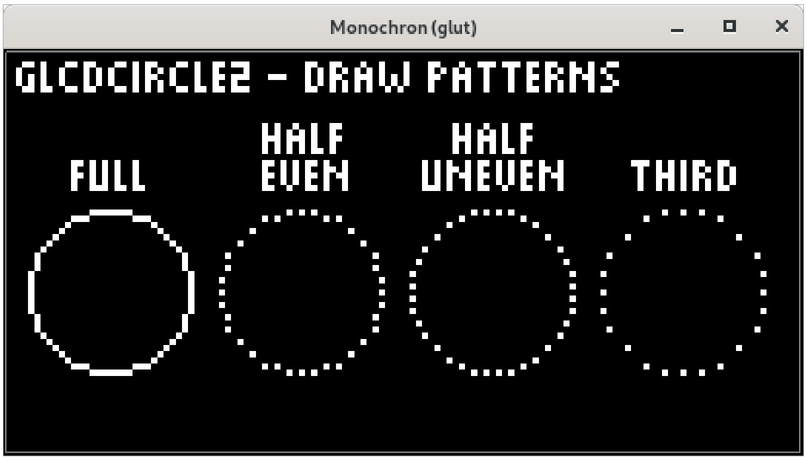
The `glcdCircle2()` function provides a method to draw a circle using several patterns.
Note that the method to determine which pixels are drawn is rather crude. The quality of the non-full draw patterns will vary depending on the radius and center of the circle being drawn.

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 15: Circle draw pattern overview

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



2.6.8 Bitmap data graphics

The `glcdBitmap()` function allows to draw graphics bitmap data onto the LCD. The bitmap data can, for example, contain a single image consisting of multiple image frames, or sprites where each individual image is a single sprite frame.

The bitmap element size defines the maximum vertical size of an image frame or sprite frame.
In glcd.h [firmware] the following bitmap data element sizes are defined:

Data element size	**Description**
`ELM_BYTE`	Element is single byte data; 1 byte (8 bits).
`ELM_WORD`	Element is word byte data; 2 bytes (16 bits).
`ELM_DWORD`	Element is double word byte data; 4 bytes (32 bits).

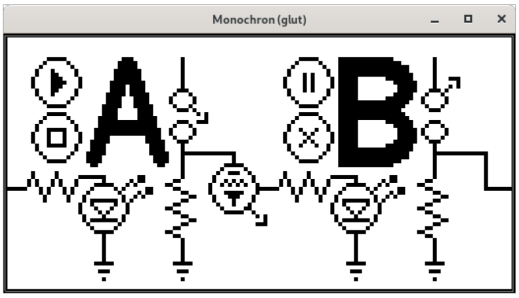
Table 16: Bitmap data element size

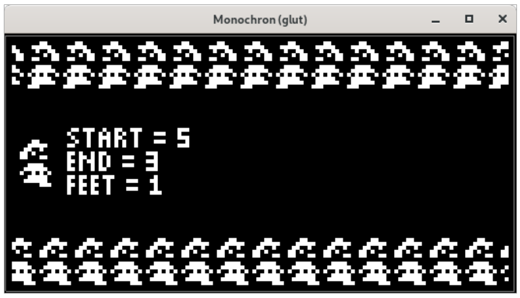
The Atmel avr environment allows to store static data in program memory (`progmem`) instead of RAM with the purpose to save precious RAM. The `glcdBitmap()` function supports both data origins for reading bitmap data.
In glcd.h [firmware] the following bitmap data origins are defined:

Data origin	**Description**
`DATA_PMEM`	The bitmap data is defined as static `progmem` data.
`DATA_RAM`	The bitmap data is dynamic and resides in RAM.

Table 17: Bitmap data origin

Refer to the screenshots below for examples of a full screen bitmap image and a series of Mario bitmap sprite frames, generated using respectively test script graphics1.txt [support] and graphics2.txt [support].



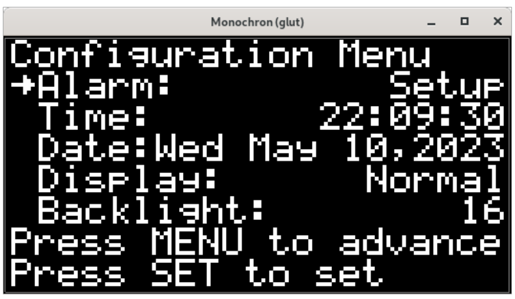


2.7 Monochron configuration screens

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 screen dump), upon pressing the 'Set' button at the 'Alarm' item, the alarm setup menu (right screen dump) 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
```

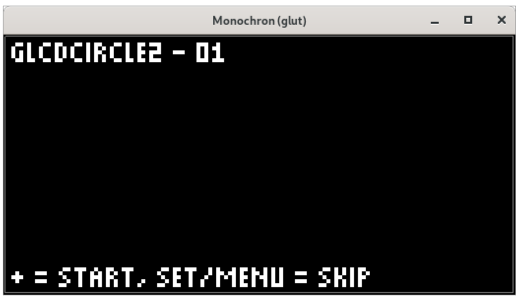
Alarming and snoozing timeouts are controlled by the following defines in alarm.h [firmware]. Note that for the emulator reduced timeouts are specified.

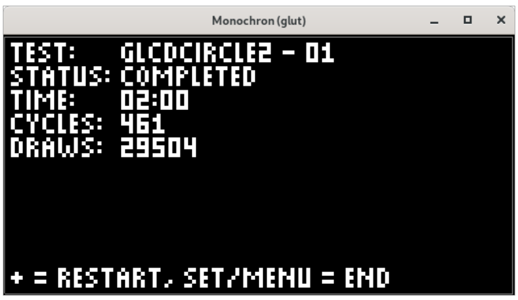
```
// Set timeouts for snooze and alarm (in seconds)
#ifndef EMULIN
#define ALM_TICK_SNOOZE_SEC   600
#define ALM_TICK_ALARM_SEC    1800
#else
// In our emulator we don't want to wait that long
#define ALM_TICK_SNOOZE_SEC   25
#define ALM_TICK_ALARM_SEC    65
#endif
```

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.

Refer to appendix B for performance test results. 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 5.8.4 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 seven 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.
graphics*X*.txt [script]	A total of seven scripts for testing high-level glcd graphics. It shows how to load and display graphics bitmap data, as well as verifying the correctness of the bitmap function. Script graphics6.txt is used to load Mario sprite data, modify and test the sprite data, and save the modified sprite data in a file.
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 18: Relevant command scripts overview

2.11 The pre-built monochron.hex firmware

This project contains pre-built monochron.hex [firmware] firmware using avr-gcc 5.4.0 (Debian 12).
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 the `emuClockPool` arrays that contain the public functions of configured clock plugins. Below is detailed info on the structure members.

Refer to anim.c [firmware] and `emuClockDict[]` in mchronutil.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 19: The `clockDriver_t` clock driver structure elements

Note that upon entering the `init()`, `cycle()` or `button()` function, the glcd draw color is set to the foreground color.

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, from top to bottom.
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`
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.
anim.c [firmware]	– Include the new clock header. `#include "clock/example.h"` – When you want to test or upload your new clock to an actual Monochron clock, add the clock id, public `init()`, `cycle()` and (optional) `button()` functions for the clock in static array `monochron[]`. See also changes for Makefile [firmware]. The example clock does not have a `button()` function.
mchron.c [firmware/emulator]	– Verify if the clock requires special handling in `doTimeAlarmSet()`. For the example clock this is not the case.
mchronutil.c [firmware/emulator]	– Include the new clock header. `#include "../clock/example.h"` – Add the clock id and public `init()`, `cycle()` and (optional) `button()` functions, and a short description for the clock in static array `emuClockDict[]`. The example clock does not have a `button()` function. – Verify if the clock requires special handling in `emuClockUpdate()`. For the example clock this is not the case.
help.txt [support]	– Modify the help text for command 'cs' by adding the example clock index from `emuClockDict[]` in mchronutil.c [firmware/emulator] with a short description.
mchrondict.h [firmware/emulator]	– Increment the upper range of domain `domNumClock` to the new total number of clocks. This allows us to select any clock in command 'cs'.
MakefileEmu [firmware]	– Add the new example.c file in variable `CSRC`. This is required for building Emuchron and the mchron command line tool that includes the example clock.
Makefile [firmware]	– When appropriate add the new example.c file in variable `SRC`. This is required for building Monochron firmware that includes the example clock. See also changes for anim.c [firmware].

Table 20: What to create/modify when adding a new clock plugin

3 Setting up the software environment

3.1 Introduction

Emuchron is supported on Debian 11 and 12 for isa amd64, and Debian 12 for isa arm64. 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. 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 issues observed.

3.2.2 Configuring a Debian VM in VirtualBox

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

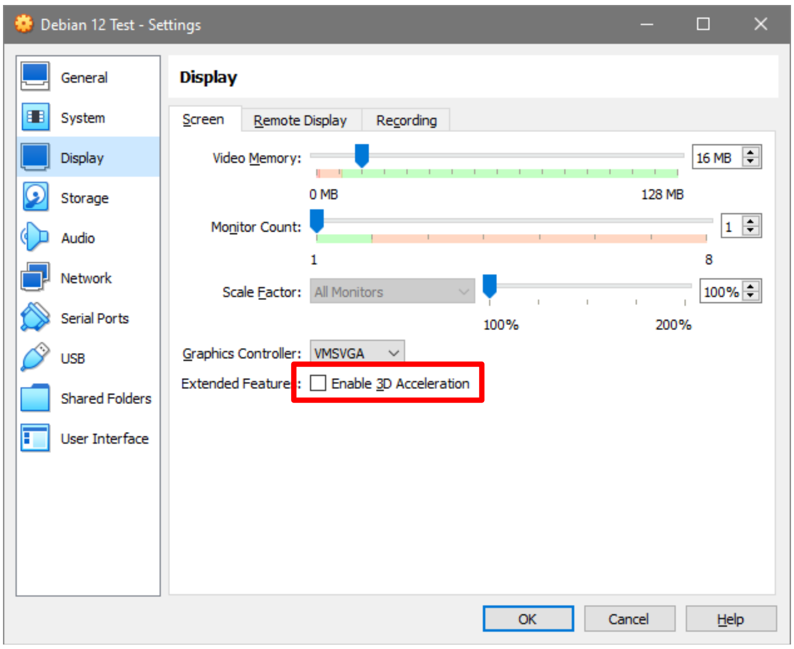


Figure 4: Disable 3D acceleration for a Debian VM in VirtualBox

In case of audio issues, such as a seemingly hanging application when using audio, it is recommended to disable audio. Only enable and keep enabled when it has proven to work correctly.

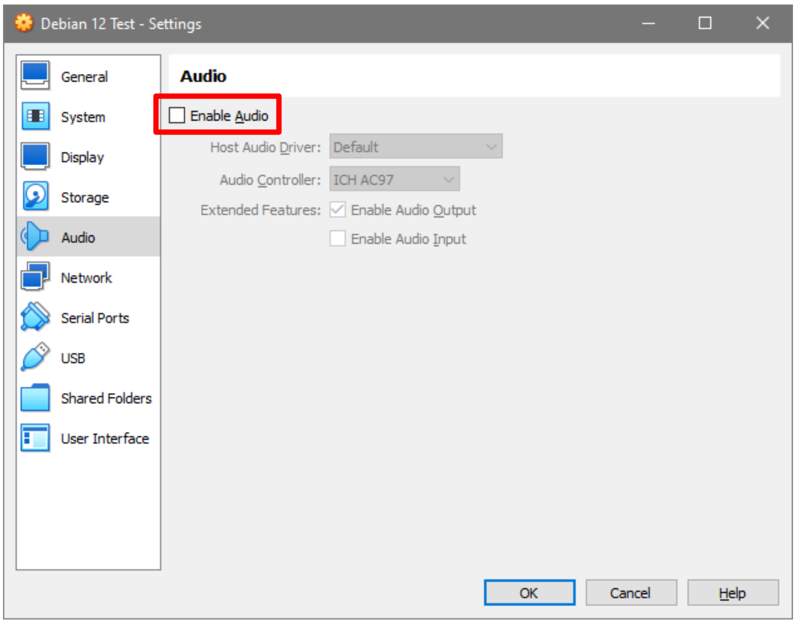


Figure 5: Disable audio for a Debian VM in VirtualBox

3.2.3 Configuring a Debian VM in VMware Fusion

Debian relies on open-vm-tools packages to integrate the VM in VMware Fusion. Upon actual use graphics issues may arise, such as apparent time skipping in clocks due to missed screen updates.
It is therefore recommended to initially disable 3D acceleration. Only enable and keep enabled when it has proven to work correctly.

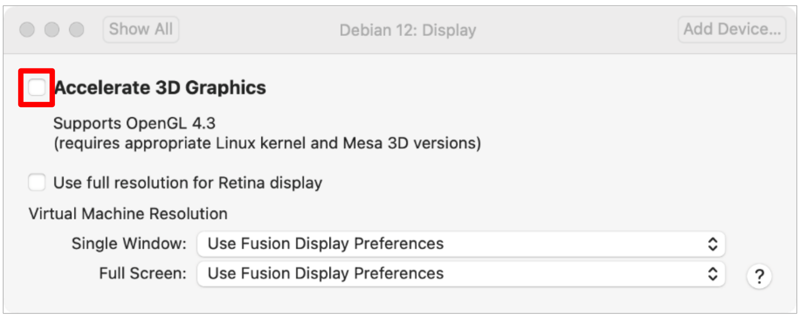


Figure 6: Disable 3D acceleration for a Debian VM in VMware

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 user 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, and debugging tools.

In the Emuchron node the shell script packages.txt [support] is available to download and install all required packages.

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: Depending on the configuration of `apt-get` it is possible that the tool asks the end-user to insert the original Debian 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 exclude physical media to be scanned via application Software & Updates, tab Other Software. Or, manually comment-out the reference(s) to physical installation media in sources.list [/etc/apt]. This will require admin rights. When needed, rerun the packages script.

3.5 Installing a gdb front-end gui

The packages.txt [support] script will not install a gdb front-end gui. The reason for this is that there are quite a few gdb front-end gui's available.
Emuchron is developed using Gede as its gdb front-end gui.

Tool	***Description***
Gede	Gede is a simple yet effective gdb front-end gui. Unfortunately, no `apt` Debian package is available, but the Gede website provides instructions on how to prepare for and build Gede. Version 2.19.3 is stable. Building Gede on Debian should be warning free. Download Gede at: http://gede.dexar.se/ Gede installation instructions for Debian (using sudo privileges) at: http://gede.dexar.se/pmwiki.php?n=Site.Building When installed, for convenience, create a main menu launcher for Gede per instructions in appendix D.2.

Table 21: The Gede gdb front-end gui

3.6 Creating an mchron configuration folder

The `mchron` tool uses two runtime information files. These files are stored in a dedicated mchron configuration folder.
Create this folder using the following command.

```
# Do NOT switch to an admin user as the folder to be created is local to the
# current user
$ mkdir ~/.config/mchron
```

If this folder is not created, the `mchron` tool will complain about not being able to create/access files in the folder.

3.7 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.

3.8 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 device type. 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 terminal size, 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, a GNOME terminal is created that can be used as an ncurses Monochron LCD stub device.

3.8.1 Creating a Monochron terminal profile

The instructions for creating a Monochron terminal profile are found in appendix C.

3.8.2 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, use the Monochron ncurses terminal application launcher, created as per instructions in appendix D.1.
Or, execute the command copied from commands.txt [support] item #3.

When started, a blank terminal is shown as below. Note the small font size of the prompt.

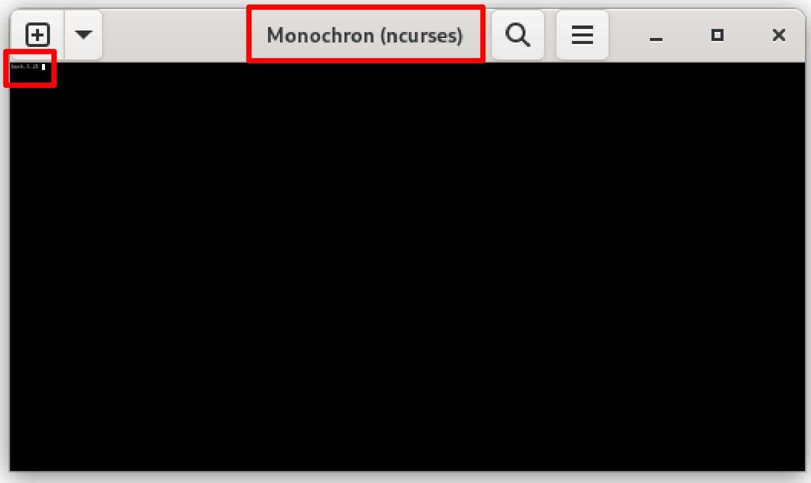


Figure 7: A blank Monochron ncurses terminal

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

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

3.8.3 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'.

3.9 Debian issues and regression in functionality

3.9.1 ALSA audio is less responsive in a VM

Note: While not ignoring the solution described here, for Debian 12 try the solution proposed in 3.9.2 below first, regardless of machine hardware or hypervisor used.

This issue only applies to Debian installations that run as a VM. To illustrate the problem, in mchron, execute script beep.txt [script].
There is a time lag between individual beeps that has gotten worse over time. In addition to that, in some cases 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 is less responsive.

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 coded in stub.c [firmware/emulator].
Note that the workaround is currently disabled as specific sox audio options appear to prevent audio clipping behavior. It 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
```

3.9.2 Stuttering ALSA audio in VMware on Apple Silicon

It appears that the emulated audio driver in VMware on Apple Silicon is for now less optimized than its Intel counterpart. You can hear this as initial stuttering when starting to play audio. The VMware Fusion community however has found an effective Debian 12 WirePlumber configuration workaround that is described here. The one-time only workaround configuration does not require any root access or VM reboot.

3.9.3 Unstable timekeeping in VirtualBox 7.x

It appears that timekeeping in a VirtualBox 7.x guest has become less stable. For example, when running multiple guests, the system time in guests may show a gap of more than a minute. Also, there are 'dips' in guest system timekeeping where time appears to be halting for a brief moment after which it jumps forward in time, thus impacting system wait timers. In Emuchron this will reveal itself by sometimes showing choppy time event-based behavior. There is also a substantially higher number of 75 msec clock cycle timeouts, as shown as metric in the Emuchron stub statistics. For now, there is no workaround or solution known to this problem.

3.10 Uninstalling Emuchron

Uninstall Emuchron by removing the mchron configuration folder and its contents, as well as all relevant Emuchron/Monochron Debian packages.

```
$ # First remove the mchron configuration folder and its contents
$ rm -rf ~/.config/mchron
$
$ # Remove all relevant Debian packages, followed by a cleanup of any
$ # remnant packages. An incomplete package list example is given below.
$ # See packages.txt [support] for the full list of packages.
$ # Only an admin user is allowed to uninstall and cleanup stuff.
$ su - root
$ apt-get remove avr-libc gcc-avr avrdude libncurses5-dev freeglut3-dev
$ # Remove any remnant packages from the removed packages above
$ apt autoremove
```

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 after:
monochron.elf  :
section                    size      addr
.data                       944   8388864
.text                     24558         0
.bss                        317   8389808
.stab                     57888         0
.stabstr                  16179         0
:
Total                    103479
```

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). Note that upon attempting to upload firmware that exceeds the 30 KB limit, the verification step of that upload will fail. For this, refer to section 4.3.

Also, a build will fail in the linker stage when the '.text' part cannot fit within the 32 KB flash memory size limit. See below.

```
:
Linking: monochron.elf
avr-gcc --output monochron.elf anim.o buttons.o config.o glcd.o i2c.o ks0108.o
monomain.o util.o    (etc)
/usr/lib/gcc/avr/5.4.0/../../../avr/bin/ld: monochron.elf section `.text' will
not fit in region `text'
/usr/lib/gcc/avr/5.4.0/../../../avr/bin/ld: address 0x800dae of monochron.elf
section `.bss' is not within region `data'
/usr/lib/gcc/avr/5.4.0/../../../avr/bin/ld: address 0x800dae of monochron.elf
section `.bss' is not within region `data'
/usr/lib/gcc/avr/5.4.0/../../../avr/bin/ld: region `text' overflowed by 1573
bytes
collect2: error: ld returned 1 exit status
make: *** [Makefile:297: monochron.elf] Error 1
$
```

If 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.

Also 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 amd64/arm64 object code is incompatible with Monochron AVR Atmel object code, resulting in link failures. See below.

```
:
Linking: monochron.elf
avr-gcc --output monochron.elf 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  -
mmcu=atmega328p -Wl,-Map=monochron.map,--cref -lm
anim.o: file not recognized: File format not recognized
collect2: error: ld returned 1 exit status
make: *** [Makefile:292: 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 amd64/arm64 object code, resulting in link failures. See below.

```
:
gcc -o mchron emulator/stub.o emulator/controller.o emulator/lcdglut.o
emulator/lcdncurses.o emulator/dictutil.o emulator/listutil.o
:
clock/pong.o clock/puzzle.o clock/qr.o clock/qrencode.o clock/slider.o
clock/spotfire.o clock/cascade.o clock/speeddial.o clock/spiderplot.o
clock/thermometer.o clock/trafficlight.o -lm -lncurses -lreadline -lglut -lGLU -
lGL -lrt -lpthread emulator/expr.o -lfl
:
/usr/bin/ld: monomain.o: relocations in generic ELF (EM: 83)
/usr/bin/ld: monomain.o: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
make: *** [MakefileEmu:104: 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 terminal 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 non-essential USB devices first.
  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 it is plugged in.
- 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 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. When this occurs a clock application may hang 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

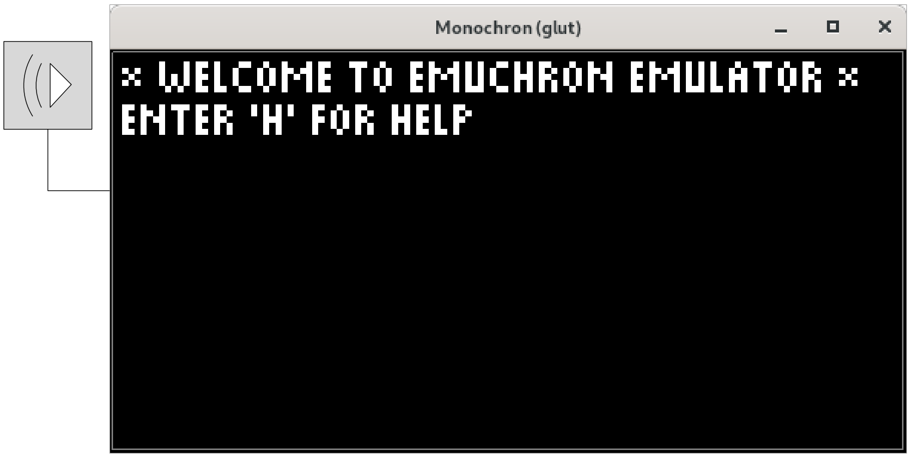
For building `mchron` refer to section 4.2. Find below an excerpt from the help file as found in help.txt [support].

```
mchron - Emuchron emulator command line tool

Use: mchron [-d <logfile>] [-g <geometry>] [-h] [-l <device>] [-p <position>]
            [-t <tty>]

  -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 ~/.config/mchron/tty

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 3.8.1 and 3.8.2 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 3.8.1 and 3.8.2 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 3.8.1 and 3.8.2 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 v8.0 ***

process id  : 3561
ncurses tty : /dev/pts/1

time   : 19:05:31 (hh:mm:ss)
date   : 23/01/2024 (dd/mm/yyyy)
alarm  : 22:09 (hh:mm)
alarm  : off

enter 'h' for help
mchron>
```

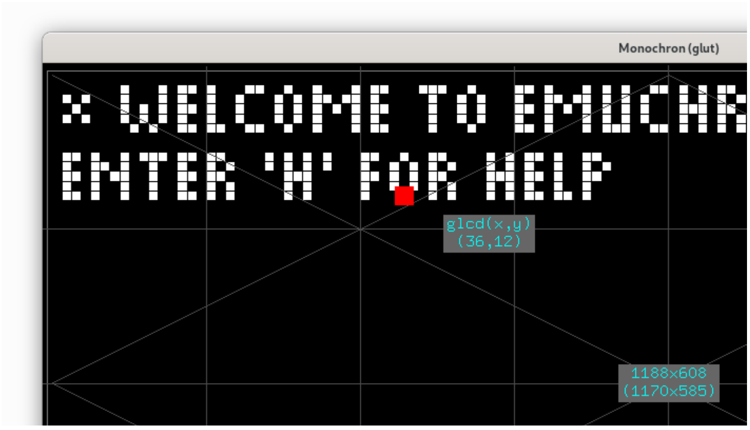
Note: In the event that mchron crashes or properly fails to initialize at startup refer to section 7.6.

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.

The OpenGL2/GLUT LCD device supports several graphics options and functions the ncurses LCD device does not provide. For Emuchron commands mentioned below refer to section 5.8.8.

- Upon resizing the window, using mouse or command 'lgs', a temporary info pop-up appears at the center of the window displaying info on the full window pixel size and the pixel size of the glcd Monochron part of the window.
- Using command 'lgg' one can enable gridlines and, when the window is large enough, activate pixel bezels to separate individual glcd pixels.
- Using command 'lge' one can edit the LCD contents by toggling individual pixels using a mouse left double-click.
- When using a right-click in the glcd area of the window, or command 'lhs', the underlying glcd pixel is highlighted in red while displaying its glcd coordinates in a pop-up.

Refer to an OpenGL2/GLUT screen dump outtake below where most options and functions described above are present.



5.3 Interrupting and stopping mchron

Within mchron there are several ways to interrupt command execution. 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 keep the stack and administers an execution resume point in case execution is to be resumed using command 'er'. The resume stack will be fully cleared in case a new stack is created. 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>
<ctrl>d - exit                        Press '<ctrl>d'
$
```

– 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 x y
3>>
<ctrl>d - abort                       Press '<ctrl>d'
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

Command line tool `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 screen dump 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 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 1773568 Feb 28 14:39 core
$
```

5.5 The mchron command list statistics and stack trace

When executing commands from a command file or multi-command input, mchron optionally provides command list execution statistics upon completing the command list, as well as a stack trace for informational purposes whenever it is interrupted or encounters an error. Controlling whether the execution statistics are provided is done using command 'sls'. For this refer to section 5.8.12.

The command list execution statistics are provided right before the next command prompt. A stack trace line consists of 4 items separated by a colon. For an example of both, 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 ---
1:../script/paint.txt:18:w 0
0:../script/demo.txt:12:e i ../script/paint.txt
-:mchron:-:e s ../script/demo.txt
time=6.200 sec, cmd=27, line=30, avgLine=5
mchron>
```

Press 'q'

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

The list execution statistics are defined as below.

KPI	***Description***
avgLine	The average number of command lines executed per second. This statistic is only provided when the runtime of the command list exceeds 100 msec.
cmd	The total number of non-whitespace (non-empty) command lines executed.
line	The total number of command lines executed. This includes whitespace (empty) command lines.
time	Total runtime in seconds to complete the command lines.

Table 22: Emuchron command list execution statistics

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 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/.config/mchron/history that is created by mchron when not present. Clear the history by stopping mchron and then removing the file. Its configuration is found in scanutil.c [firmware/emulator]. See below.

```
// The readline unsaved cache and history file with size parameters
#define READLINE_CACHESIZE    15
#define READLINE_HISFILE      "/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.
– The `mchron` interpreter 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:

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

Some regular expression examples for searching mchron variable names:

Regex	***Description***
ab	All variables containing the sequence 'ab'.
^ab	All variables starting with 'ab'.
^ab$	Variable 'ab'.
[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 screen dump 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 purpose other 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, 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 '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.
- It is possible that short audio pulses from the Linux ALSA audio system are clipped. Emuchron provides a configurable workaround for this, as described in section 3.9.1.
- 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>
```

5.8.3 '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: <startmode>
              startmode: 'c','r' (c = single cycle, r = run)
  'cp'  - Print overview of available clocks for command 'cs'
  '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 = marioworld,
                     25 = wave, 27 = dali, 27 = perftest
```

Usage specifics:

- For the clock commands, mchron uses the clocks defined in the `emuClockPool` 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 27, 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> # Print overview of available clocks. Note the '*' after clock 2,
mchron> # indicating it is currently the active one.
mchron> cp
clocks:
clock clockId              description
 0    CHRON_NONE           [detach from active clock]
 1    CHRON_EXAMPLE        example
 2*   CHRON_ANALOG_HMS     analog format hms
 3    CHRON_ANALOG_HM      analog format hm
 :
25    CHRON_WAVE           wave banner
26    CHRON_DALI           dali
27    CHRON_PERFTEST       performance test
mchron>
```

```
mchron> # Start the active clock in a testbed environment
mchron> cs 3
mchron> cf r
<run: info = e/p/r/t, hardware = a/s/+, c = cycle, h = help, q = quit>
emuchron clock emulator keypress overview                              Press 'h'
info:
  e = print monochron eeprom settings
  p = print performance statistics
  r = reset performance statistics
  t = print time/date/alarm
hardware:
  a = toggle alarm switch
  s = set button
  + = + button
other:
  c = execute single application clock cycle
  h = provide emulator help
  q = quit

<run: info = e/p/r/t, hardware = a/s/+, c = cycle, h = help, q = quit>
```

Clock emulator specifics:
- Keypress 'a' is identical to command 'tat'
- Keypress 'e' is identical to command 'mep'.
- Keypress 'p' is identical to command 'sp'.
- Keypress 'r' is identical to command 'sr'.
- Keypress 't' is identical to command 'tp'.

Using keypress 'c', the emulator moves to a single application clock cycle mode. This will also change the user prompt.

```
mchron> # Start the active clock in a testbed environment in single cycle mode,
mchron> # or, when in run mode press 'c' to enter single cycle mode
mchron> cf c
<cycle: c = next cycle, p = next cycle + stats, q = quit, other key = run>
cycle statistics:
time    : 13:01:45 (hh:mm:ss)                          Press 'p'
date    : 21/02/2024 (dd/mm/yyyy)
glcd    : dataWrite=96, dataRead=100, addressSet=8
        : ctrlSet=8
ctrl-0  : write=46 (0%), read=48 (96%), display=0 (-%)
        : x=4 (100%), y=2 (100%), startline=0 (-%)
ctrl-1  : write=50 (36%), read=52 (96%), display=0 (-%)
        : x=4 (100%), y=2 (100%), startline=0 (-%)

<cycle: c = next cycle, p = next cycle + stats, q = quit, other key = run>
```

Single application clock cycle keypress specifics:
- Keypress 'c' executes the next application clock cycle.
- Keypress 'p' executes the next application clock cycle, followed by printing the glcd and controller statistics for the cycle.
  For details on glcd and controller statistics refer to section 5.8.12.
- Keypress 'q' quits the clock emulator.
- Any other key will resume to the emulator run mode for continuous execution of clock cycles.

5.8.4 '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
  'er'  - Resume interrupted execution
```

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_STACK_DEPTH_MAX` in listutil.c [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 nested command files. 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.
- Upon completing a script an optional run statistics overview is provided. For this refer to section 5.5, and command 'sls' in section 5.8.12.
- The 'er' command resumes an interrupted script. A command can be interrupted after its completion. A wait command can be interrupted while waiting but is considered to be completed upon resuming. A script that was aborted due to a syntax or parsing error cannot be resumed.

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 ~29 seconds for the script to finish)
time=28.800 sec, cmd=11685, line=11686, avgLine=406
mchron> # Rerun the script again, but interrupt it using 'q' and then resume it
mchron> e s ../script/time-hm.txt
quit
--- stack trace ---                                         Press 'q'
0:../script/time-hm.txt:28:    wte w
-:mchron:-:e s ../script/time-hm.txt
time=6.840 sec, cmd=2785, line=2786, avgLine=407
mchron> # Resume the script at where it was interrupted and complete it
mchron> er
time=21.940 sec, cmd=8900, line=8900, avgLine=406
mchron>
```

5.8.5 'g' – Graphics data

The graphics data commands allow to manage graphics bitmap data buffers.

```
Commands:
  'gbc' - Copy graphics buffer
          Arguments: <from> <to>
              from: 0..9
              to: 0..9
  'gbi' - Show graphics buffer info
          Argument: <buffer>
              buffer: -1..9 (-1 = all, other = buffer)
  'gbr' - Reset graphics buffer
          Argument: <buffer>
              buffer: -1..9 (-1 = all, other = buffer)
  'gbs' - Save graphics buffer to file
          Arguments: <buffer> <width> <filename>
              buffer: 0..9
              width: 0..128 (0 = max 80 chars/line, other = elements/line)
              filename: full path or relative to startup directory mchron
  'gci' - Load lcd controller image data
          Arguments: <buffer> <format> <x> <y> <xsize> <ysize>
              buffer: 0..9
              format: 'b','w','d' (b = 8-bit, w = 16-bit, d = 32-bit)
              x: 0..127
              y: 0..63
              xsize: 1..128
              ysize: 1..64
  'gf'  - Load file graphics data
          Arguments: <buffer> <format> <filename>
              buffer: 0..9
              format: 'b','w','d' (b = 8-bit, w = 16-bit, d = 32-bit)
              filename: full path or relative to startup directory mchron
  'gfi' - Load file image data
          Arguments: <buffer> <format> <xsize> <ysize> <filename>
              buffer: 0..9
              format: 'b','w','d' (b = 8-bit, w = 16-bit, d = 32-bit)
              xsize: 1..128 (image width)
              ysize: 1..64 (image height)
              filename: full path or relative to startup directory mchron
  'gfs' - Load file sprite data
          Arguments: <buffer> <xsize> <ysize> <filename>
              buffer: 0..9
              xsize: 1..128 (sprite width)
              ysize: 1..32 (sprite height)
              filename: full path or relative to startup directory mchron
```

Usage specifics:

- There are three command to load bitmap data from a file.
  Command 'gf' is the generic one and loads sequential bitmap data into a buffer.
  Command 'gfi' assumes that the bitmap file contains a single image spread over one or more image frames.
  Command 'gfs' assumes the bitmap data contains sprites, being multiple individual images, each stored in a single image frame.
- There is a single command to load bitmap data from the LCD controllers.
  Command 'gci' loads LCD controller image data into a buffer.
- When a buffer is loaded using 'gf', you can only use command 'pb' to paint its contents on the LCD.
- When a buffer is loaded using 'gfi' or 'gci', you can use commands 'pb' or 'pbi' to paint its contents on the LCD. Command 'pbi' paints the image using the buffer image metadata.
- When a buffer is loaded using 'gfs', you can use commands 'pb' or 'pbs' to paint its contents on the LCD. Command 'pbs' paints the sprite using the buffer sprite metadata.

Example:

```
mchron> le
mchron> # First load double-word bitmap image data of size 128x64 into buffer 2
mchron> gfi 2 d 128 64 ../script/bitmap/image-dword.txt
data origin     : ../script/bitmap/image-dword.txt
data loaded at  : 2023-12-20 12:27:40
data format     : double word (4 bytes per element)
data elements   : 256 (1024 bytes)
data contents   : image (single fixed size image)
content details : image size 128x64 pixels requiring 2 frame(s)
mchron> # As the data is registered as image data we use command pbi to paint it
mchron> # on the lcd
mchron> pbi 2 0 0
mchron> # However, we can also use the generic pb command to achieve the same,
mchron> # but it takes more work as pb can draw only one frame at a time
mchron> le
mchron> pb 2 0 0 0 0 128 32
mchron> pb 2 0 32 128 0 128 32
mchron> # Fill buffer 3 with some lcd controller image data in word format
mchron> gci 3 w 5 5 37 29
data origin     : lcd controllers
data loaded at  : 2023-12-20 12:30:35
data format     : word (2 bytes per element)
data elements   : 74 (148 bytes)
data contents   : image (single fixed size image)
content details : image size 37x29 pixels requiring 2 frame(s)
mchron> # And save it to a file (after which we could read it back in as a 37x29
mchron> # image using command bfi)
mchron> gbs 3 0 ../script/bitmap/myImgDump-word.txt
mchron> # Paint a part of the 2nd frame of our original image buffer 2 to
mchron> # demonstrate we can access any rectangular subset of the bitmap data
mchron> le
mchron> pb 2 15 32+5 128+15 5 98 20
mchron> # Now load the Mario sprite data in buffer 5
mchron> gfs 5 9 12 ../script/bitmap/mario.txt
data origin     : ../script/bitmap/mario.txt
data loaded at  : 2023-12-20 12:32:40
data format     : word (2 bytes per element)
data elements   : 36 (72 bytes)
data contents   : sprite (multiple fixed size image frames)
content details : sprite size 9x12 pixels, 4 frame(s)
mchron> # As the data is registered as sprite data we use command pbs to paint it
mchron> # on the lcd. Paint the last sprite frame (frame 3) from the buffer.
mchron> le
mchron> pbs 5 0 0 3
mchron> # And achieve the same using generic command pb
mchron> pb 5 0 15 9*3 0 9 12
mchron> # Paint all sprites in the buffer
mchron> pb 5 0 30 0 0 9*4 12
mchron>
```

5.8.6 'h' – Help

The help commands provide generic help utilities.

```
Commands:
  'h'   - Help
  'hc'  - Search command
          Arguments: <search> <pattern>
              search: 'a','d','n','.' (a = arg, d = descr, n = name, . = all)
              pattern: mchron command search regex 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]. It can search on command name, command argument name, command description or all at once.
- 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 name 'pd' (paint dot)
mchron> hc n pd
------------------------
command: pd (paint dot)
usage  : pd <x> <y>
         x: 0..127
         y: 0..63
handler: doPaintDot()
------------------------
registered commands: 1
mchron> # Find the commands that include 'read' in its command name,
mchron> # command argument names or command description.
mchron> hc . read
------------------------
command: lr (read data from active lcd controller)
usage  : lr <variable>
         variable: word of [a-zA-Z_] characters, 'null' = ignore
handler: doLcdRead()
------------------------
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>
```

5.8.7 '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 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>
```

5.8.8 '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:
  'las' - Set active lcd controller
          Argument <controller>
              controller: 0, 1
  'lbs' - Set lcd backlight brightness
          Argument: <backlight>
              backlight: 0..16 (0 = dim .. 16 = bright)
  'lcr' - Reset lcd cursor administration
  'lds' - Switch lcd controller display on/off
          Arguments: <controller-0> <controller-1>
              controller-0: 0 = off, 1 = on
              controller-1: 0 = off, 1 = on
  'le'  - Erase lcd display
  'lge' - Edit glut lcd display
  'lgg' - Set glut graphics options
          Arguments: <pixelbezel> <gridlines>
              pixelbezel: 0 = off, 1 = on
              gridlines: 0 = off, 1 = on
  'lgs' - Set glut window size
          Arguments: <axis> <size>
              axis: 'w','h' (w = width, h = height)
              size: 66..2080 (width-range = 130..2080, height-range = 66..1056)
  'lhr' - Reset glut glcd pixel highlight
  'lhs' - Set glut glcd pixel highlight
          Arguments: <x> <y>
              x: 0..127
              y: 0..63
  'li'  - Inverse lcd display and draw colors
  'lng' - Set ncurses graphics options
          Argument: <backlight>
              backlight: 0 = off, 1 = on
  'lp'  - Print controller state/registers
  'lr'  - Read data from active lcd controller
          Argument: <variable>
              variable: word of [a-zA-Z_] characters, 'null' = ignore
  'lss' - Set controller lcd start line
          Arguments: <controller-0> <controller-1>
              controller-0: 0..63
              controller-1: 0..63
  'lw'  - Write data to active lcd controller
          Argument: <data>
              data: 0..255
  'lxs' - Set active lcd controller x cursor
          Argument: <x>
              x: 0..63
  'lys' - Set active lcd controller y cursor
          Argument: <yline>
              yline: 0..7
```

Usage specifics:

- (Re)setting a glcd pixel highlight using the 'lhr' and 'lhs' commands can also be achieved using right-clicks in the OpenGL2/GLUT LCD device.
- 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 'lge' command provides a rudimentary method to edit the LCD controller data contents using the glut LCD device and a mouse. Its main purpose is to edit graphics image data that is displayed on the LCD. After being modified

using this command, the modified data can be read into a graphics data buffer. Refer to section 5.8.5 to manipulate graphics data buffers, section 5.8.10 to draw graphics bitmap data on the LCD, and graphics6.txt [script] for an actual use-case.

- The 'lgg' command enables drawing pixel bezels in the OpenGL2/GLUT LCD device, but will only do so only after a certain minimum Monochron window pixel width is reached to avoid blurred pixels.
  For this, the minimum aspect ratio width is set by `#define GLUT_PIXBEZEL_WIDTH_PX` in lcdglut.c [firmware/emulator].
- The 'lgs' command allows to resize the OpenGL2/GLUT window on window pixels. By specifying the size of either the window width or height, the size of the other axis is calculated to maintain the proper window aspect ratio.
- The ks0108.c [firmware] module keeps a software administration of the active LCD controller and the controller y cursor for graphics speed optimization purposes. However, controller commands 'lcs', 'lr', 'lw', 'lxs' and 'lys' make changes to the active controller and its cursor that bypass this software administration. This means that after using, for example, the 'lys' 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 and sync the hardware and software controller and controller cursor administration.

Example:

```
mchron> # Paint a clock so we have something on the LCD display
mchron> cs 11
mchron> # Set the glut window size via the width-axis (and apply the proper
mchron> # height-axis value to maintain the x/y aspect ratio)
mchron> lgs w 700
mchron> # Set LCD backlight brightness to a medium setting and back to full
mchron> lbs 8
mchron> lbs 16
mchron> # Disable variable backlight in the ncurses LCD device
mchron> lng 0
mchron> # Enable pixel bezels and gridlines in the glut LCD device.
mchron> # Note that the glut window must be made big enough to show the bezels.
mchron> lgg 1 1
mchron> # Set and reset a glcd pixel highlight in the glut LCD device.
mchron> # You can also achieve this using a right-click in the glut window.
mchron> lhs 20 13
mchron> lhr
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> # Double-click some pixels in the glut window to toggle them on/off.
mchron> # Press 'q' to exit the edit mode.
mchron> lge
<edit: double-click left button = toggle pixel, q = quit>
mchron> # Set cursor in controller 0 and write a byte to the LCD at bottom left
mchron> lcs 0
mchron> lxs 0
mchron> lys 7
mchron> lw 85
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> # so ignore the first read by not copying its result into a variable.
mchron> lxs 0
mchron> lr null
```

```
mchron> lr myLcdByte
myLcdByte=85
mchron> # Reset and sync hardware and software controller and 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> # Note that the active lcd controller is indicated by a '*' after the
mchron> # controller number, in this case controller 0.
mchron> lp
controllers:
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***
*	The active controller. Use command 'lcs' to set its value.
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 active 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 active LCD controller.
x	The current cursor x position. Use command 'lxs' to set its value on the active LCD controller.
y	The current cursor y line (containing 8 vertical pixels) position. Use command 'lys' to set its value on the active LCD controller.

Table 23: Controller state and register values

5.8.9 'm' – Monochron

The Monochron commands will run an emulated Monochron application, the Monochron configuration pages and allow interacting with the stubbed Monochron eeprom.

```
Commands:
  'm'   - Run Monochron application
          Argument: <startmode>
              startmode: 'c','r' (c = single cycle, r = run)
  'mc'  - Run Monochron configuration
          Arguments: <startmode> <timeout> <restart>
              startmode: 'c','r' (c = single cycle, r = run)
              timeout: 0 = off, 1 = on
              restart: 0 = off, 1 = on
  'mep' - Print Monochron eeprom settings
  'mer' - Reset Monochron eeprom
  'mew' - Write data to Monochron eeprom
          Arguments: <address> <data>
              address: 0..1023
              data: 0..255
```

Usage specifics:

- Note that the 'm' command runs the Monochron application, being similar to the firmware that is uploaded to an actual Monochron clock. It uses the collection of clocks configured in array monochron in anim.c [firmware].
- Although the Monochron configuration pages are part of the Monochron application, command 'mc' allows to run the configuration separately. Argument `<timeout>` allows to ignore the built-in 10 seconds inactivity timeout timer. Argument `<restart>` can restart the configuration pages after moving beyond its latest configuration item, making it easier to continue testing without having to re-enter the 'mc' command.
- 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 or a configuration page.
- The Monochron eeprom settings are initialized at startup of mchron and are changed when using the Monochron configuration or command 'mew'.

Example:

```
mchron> # Reset the Monochron eeprom and initialize with default values
mchron> mer
eeprom reset
mchron> # Lower the LCD backlight brightness saved in eeprom
mchron> mew 1 8
mchron> # Print the eeprom contents. Notice the reduced backlight brightness
mchron> # in address 1 (EE_BRIGHT).
mchron> mep
eeprom:
monochron eeprom offset = 0 (0x000)
monochron eeprom status = initialized
byte address name            value
 0   0x000   EE_INIT          90 (0x5a)
 1   0x001   EE_BRIGHT         8 (0x08)
 2   0x002   EE_VOLUME         1 (0x01)
 3   0x003   EE_REGION         5 (0x05)
 4   0x004   EE_TIME_FORMAT    1 (0x01)
 :
12   0x00c   EE_ALARM_HOUR3   10 (0x0a)
13   0x00d   EE_ALARM_MIN3    30 (0x1e)
14   0x00e   EE_ALARM_HOUR4   11 (0x0b)
15   0x00f   EE_ALARM_MIN4    45 (0x2d)
mchron>
```

```
mchron> # Start an emulated Monochron application.
mchron> # Note: running command 'mc' will result in the same keypress menu.
mchron> m r
<run: info = e/p/r/t, hardware = a/m/s/+, c = cycle, h = help, q = quit>
emuchron monochron emulator keypress overview
info:                                                     Press 'h'
  e = print monochron eeprom settings
  p = print performance statistics
  r = reset performance statistics
  t = print time/date/alarm
hardware:
  a = toggle alarm switch
  m = menu button
  s = set button
  + = + button
other:
  c = execute single application clock cycle
  h = provide emulator help
  q = quit

<run: info = e/p/r/t, hardware = a/m/s/+, c = cycle, h = help, q = quit>
```

Monochron emulator specifics:
- Keypress 'a' is identical to command 'tat'
- Keypress 'e' is identical to command 'mep'.
- Keypress 'p' is identical to command 'sp'.
- Keypress 'r' is identical to command 'sr'.
- Keypress 't' is identical to command 'tp'.

Using keypress 'c', the emulator moves to a single application clock cycle mode. This will also change the user prompt.

```
mchron> # Start an emulated Monochron application in single cycle mode,
mchron> # or, when in run mode press 'c' to enter single cycle mode
mchron> m c
<cycle: c = next cycle, p = next cycle + stats, q = quit, other key = run>
cycle statistics:
time    : 13:50:19 (hh:mm:ss)                                 Press 'p'
date    : 21/02/2024 (dd/mm/yyyy)
glcd    : dataWrite=194, dataRead=212, addressSet=64
        : ctrlSet=29
ctrl-0  : write=35 (6%), read=38 (79%), display=0 (-%)
        : x=17 (100%), y=8 (100%), startline=0 (-%)
ctrl-1  : write=159 (32%), read=174 (87%), display=0 (-%)
        : x=47 (100%), y=22 (100%), startline=0 (-%)

<cycle: c = next cycle, p = next cycle + stats, q = quit, other key = run>
```

Single application clock cycle keypress specifics:
- Keypress 'c' executes the next application clock cycle.
- Keypress 'p' executes the next application clock cycle, followed by printing the glcd and controller statistics for the cycle.
  For details on glcd and controller statistics refer to section 5.8.12
- Keypress 'q' quits the Monochron emulator.
- Any other key will resume to the emulator run mode for continuous execution of clock cycles.

5.8.10 'p' – Paint

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

```
Commands:
  'pa'  - Paint ascii
          Arguments: <x> <y> <font> <orientation> <xscale> <yscale> <text>
              x: 0..127
              y: 0..63
              font: '5x5p' = 5x5 proportional, '5x7m' = 5x7 monospace
              orientation: 'b' = bottom-up vertical, 'h' = horizontal,
                           't' = top-down vertical
              xscale: 1..64
              yscale: 1..32
              text: ascii text
  'pb'  - Paint buffer
          Arguments: <buffer> <x> <y> <xo> <yo> <xsize> <ysize>
              buffer: 0..9
              x: 0..127
              y: 0..63
              xo: 0..1023 (data element x offset)
              yo: 0..31 (data element y offset)
              xsize: 0..128
              ysize: 0..32
  'pbi' - Paint buffer image
          Arguments: <buffer> <x> <y>
              buffer: 0..9
              x: 0..127
              y: 0..63
  'pbs' - Paint buffer sprite
          Arguments <buffer> <x> <y> <frame>
              buffer: 0..9
              x: 0..127
              y: 0..63
              frame: 0..127
  'pc'  - Paint circle
          Arguments: <x> <y> <radius> <pattern>
              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: <x> <y> <radius> <pattern>
              x: 0..127
              y: 0..63
              radius: 0..32
              pattern: 0 = full, 1 = half, 2 = 3rd up, 3 = 3rd down
                       4 = inverse, 5 = blank
  'pd'  - Paint dot
          Arguments: <x> <y>
              x: 0..127
              y: 0..63
  'pl'  - Paint line
          Arguments: <xstart> <ystart> <xend> <yend>
              xstart: 0..127
              ystart: 0..63
              xend: 0..127
              yend: 0..63
```

```
Commands (cont'd):
  'pn'  - Paint number
         Arguments: <x> <y> <font> <orientation> <xscale> <yscale> <number>
                    <format>
             x: 0..127
             y: 0..63
             font: '5x5p' = 5x5 proportional, '5x7m' = 5x7 monospace
             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: <x> <y> <xsize> <ysize>
             x: 0..127
             y: 0..63
             xsize: 1..128
             ysize: 1..64
  'prf' - Paint rectangle with fill pattern
         Arguments: <x> <y> <xsize> <ysize> <align> <pattern>
             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 = blank
  'ps'  - Set draw color
         Argument: <color>
             color: 0, 1 (0 = off (black), 1 = on (white))
  'psb' - Set draw color to background color
  'psf' - Set draw color to foreground color
```

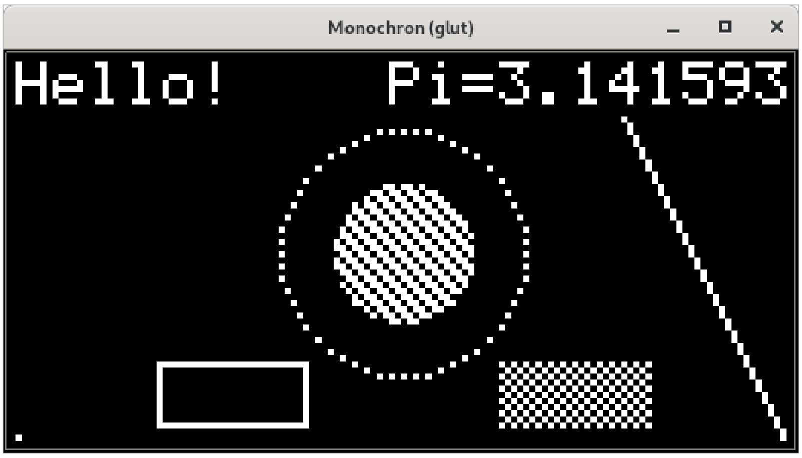
Usage specifics:

- 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> # Set paint color to foreground color
mchron> psf
mchron> # Paint ascii
mchron> pa 1 1 5x7m h 1 1 Hello!
hor px=36
mchron> # Paint dotted circle
mchron> pc 64 32 20 1
mchron> # Set paint color to background color
mchron> psb
mchron> # Paint filled circle
mchron> pcf 64 32 11 3
mchron> # Set paint color back to foreground color
mchron> psf
mchron> # Paint dot at bottom left
mchron> pd 1 62
mchron> # Paint line
mchron> pl 100 10 126 62
mchron> # Paint number
mchron> pn 62 1 5x7m h 1 1 pi Pi=%f
hor px=66
mchron> # Paint rectangle
mchron> pr 24 50 25 11
mchron> # Paint filled rectangle
mchron> prf 80 50 25 11 0 1
mchron>
```

These commands will produce the following output.



For using the paint graphics buffer commands 'pb', 'pbi' and 'pbs' refer to section 5.8.5 where these commands are used in combination with associated graphics buffer commands.

5.8.11 '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:
  'rb'  - Repeat break
  'rc'  - Repeat continue
  '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 a 'rn' command in the very same file. Commands 'rb' and 'rc' must reside in a 'rf'-'rn' block.
- From a functional point of view, commands 'rb' and 'rc' are identical to resp. C statements 'break' and 'continue'.
- 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
time=30.161 sec, cmd=5858, line=5858, avgLine=194
mchron>
```

5.8.12 '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:
  'sls' - Set list runtime statistics
          Argument: <enable>
            enable: 0 = off, 1 = on
  'sp'  - Print application statistics
  'sr'  - Reset application statistics
```

Usage specifics:

- Command 'sls' allows to enable/disable the reporting of list execution runtime statistics upon its completion. It is enabled by default. Refer to section 5.5 for an example.
- 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, addressSet=18797
       : ctrlSet=2904
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 24: Emuchron stub statistics

The statistics for the glcd interface are as follows:

KPI	***Description***
addressSet	The number of times that a controller cursor is set. It is administered by counting the number of calls to `glcdSetXAddress()`. Note: Changing a controller cursor will always result in a command to update the controller display x position, but for optimization purposes the controller y position is only updated when it differs from the software maintained y position administration. For this, refer to the description of the x and y statistics in the controller statistic section below. Incrementing the counter will occur upon calling `glcdSetAddress()`, or internally in ks0108.c [firmware] when an explicit update of the cursor is required. Note that auto-incrementing a cursor is an automatic hardware action that does not require an explicit cursor update.
ctrlSet	The number of times that the active LCD controller is switched. It is administered by counting the number of calls to `glcdControlSelect()`.
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. For this, refer to the description of the read statistic in the controller statistic section below.
dataWrite	The number of pixel bytes written to the LCD. It is administered by counting the number of calls to `glcdDataWrite()`.

Table 25: 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 an on/off change of the display.
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 LCD byte, which is a hardware limitation. 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.
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 of the startline.
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 26: 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 somewhat higher than the lcdByteRx KPI. This is explained by a number of controller and backlight commands sent to the GLUT interface during runtime or at mchron initialization time.

KPI	***Description***
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.
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 27: 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 28: Emuchron ncurses statistics

5.8.13 't' – Time

The time commands allow setting, resetting and reporting the time, alarm time and alarm switch, and the date as used in mchron.

```
Commands:
  'tap' - Set alarm switch position
          Argument: <position>
              position: 0 = off, 1 = on
  'tas' - Set alarm time
          Arguments: <hour> <min>
              hour: 0..23
              min: 0..59
  'tat' - Toggle alarm switch position
  'tdr' - Reset clock date to system date
  'tds' - Set date
          Arguments: <day> <month> <year>
              day: 1..31
              month: 1..12
              year: 0..99 (year in 20xx)
  'tf'  - Flush Monochron time and date to active clock
  'tg'  - Get date/time
          Arguments: <varday> <varmonth> <varyear> <varhour> <varmin> <varsec>
              varyear: word of [a-zA-Z_] characters, 'null' = ignore
              varmonth: word of [a-zA-Z_] characters, 'null' = ignore
              varday: word of [a-zA-Z_] characters, 'null' = ignore
              varhour: word of [a-zA-Z_] characters, 'null' = ignore
              varmin: word of [a-zA-Z_] characters, 'null' = ignore
              varsec: word of [a-zA-Z_] characters, 'null' = ignore
  '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.
- When setting a date, 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 'tds' command verifies whether the requested date is valid. For example, date April 31st will be rejected.
- The 'tg' command allows to retrieve only those time/date elements that are needed. For example, if only the date is relevant specify for the time element variables 'null'.

Example for alarm:

```
mchron> # Set alarm time to 14:51
mchron> tas 14 51
time  : 17:03:34 (hh:mm:ss)
date  : 28/10/2023 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : off
```

```
mchron> # Set alarm switch to 'on'
mchron> tap 1
time  : 17:03:50 (hh:mm:ss)
date  : 28/10/2023 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : on
mchron> # Toggle alarm switch
mchron> tat
time  : 17:04:10 (hh:mm:ss)
date  : 28/10/2023 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : off
mchron>
```

Example for date:

```
mchron> # Set our own date to Jan 27 2035
mchron> tds 27 1 35
time  : 17:08:10 (hh:mm:ss)
date  : 27/01/2035 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : off
mchron> # Reset to system date
mchron> tdr
time  : 17:08:26 (hh:mm:ss)
date  : 28/10/2023 (dd/mm/yyyy)
alarm : 14:51 (hh:mm)
alarm : off
mchron> # September 31 does not exist
mchron> tds 31 9 23
date? invalid
mchron> # Extract the current date into variables
mchron> tg year month day null null null
mchron> vp .
day=28        month=10      year=23
registered variables: 3
mchron>
```

Example for time:

```
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  : 28/10/2023 (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  : 28/10/2023 (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  : 28/10/2023 (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  : 28/10/2023 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : off
mchron>
```

5.8.14 'v' – Variable

The `mchron` interpreter 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 regex 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: f=key
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 .
index=400     key=40
registered variables: 2
mchron> # Reset all variables
mchron> vr .
reset variables: 2
mchron> vp .
registered variables: 0
mchron>
```

5.8.15 '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 commands are used in many scripts to temporarily halt script execution or wait a while after updating the LCD display with new information.
- For waiting time within loops, such as in script time-hm.txt [script], wait timer commands 'wts'/'wte' 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 averaged 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.

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> # When the timer expires it is automatically restarted.
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>
```

5.8.16 '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

The `mchron` interpreter 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 2 8+7 5x5p h 3 2 hello world!

2  cmdLineExecute(): Execute a single command line

3  cmdArgInit(): Init scan + get command and associated command dictionary

cmdCommand_t cmdGroupPaint[] =
{ { "pa", PCCTRLTYPE(PC_CONTINUE),
    CMDARGS(argPaintAscii), CMDHANDLER(doPaintAscii), "paint ascii" }, .. };

4  cmdArgGet(): Get command arguments using command dictionary +
               verify non-numeric domain arguments

// cmdArgValidateChar(): 'b', 'h' or 't'
DOMAIN(domCharOrient, DOM_CHAR_VAL, "bht", 0, 0, "b = bottom-up, h =
  horizontal, t = top-down");

cmdArg_t argPaintAscii[] =                              cmdLine_t cmdLine.args[] =
{ { ARG_NUM,    "x",           &domNumPosX },                  "2\n",
  { ARG_NUM,    "y",           &domNumPosY },                  "8+7\n",
  { ARG_STRING, "font",        &domWordFont },                 "5x5p",
  { ARG_CHAR,   "orientation", &domCharOrient },   =>          "h",
  { ARG_NUM,    "xscale",      &domNumScaleX },                "3\n",
  { ARG_NUM,    "yscale",      &domNumScaleY },                "2\n",
  { ARG_STRING, "text",        &domStrText } };                "hello world!" };

5  cmdArgPublish(): Publish arguments + verify numeric domain arguments

// cmdArgValidateNum(): range 0..63
DOMAIN(domNumPosY, DOM_NUM_RANGE, NULL, 0, GLCD_YPIXELS - 1, NULL);

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

6  Command "pa" handler = doPaintAscii(): Paint text string

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

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, being 'pa', and based on that retrieves its command dictionary entry.
From this point on all scanning and parsing functionality will be driven by information retrieved 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 'pa' command dictionary. Each argument is copied into a dynamic array within the `cmdLine_t` structure. For character and string arguments additional functionality for an argument value is provided via structure `cmdDomain_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 'orientation' character argument must have either value 'b', 'h' or 't'. 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 strings, characters and doubles. They are respectively `argString[]`, `argChar[]` and `argDouble[]`. In the example above, the `ARG_STRING` font argument is added in `argString[]`. Note that numeric arguments are now run through the expression evaluator first, and then have their domain value checked, when configured. In this case the 'y' number expression '8+7\n', resulting in 15, must be in the range 0..63.

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 now called.
In `doPaintAscii()`, after converting the font argument into an enum value, function `glcdPutStr3()` is called to paint the requested text string on the LCD.
Note that all command handler functions are defined in mchron.c [firmware/emulator].

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
time=73.714 sec, cmd=5858, line=5858, avgLine=79
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.

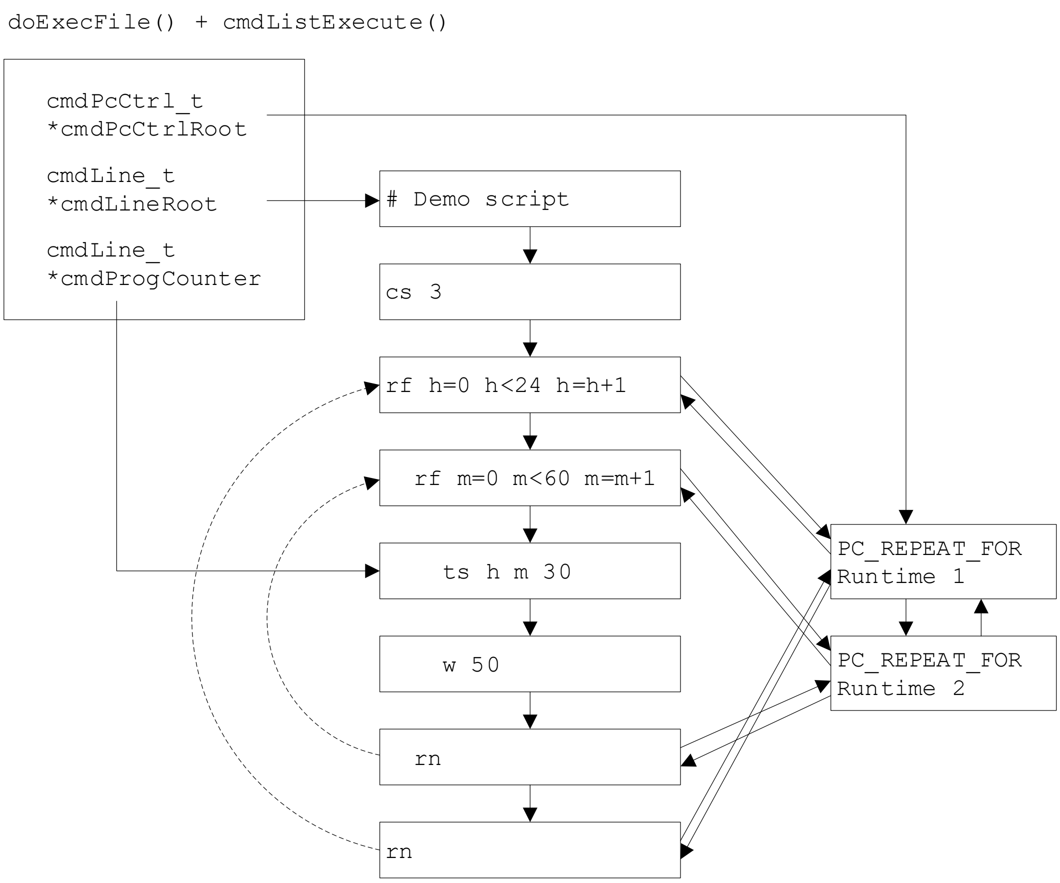


Figure 9: Creating and executing an mchron command list

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 `doExecFile()`.
- In `doExecFile()`, via function `cmdStackPush()`, 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 `cmdStackPush()`, 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. This will deliver the list resume pointer for the present stack. The stack can be resumed for execution using command 'er'.
- 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 for the current stack level when a list element has no pointer to a next one. The current stack level is then popped from the stack.
- When list execution is completed, command and control block list cleanup will take place after which `doExecFile()` 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 front-end 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 global.h [firmware].

```
// Debugging macros.
// Select to report generic application 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 global.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 3.7 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 3.7)
$ 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.

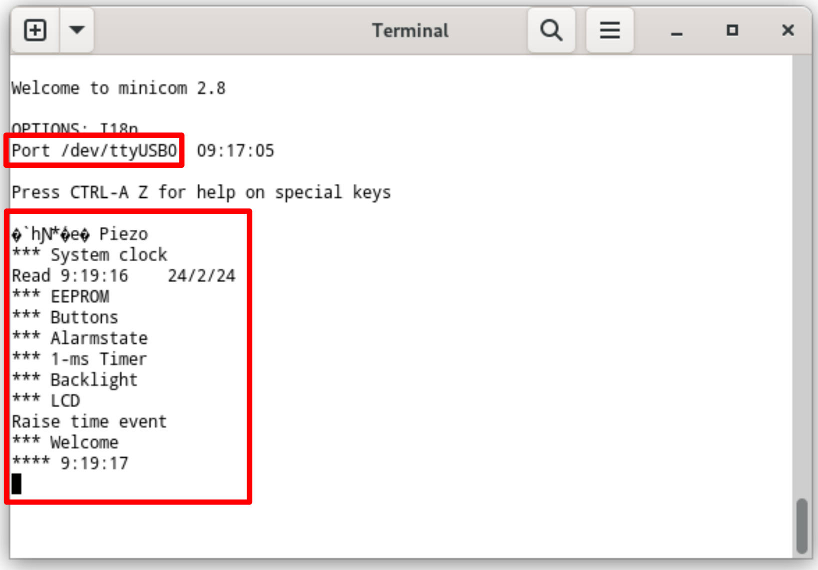


Figure 10: Minicom receiving Monochron debug strings

– For help on minicom enter '<ctrl>a', followed by keypress 'z'. See below.

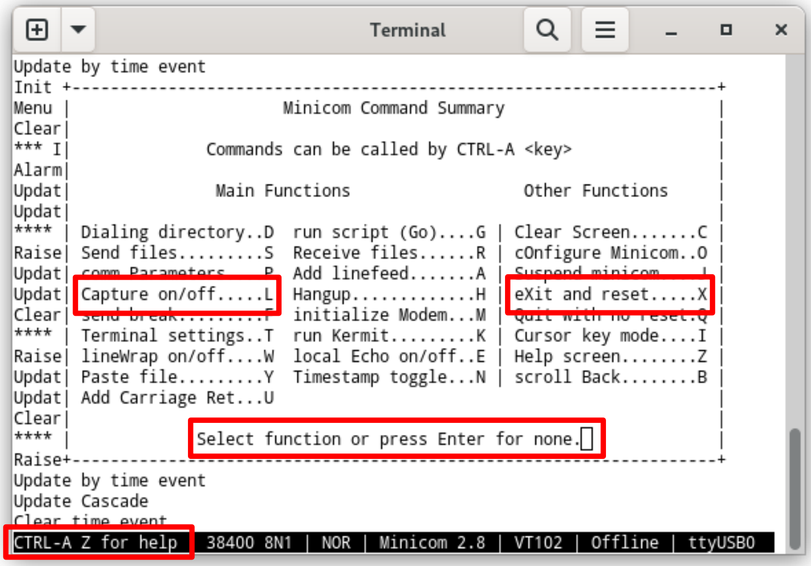


Figure 11: Minicom command summary via '<ctrl>az'

– 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 can 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 an error when invoked with the -d flag, indicating that debug output logging cannot be used. See below.

```
$ ./mchron -d debug.log
mchron: -d: master debugging is off
- Assign value 1 to "#define DEBUGGING" in global.h [firmware] and rebuild
  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' (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, except that in the Emuchron log each line is prefixed with the current system time for convenience.

```
$ tail -f debug.log
13:33:52.926 **** logging started
13:33:53.023 Read 13:33:53    10/8/22
13:33:53.023 Clear time event
13:33:53.023 Raise time event
13:33:57.750 *** UART
13:33:57.750 *** Piezo
13:33:57.750 *** System clock
13:33:57.750 Read 13:33:57    10/8/22
13:33:57.750 *** EEPROM
13:33:57.750 *** Buttons
13:33:57.750 *** Alarmstate
13:33:57.750 *** 1-ms Timer
13:33:57.750 *** Backlight
13:33:57.750 *** LCD
13:33:57.822 *** Welcome
13:33:59.379 *** Start initial clock
13:33:59.379 Clear time event
13:33:59.379 **** 13:33:59
13:33:59.379 Raise time event
13:33:59.379 Update by time event
13:33:59.380 Init Cascade
(etc..)
```

6.3 Debugging Emuchron using gdb

Emuchron and its mchron tool 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 front-end gui's available. In this manual we consider the use of Gede.

When using only the GLUT LCD device, the mchron program can be loaded and started in gdb with Gede 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 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, the

mchron terminal will have undesired side-effects on the ncurses tty, for example upon resizing the mchron terminal.
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 a front-end gui) to the running mchron process.
When this ncurses/gdb debug startup sequence method is applied, no other limitations apply.

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

6.3.2 Debugging Emuchron with ncurses device using Gede

First startup mchron and make sure there is a command prompt. Note the mchron process id.

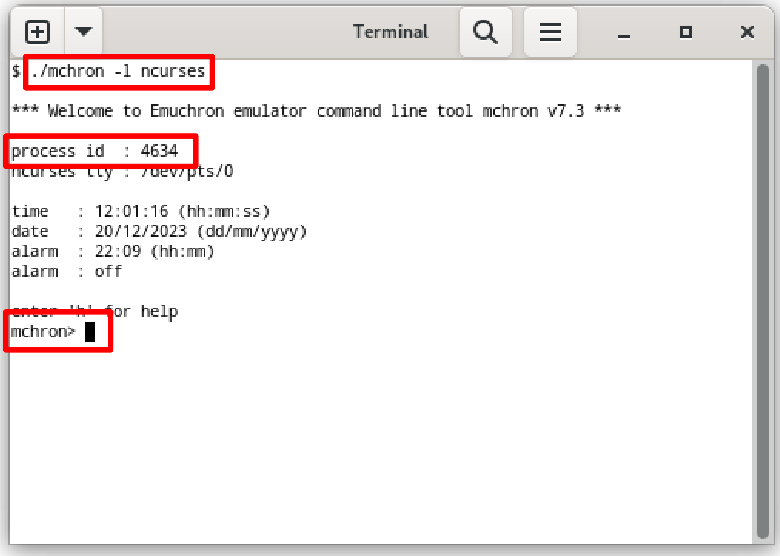


Figure 12: Prepare to debug mchron with an ncurses LCD device

Start Gede using its program launcher, as created per instructions in appendix D.2.
Or, execute the command copied from commands.txt [support] item #5.

When Gede is started it needs to be configured.
At the top select the provided default, being the user home folder, use /usr/bin/gdb as debugger, and specify the full location of the mchron executable.
At the bottom select to connect to a running process and enter the mchron process id, or select it from the process list selector at the right.
Click the 'OK' button to connect to the mchron process.
See below.

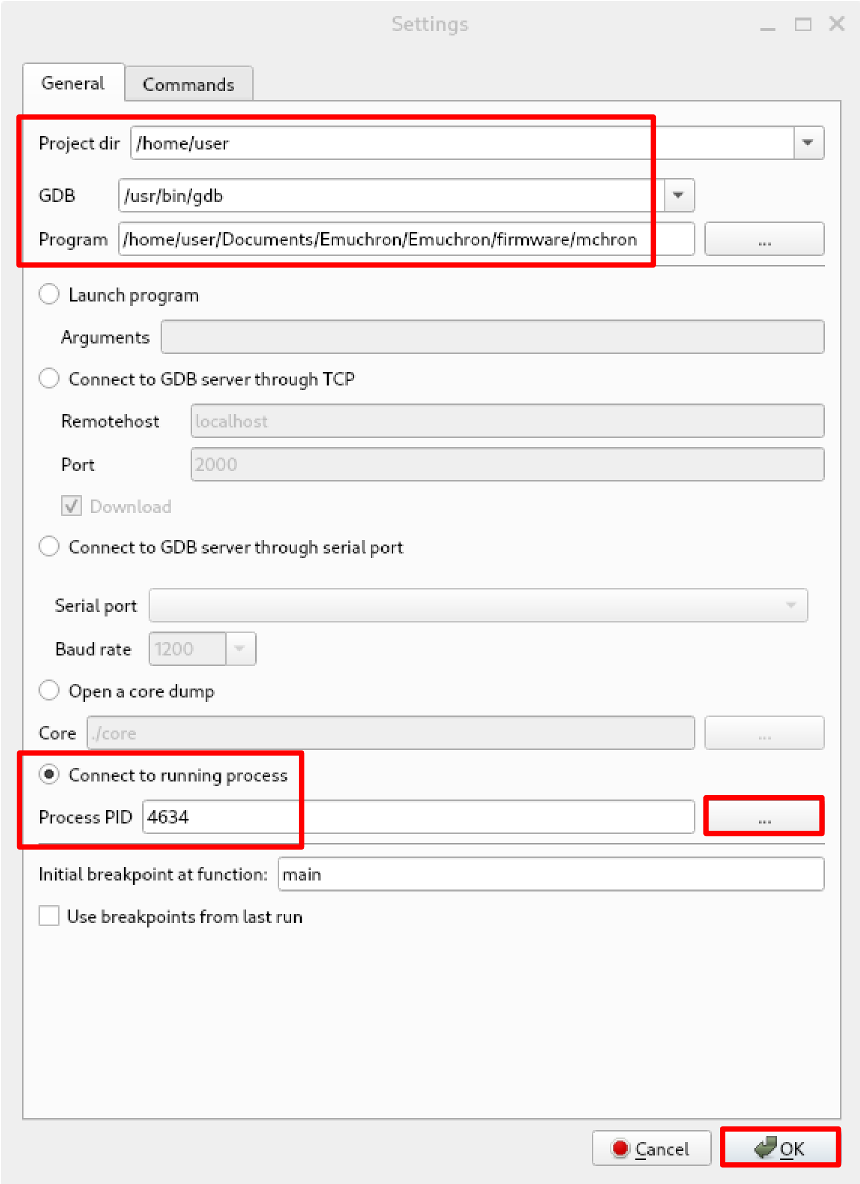


Figure 13: Setting up Gede to attach to the mchron process

Gede now tries to attach to the process, but may not always succeed in this. The reason for this is that the mchron process may not be 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 now under full control of Gede.

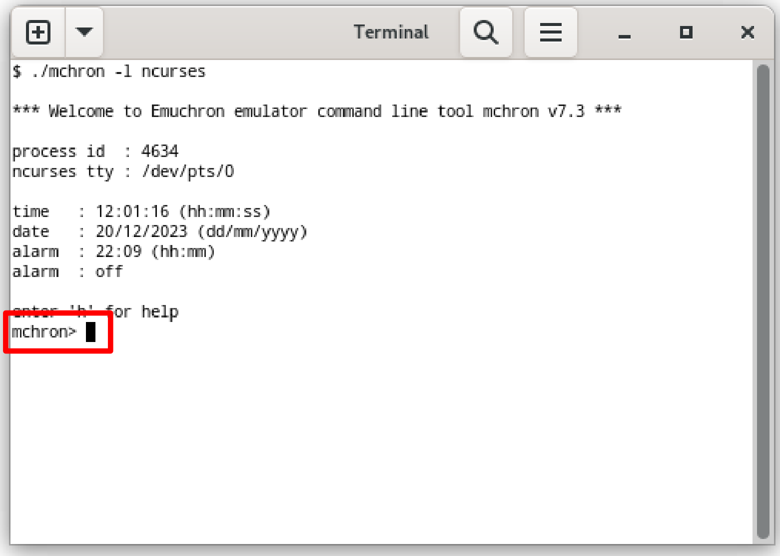


Figure 14: mchron is put under control of Gede

In Gede, on the left, we are now be able to browse the application files. A quick way to go to `main()` is selecting menu item 'Search→Go To main()'. Or, at the bottom-right, one can select a thread and select `main()` or another function from the stack. See below.

When a source or function is selected, the source file is opened. You are now able to jump to a function within the file, set and remove breakpoints by double-clicking a source line number, as well as verifying local and global data.

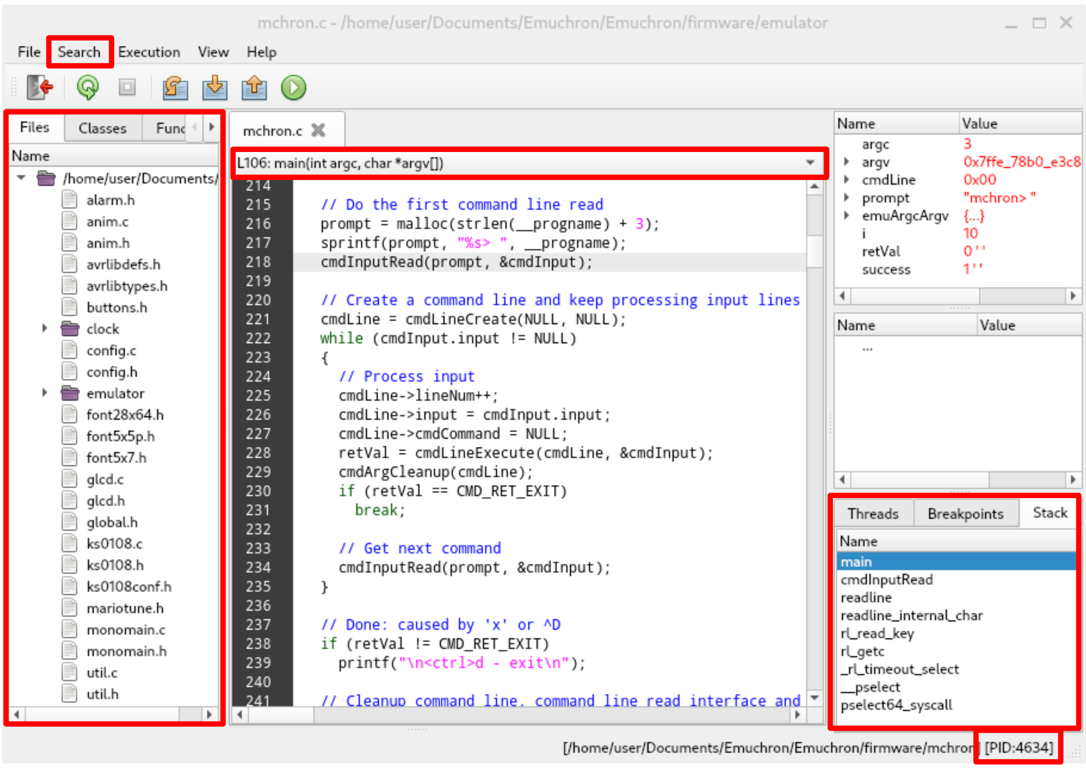


Figure 15: Debugging mchron in Gede

6.3.3 Debugging an mchron coredump file

The method of debugging an mchron coredump file does not differ from debugging a coredump of another application. For example, refer to the Gede settings form that has an option to point to a coredump file.

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 dedicated code sections for Monochron and Emuchron.

Another difference to consider is the way memory is handled. The Atmel environment supports compiler directive `progmem` that allows storing static data in flash program memory, thereby saving precious RAM. In Linux this directive is unsupported and in Emuchron it is ignored at compile time. This means that in Linux Emuchron all `progmem` related data and functionality is redirected to use RAM. In Atmel Monochron however, one must be careful not to mix-up references to `progmem` and regular RAM related data. For example, consider `glcdBitmap()` in glcd.c [firmware]. This function requires to provide a data origin type for the bitmap data to be displayed, being either progmem (`DATA_PMEM`) or RAM (`DATA_RAM`). When specified incorrectly, in Emuchron bitmap data may still display properly, but will fail to do so on an actual Monochron clock.

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 non-integer 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, division by zero, or 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 second 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 second indicator. However, for clocks with a second indicator, every now and then this results in visually choppy behavior of the second indicator by showing an unusually long or short time to switch from one second 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 second indicator behavior in a clock.
The RTC scan frequency is controlled using the following defines in monomain.c [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 config file $HOME/.config/mchron/tty. The content of this file is created upon starting a Monochron ncurses terminal. For this, refer to section 3.8.2.

What mchron cannot anticipate is the situation where the Monochron terminal is deleted while the tty config still exists, and its tty gets re-used by another bash shell.
Upon starting mchron it is detected that the tty as read from the tty config file 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 the tty config file 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 OpenGL2/GLUT and/or ncurses screen update behavior.
Refer to section 3.9.1 and 3.9.2 for configuring several workarounds when observing issues upon playing audio in a VM.

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
time=8.506 sec, cmd=32000002, line=32000002, avgLine=3761934
mchron>
```

On the arm64 VMware hypervisor VM that is used to develop and test Emuchron the repeat loop will take about 8.5 seconds to complete.

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.

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 10, 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.
However, upon installing an avr toolchain on a non-Debian machine, please be aware that the version of `avr-gcc` may differ from the one that is used on Debian (5.4.0). It may be that upon using another version of `avr-gcc` the Monochron code will not compile.

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 5.8.3 and 5.8.9), 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 Screen dumps of example clocks

All LCD device output screen dumps in this document are taken using the default screen capture utility 'Screenshot'. 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 OpenGL2/GLUT LCD device. When using no mchron device argument the OpenGL2/GLUT device is used.
- 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 2019, (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 a screen capture utility. The resulting clock layout can be seen in appendix A.4.
- For other screen dumps using the same date and time, select another clock using command 'cs'.

```
$ ./mchron
:
mchron> cs 4
mchron> tap 1
time  : 19:20:15 (hh:mm:ss)
date  : 04/07/2021 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : on
mchron> tds 14 9 19
time  : 19:20:33 (hh:mm:ss)
date  : 14/09/2019 (dd/mm/yyyy)
alarm : 22:09 (hh:mm)
alarm : on
mchron> tas 6 45
time  : 19:20:40 (hh:mm:ss)
date  : 14/09/2019 (dd/mm/yyyy)
alarm : 06:45 (hh:mm)
alarm : on
mchron> ts 22 9 30
time  : 22:09:30 (hh:mm:ss)
date  : 14/09/2019 (dd/mm/yyyy)
alarm : 06:45 (hh:mm)
alarm : on
mchron>
```

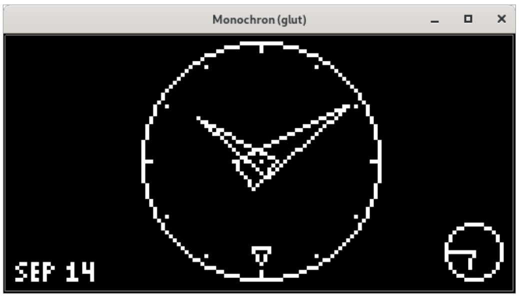
A.1 Analog clocks

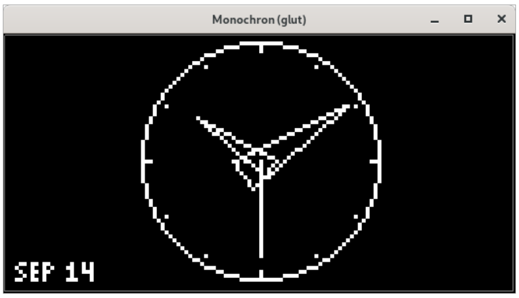
Clock Ids: `CHRON_ANALOG_HMS` and `CHRON_ANALOG_HM`

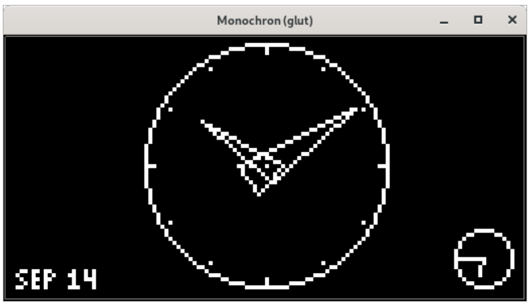
These are basic analog clocks with h/m/s or h/m time notification.
When the alarm switch is on, the alarm time will appear at the bottom right in a small analog clock. When alarming or snoozing, the alarm time will blink or show a snooze countdown timer.
There are several build options for an analog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





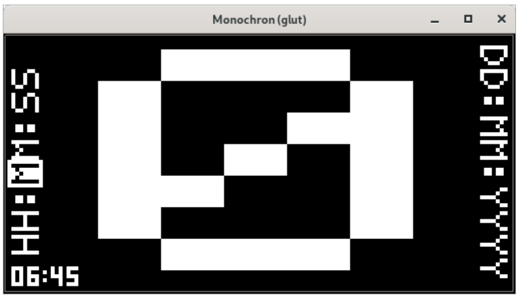


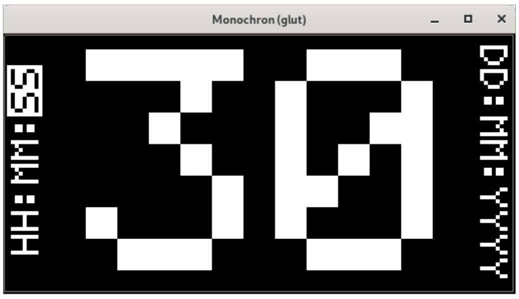
A.2 Big Digit clocks

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 or show a snooze countdown timer.

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





A.3 Dali clock

Clock Id: `CHRON_DALI`

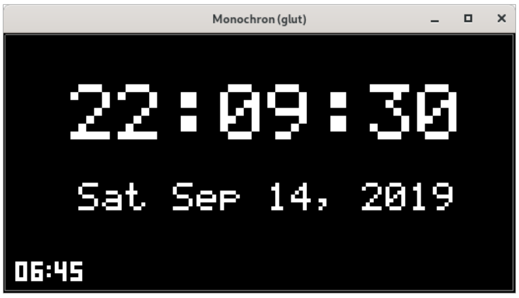
This clock is based on the xdali clock code from Multichron, available at https://github.com/CaitSith2/monochron/tree/MultiChron/firmware, which itself is based on a Monochron version of xdaliclock. Some functional and graphics features are added or modified. When alarming or snoozing, the clock will continuously switch every three seconds between the current and alarm time. For code refer to dali.c [firmware/clock].

A.4 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 or show a snooze countdown timer.
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.5 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 75 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 or show a snooze countdown timer.
For code refer to example.c [firmware/clock].

A.6 Marioworld clock

Clock Id: `CHRON_MARIOWORLD`

This clock is based on the great MarioChron clock available at https://github.com/techninja/MarioChron. A lot of clock code has been rewritten and several game play graphics features are added. The clock is a showcase for the glcd graphics function `glcdBitmap()`, introduced in Emuchron v6.1.
There are two versions available for the Mario sprite frames. For this, refer to the definition of static progmem array `mario[]`.
When the alarm switch is on, the clock date at the top-left is replaced with the alarm time. When alarming or snoozing, both '? '-blocks will blink.
For code refer to marioworld.c [firmware/clock].

A.7 Mosquito clock

Clock Id: `CHRON_MOSQUITO`

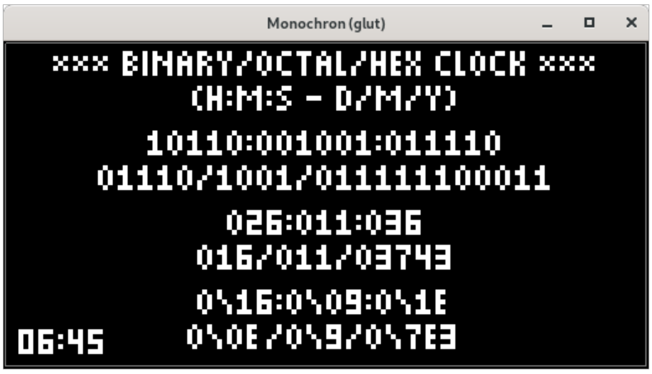
This clock implements the time as separate elements that randomly float over the LCD display. After starting the clock it will initially show the time with static elements. After a few seconds however, first the seconds element will start moving, then the minutes element and finally the hours element 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 or show a snooze countdown timer.
For code refer to mosquito.c [firmware/clock].

A.8 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 or show a snooze countdown timer.
For code refer to nerd.c [firmware/clock].



A.9 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.
Over time, 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.
In Emuchron v5.0 the clock was subject to a major code revision that saved ~30% on code and resulting object size.
When the clock is alarming, whereas the original code will inverse the clock layout every second, in Emuchron the alarming state is identified by flashing the center of the paddles.
For code refer to pong.c [firmware/clock].

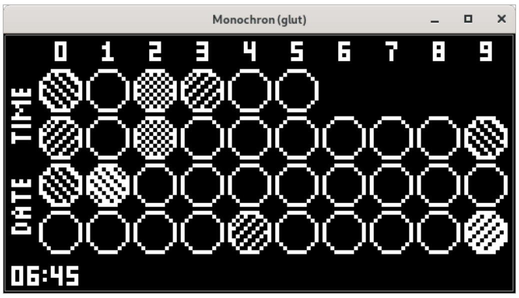
A.10 Puzzle clock

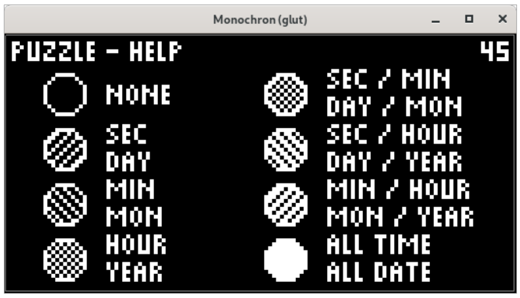
Clock Id: `CHRON_PUZZLE`

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 or show a snooze countdown timer.

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





A.11 QR clocks

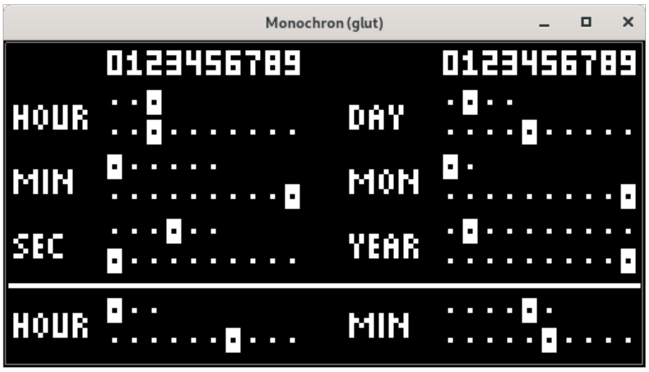
Clock Ids: `CHRON_QR_HMS` and `CHRON_QR_HM`

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 or show a snooze countdown timer.
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.12 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.13 Spotfire and QuintusVisuals clocks

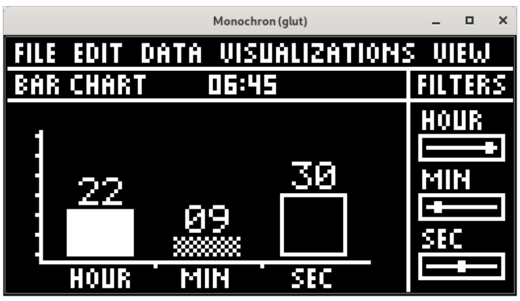
Clock Ids: `CHRON_BARCHART`, `CHRON_CASCADE`, `CHRON_CROSSTABLE`, `CHRON_LINECHART`, `CHRON_PIECHART`, `CHRON_SPEEDDIAL`, `CHRON_SPIDERPLOT`, `CHRON_THERMOMETER` and `CHRON_TRAFLIGHT`

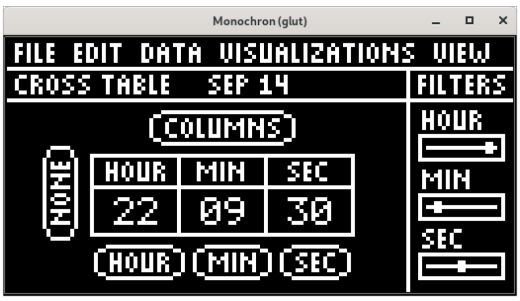
Spotfire (https://www.spotfire.com) is a professional business analytics tool that provides insight in very large amounts of data using visualizations. QuintusVisuals is an extension to Spotfire and provides additional visualization types.
The clocks below are minimalistic implementations of the Spotfire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

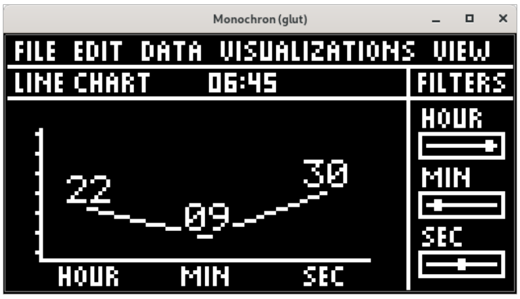
The (non-functional) header of a visualization represents the header of Spotfire. However, the clocks include a hard-coded calendar that will change the header on specific dates to a dedicated message. See the spider plot example for March 14th below.
The filter panel on the right side contains sliders for the hour, minutes and seconds elements that are similar to those in 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 or show a snooze countdown timer.

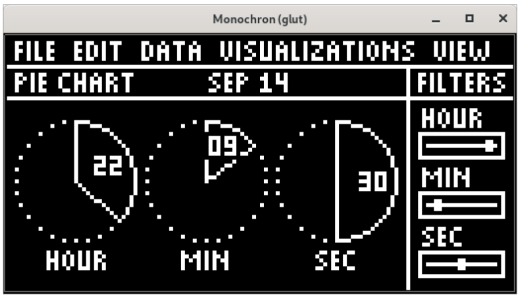
For code refer to spotfire.c (generic module for all Spotfire and QuintusVisuals clocks, including the calendar), barchart.c, cascade.c, crosstable.c, linechart.c, piechart.c, speeddial.c, spiderplot.c, thermometer.c and trafficlight.c [firmware/clock].

The Spotfire clocks are as follows:

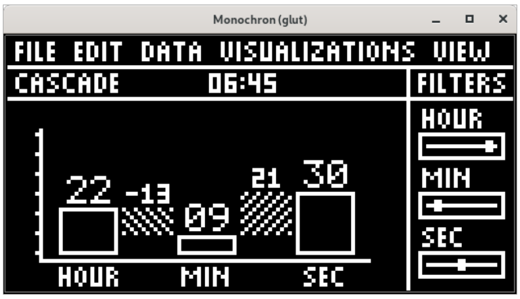


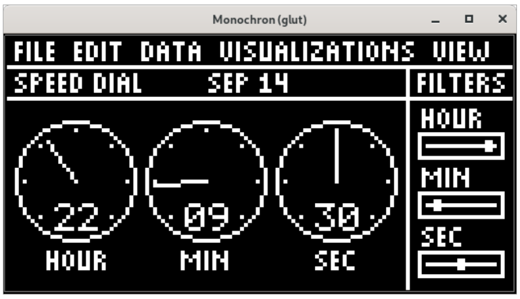


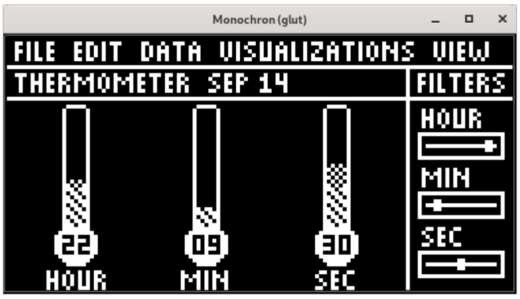


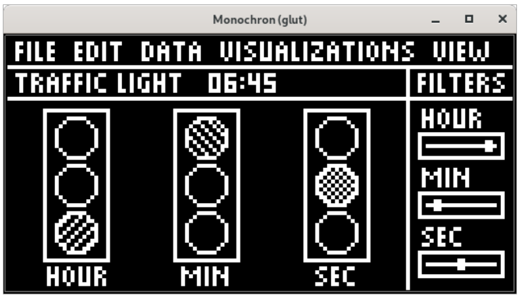


The QuintusVisuals clocks are as follows:









A.14 Wave clock

Clock Id: `CHRON_WAVE`

This clock shows the time and date as a continuously waving banner. The hour and minute time separator will blink every other second.
When the alarm switch is on, the alarm time will appear at the bottom center.
When alarming or snoozing, the alarm time will blink or show a snooze countdown timer.
For code refer to wave.c [firmware/clock].


Monochron (glut)
22:09 14:09
06:45

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 v8.0, as well as bugfixes in glcd graphics code, were ported back to Emuchron v1.0/v6.1. Each test set is run with display brightness setting 4.

High-level glcd graphics function name	***Draw performance Emuchron v1.0/v6.1***	***Draw performance Emuchron v8.0***
`glcdDot()`	1.0x (v1.0)	2.6x
`glcdLine()`	1.0x (v1.0)	4.4x
`glcdCircle2()`	1.0x (v1.0)	5.3x
`glcdFillCircle2()`	1.0x (v1.0)	4.4x
`glcdRectangle()`	1.0x (v1.0)	4.3x
`glcdFillRectangle2()`	1.0x (v1.0)	4.9x
`glcdPutStr3()`	1.0x (v1.0)	3.3x
`glcdPutStr3v()`	1.0x (v1.0)	6.9x
`glcdPutStr()`	1.0x (v1.0)	1.8x
`glcdBitmap()`	1.0x (v6.1)	1.1x

Table 29: Draw performance Emuchron v8.0 (avr-gcc 5.4.0)

B.1 Test results Emuchron v7.2 vs Emuchron v8.0

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

Version avr-gcc	***Emuchron v7.2 Object size (bytes)***	***Emuchron v8.0 Object size (bytes)***
5.4.0 (Debian 11/12)	.data: 944 .text: 24558 **Total: 25502**	.data: 944 .text: 24512 **Total: 25456**

Table 30: Default firmware size Emuchron v7.2 vs Emuchron v8.0

Some remarks on the build statistics:

– The v8.0 build is slightly smaller than the v7.2 build due to a few code refinements.

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.

In Emuchron v8.0 no changes were made in the glcd component. The performance test run on v8.0 code yields the same results as the test run for v7.2.

Test	Test Name	Emuchron v7.2		Emuchron v8.0	
1	glcdDot-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:59 1146880 2293760 2302720 17920	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:59 1146880 2293760 2302720 17920
2	glcdDot-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:58 688128 2752512 2069760 12096	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:58 688128 2752512 2069760 12096
3	glcdLine-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:53 347986 1763047 457879 140614	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:53 347986 1763047 457879 140614
4	glcdLine-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 1378380 1612500 470580 116999	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 1378380 1612500 470580 116999
5	glcdCircle2-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 1830474 2114760 624200 282748	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 1830474 2114760 624200 282748
6	glcdCircle2-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:47 1829880 2260440 861120 3588	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:47 1829880 2260440 861120 3588
7	glcdFillCircle2-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:57 1622752 927242 1260710 217287	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:57 1622752 927242 1260710 217287
8	glcdFillCircle2-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:56 1131200 1575600 1232200 2020	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:56 1131200 1575600 1232200 2020
9	glcdRectangle-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 1605168 2169426 1128516 81012	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 1605168 2169426 1128516 81012
10	glcdRectangle-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2568000 2589600 403800 187200	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2568000 2589600 403800 187200
11	glcdFillRectangle2-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:48 1958040 2237760 559440 74592	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:48 1958040 2237760 559440 74592

Test	***Test Name***	***Emuchron v7.2***		***Emuchron v8.0***	
12	glcdFillRectangle2-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:47 3789500 1419340 158470 110240	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:47 3789500 1419340 158470 110240
13	glcdFillRectangle2-03	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 4205376 1068032 83440 83440	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 4205376 1068032 83440 83440
14	glcdFillRectangle2-04	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 4078368 1035776 80920 80920	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:46 4078368 1035776 80920 80920
15	glcdPutStr3-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2127384 2161152 67536 67536	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2127384 2161152 67536 67536
16	glcdPutStr3-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:45 2981160 1514240 70980 70980	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:45 2981160 1514240 70980 70980
17	glcdPutStr3-03	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2090640 2124360 67440 67440	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2090640 2124360 67440 67440
18	glcdPutStr3-04	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2840400 1732644 75744 75744	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:44 2840400 1732644 75744 75744
19	glcdPutStr3-05	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:43 2202480 2237440 69920 69920	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:43 2202480 2237440 69920 69920
20	glcdPutStr3v-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:58 800520 242062 209660 19060	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:58 800520 242062 209660 19060
21	glcdPutStr3v-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 2331504 610632 157284 37008	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:52 2331504 610632 157284 37008
22	glcdPutStr3v-03	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:57 874160 249760 156100 2230	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:57 874160 249760 156100 2230

Test	Test Name	Emuchron v7.2		Emuchron v8.0	
23	glcdPutStr3v-04	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:53 2261280 608392 215360 53840	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:53 2261280 608392 215360 53840
24	glcdPutStr-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:45 6058080 0 96160 96160	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:45 6058080 0 96160 96160
25	glcdBitmap-01	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:51 2513472 916192 141540 71004	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:51 2513472 916192 141540 71004
26	glcdBitmap-02	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:49 1844224 1901856 115264 1	Time 5.4.0: dataWrite: dataRead: addressSet: ctrlSet:	01:49 1844224 1901856 115264 1

Table 31: Performance test run Emuchron v7.2 vs Emuchron v8.0

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.

C.1 Setting up a Monochron terminal profile in Debian

– From the main menu start a terminal and select 'Hamburger→Preferences'.

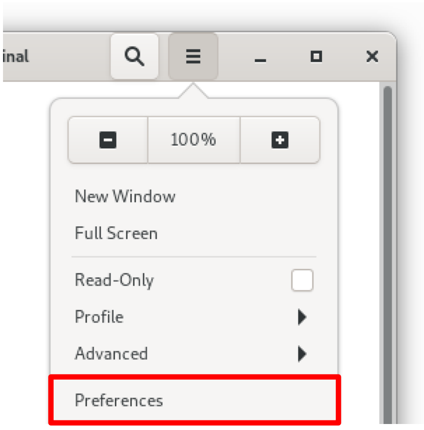


Figure 16: Access terminal preferences

– In the window that pops up, on the left under 'Profiles' at the default 'Unnamed' profile select to 'Clone'. Name the cloned profile 'Monochron'.

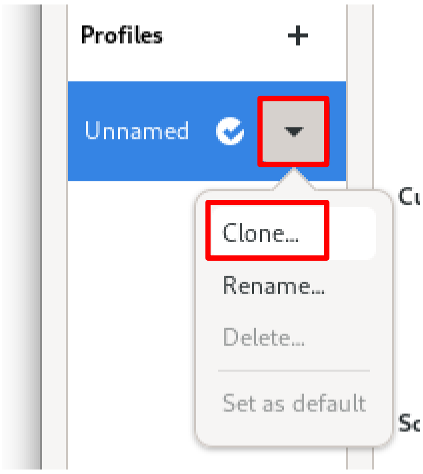


Figure 17: Create new terminal profile by cloning the default

For the new 'Monochron' profile several configuration tabs are available. Per tab set the options **exactly** as per screen dump and info below.

– Tab 'Text'.
  Set the number of columns and rows to respectively 258 and 66.
  Select a custom font:
  - Debian 11: use 'Monospace' with point size 2.
  - Debian 12: use 'Noto Mono' with point size 2.
  The combination of the font and very small point size allows creating square pixels with a proper size.

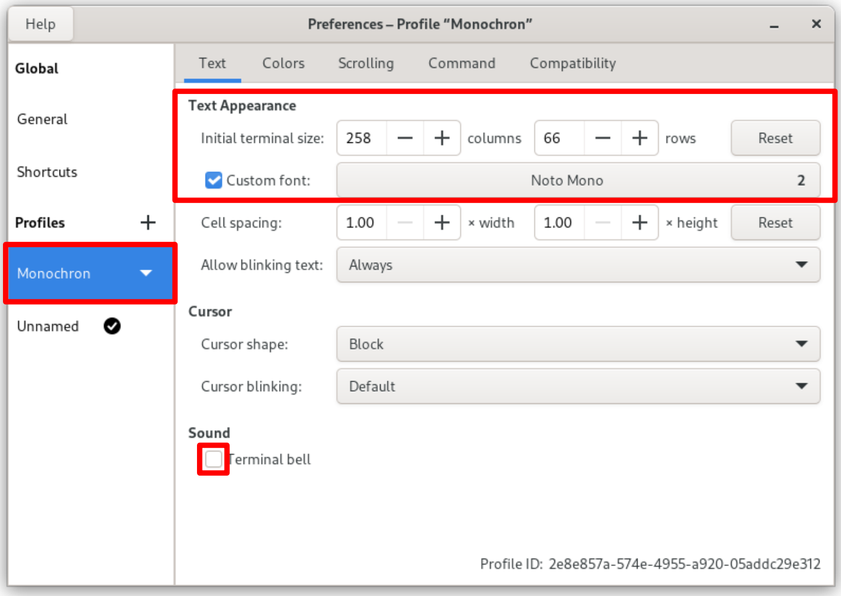


Figure 18: Terminal profile tab 'Text'

– Tab 'Colors'.
  Deselect the system theme colors option and set the built-in schema to 'White on black'.

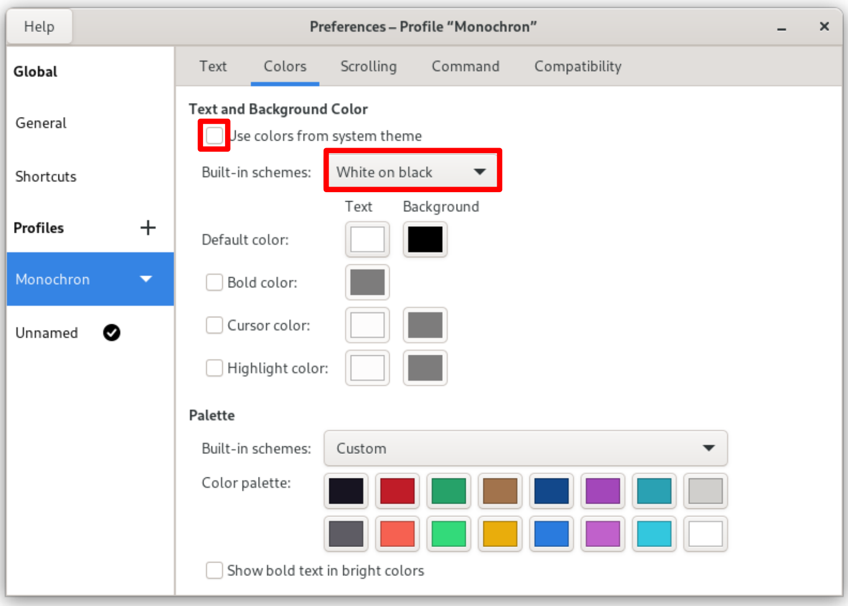


Figure 19: Terminal profile tab 'Colors'

– Tab 'Scrolling'.
  Disable all options.

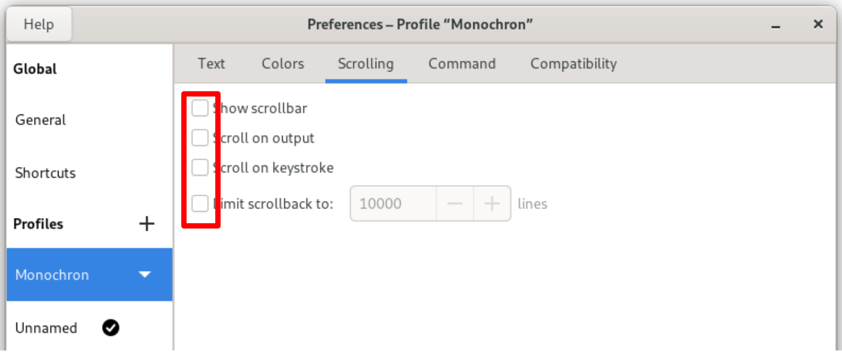


Figure 20: Terminal profile tab 'Scrolling'

– Tab 'Command'.
  Enable to run a custom command.
  The command to use can be copied from commands.txt [support] item #1.
  It will copy the tty of the new terminal to ~/.config/mchron/tty, and start a clean bash shell.

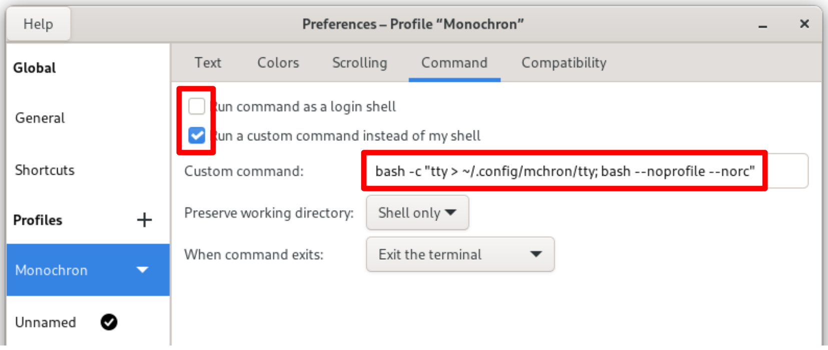


Figure 21: Terminal profile tab 'Command'

– Tab 'Compatibility'.
  No changes are needed on this tab.

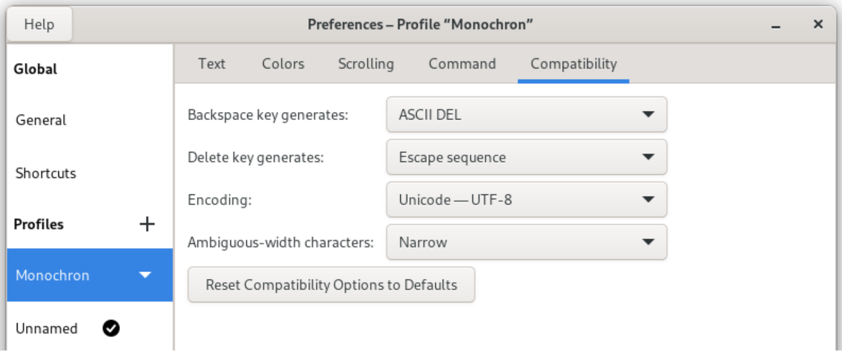


Figure 22: Terminal profile tab 'Compatibility'

Close the form to complete the setup of the Monochron terminal profile.

D Setting up main menu program launchers

In this appendix is explained how to setup program launchers for relevant Emuchron related tools. Creating a main menu launcher requires Debian package alacarte (https://en.wikipedia.org/wiki/Alacarte) that is installed by packages.txt [support].

Some configuration tips apply. By default, Debian starts with the Activities menu. One can also opt for a more classic Applications menu where applications are stored in application folders and subfolders. Package alacarte makes use of the folder structure.

Switch between the Activities menu and Applications menu as follows.

- Debian 11: From the main menu start the 'Tweaks' application and go to tab Extensions.
  Debian 12: From the main menu start the 'Extensions' application.
- At the top-right enable extensions.
- Enable extension 'Applications menu' to use the Applications menu.
  Note: Disable this extension to switch back to the Activities menu.

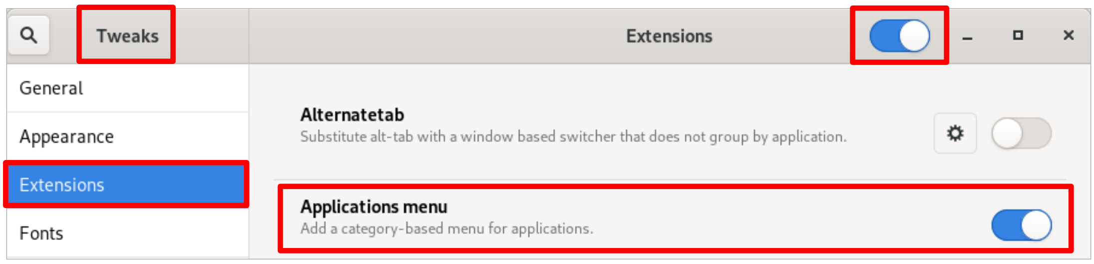


Figure 23: Debian 11 - Enabling the 'Applications' main menu

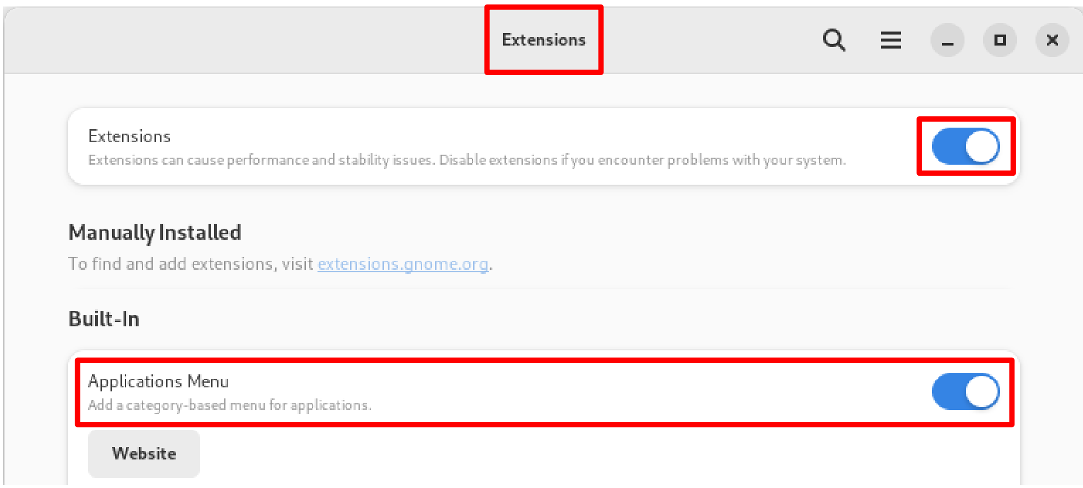


Figure 24: Debian 12 - Enabling the 'Applications' main menu

D.1 Setting up a Monochron ncurses terminal launcher

Setup the launcher as follows.

- From the main menu select application 'Main Menu'. When using the Applications menu it is found in folder System Tools.
  In the form that opens, on the left click on folder 'Programming' and then, on the right, click 'New Item' to create a new launcher.
- Provide a name and a comment (see below).

- In the launcher on the left click the icon to select a new icon. The icon selected here is:
  Debian 11: /var/lib/app-info/icons/debian-bullseye-main/48x48/termit_utilities-terminal.png
  Debian 12: /var/lib/app-info/icons/debian-bookworm-main/48x48/termit_utilities-terminal.png
- The command to use can be copied from commands.txt [support] item #2.
- Close the form. The main menu now contains in the Programming section an entry named 'Monochron'.

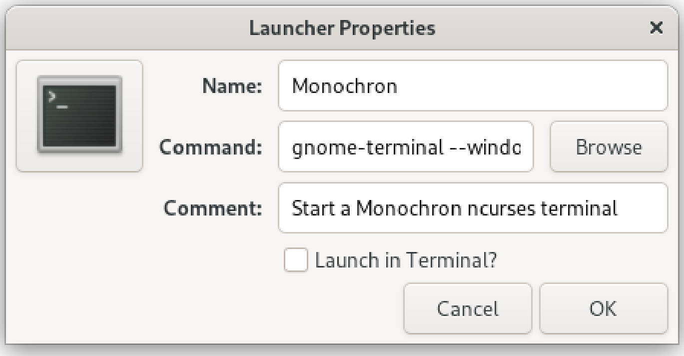


Figure 25: Launcher properties for Monochron ncurses terminal

D.2 Setting up a Gede debugger launcher

Setup the launcher as follows.

- From the main menu select application 'Main Menu'.
  In the form that opens, on the left click on folder 'Programming' and then, on the right, click 'New Item' to create a new launcher.
- In the launcher on the left click the icon to select a new icon. The icon selected here is:
  Debian 11: /var/lib/app-info/icons/debian-bullseye-main/48x48/gnome-control-center_org.gnome.Settings.png
  Debian 12: /var/lib/app-info/icons/debian-bookworm-main/48x48/gnome-control-center_org.gnome.Settings.png
- The command to use can be copied from commands.txt [support] item #4.
- Close the form. The main menu now contains in the Programming section an entry named 'Gede'.

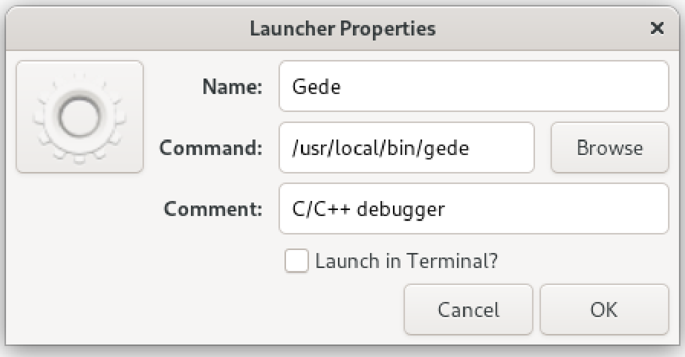


Figure 26: Launcher properties for Gede debugger