Diploma Thesis

University of Applied Sciences Augsburg
Department of Computer Science

A framework for menu structured user interfaces on embedded systems

---

Submitted by Petr Novotník, winter semester 2005/2006

Examiner: Prof. Dr. Hubert Högl
Examiner: Prof. Dr. Nikolaus Klever

Diploma Thesis

University of Applied Sciences Augsburg
Department of Computer Science

I assure that the diploma thesis is my own work and has never been used before for any auditing purposes. All used sources, additional used information and citations are quoted as such.

Petr Novotník

A framework for menu structured user interfaces on embedded systems

Petr Novotník

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Embedded systems – systems with a limited amount of resources and special hardware – are becoming more and more integrated in our lives. Examples are modern microwave ovens or heating systems, but also more complicated devices like mobile phones or PDAs. These are examples for devices where software, running on a spartan configuration of hardware, interacts with the user and takes control over the state of the system.

When we compare such devices, as a user we will notice, that a lot of them talk to us over a small display, and accept input from devices like a special keyboard or a touchscreen. We may even have the impression that the way, in which such systems represent data on the screen, is similar among them. For a programmer, such a similarity in function and design of an application suggests to write libraries. Libraries can be reused in several programs without having to struggle with the implementation details of the provided functionality. This, of course, is a great help for an application developer.

In the present work, we will introduce a library targeted at providing a framework for developing user interface applications for embedded systems. This framework, called *mlx/mexc*, concentrates on presenting data to the user in a menu structured way, which is well known from systems running on mobile phones, for example. Creating programs with this library differs from programming with libraries available for desktop computers, because of the target platform, an embedded system.

1.1 Original idea

The idea of the *mlx/mexc* framework was born in the late nineties by Hubert Högl. Indeed, there is already a realization of the framework. It can be found at [http://www.hhoegl.com/mel/mel.html] and is called “MEL/MEX”. Of course, it is no accident that both projects have similar names. In fact, the framework introduced in this paper, is just a complete rewrite of Mr. Högl’s work. Many things have been rethought, changed, and a few added, on the other hand, many things have been kept the same.

It could be interesting to compare both projects with each other and point out in what exactly they differ, however, as *mlx/mexc* has been written from scratch and is considered to be the successor of MEL/MEX, such an analysis is not given here.

1.2 How to read this document

Unlike others, the chapters in this document are not built on top of each other. However, being new to the subject, it is best to read them in sequence. The following list gives an overview of chapters considered to be essential for a complete understanding of *mlx/mexc*.

- Chapter 2 will be of interest to anybody new to the project. It serves as an general introduction to *mlx/mexc*.
- Chapter 3 is the official documentation of *mexc*, the menu library. It introduces the library's related programming details and shows how to use *mexc*. For programmers who want to create an application using the library, this chapter will definitely be of interest.
- Chapter 4 covers the *mlx* byte code compiler, which is used to create the menu specification for use with *mexc*. Beside explaining the usage of the program, this chapter also describes the produced byte code in full length.
- Chapter 5 gives a walk through the source code of *mexc* and *mlx*. Many passages of the chapter require an advanced knowledge of the C and Python programming language. This chapter is provided in the hope of giving some information about the implementation to invite developers to contribute to the project.

The last chapter – chapter 6 – gives a short overview of what has been achieved with this work, and provides some thoughts about how this project can be extended.

The following typographic conventions are used throughout this paper:

italic	Used for the names of parts of the project and to emphasize terms.
`typewriter`	Used for LCD examples, commands and source code.
sans serif	Used for external links which are additionally enclosed with brackets.

Chapter 2

Anatomy of the project

In this chapter we will define what parts make up the *mlx/mexc* project and how they cooperate with each other. Being new to *mlx/mexc* or MEL/MEX, reading this chapter is strongly recommended.

2.1 Why another library?

Currently, there are many graphical libraries for use on embedded systems. Articles like [1] and [2] give a good overview of available closed source and Open Source solutions. However, none of them is intended for use with character based liquid crystal displays (LCD), and many of them are too large to be used for tiny systems.

mlx/mexc was designed to be small to fit into memory modules with only a few kilobytes and to function with both character and graphical based LCDs. As the intent is only to display menus and interact with the user, it would be wasting to use a full featured graphical library for such a purpose.

2.2 Legal Issues

The whole project, including this document, is released under different licenses. Intentionally, free licenses have been used to grant distribution of this project in the sense of Free Software[1].

2.3 Splitting up

As the name itself denotes, this project consists out of two parts. Actually, there are more than two. Let's introduce each shortly and finally have a look at how these parts work together.

[1] For more information on the philosophy of Free Software refer to [http://www.gnu.org/philosophy/].

2.3.1 Application

The term “application” is frequently used in this document. Depending on the context it refers to different things. Whenever we talk about an application in the sense of code that is to be linked with *mexc*, we refer to the code that a programmer has to write, the `main` routine for example, in order to create a program with *mexc*.

2.3.2 Menu compiler

mlx aims to provide a way of describing and creating a menu hierarchy. Thereby, the following requirements have to be met:

- The description of a menu has to be human readable, so defining the menu hierarchy can be made with a usual text editor.
- It must be independent of a programming language to allow also non-programmers to specify a menu.
- It has to be independent of the platform the description is stored at to avoid problems with endianness (see [3]).
- It must provide an efficient way of being parsed by a computer program.
- It has to be very compact, especially because it is targeted at systems with only a few kilobytes of memory.

To meet all the requirements at the same time isn’t possible, as some of them are opposite to each other. Therefore, it has been decided to split the creation of a menu into two steps. A programmer, using a text editor, defines the structure of the menu, and then transforms it to a more computer program friendly format using a special compiler like *mlx*.

Strictly speaking, *mlx* isn’t a compiler as those introduced in [4]. However, it behaves and is used as such, and therefore, we will continue to refer to *mlx* as a “compiler” throughout this document.

mlx, explained in chapter 4 in full detail, was written entirely in Python [http://www.python.org], an interpreted, object oriented, and well-documented language available for many operating systems. The decision for the language had the following backgrounds:

- Python code is interpreted, and thus *mlx* has to be distributed in source code form. This allows everyone an insight into the program, and contributes to understanding and improving the final application.
- “Python comes with batteries included!”[2] This means that the language ships with a large quantity of library functions which can be immediately used by Python programmers. Further, as the standard library is distributed with the Python interpreter itself, it is ensured

[2] This is a popular slogan for Python and is often accompanied by a funny picture like at [http://www.python.org/doc/].

that a program using functions from this library will run also on any other machine having Python installed.

- Python supports programming of object oriented code in a very clear way. Although object orientation is available in many languages, Python brings a uniquely clear and elegant notation for OO code. This simplicity has a great impact on the understanding and writing of Python code.

2.3.3 Menu description language

The input language to the *mlx* compiler is called *melx* throughout this document and is a subset of the Extensible Markup Language (XML). The exact structure of a *melx* document is described in section 4.5 in full detail. It is no accident that XML was chosen as a base. The following considerations led to the choice of this technology:

- XML has become very popular and is a well known technology. Today, we can find examples for use of XML nearly everywhere in the world of computers. Due to the popularity of XML, programmers already familiar with XML don't have to learn a completely new language to describe a menu hierarchy.
- XML is an easy to understand subject, and is easy to be parsed by computer programs. There are only a few rules that define a well-formed XML document. Due to its popularity, there is a considerable amount of parsers written in various programming languages allowing programs to move through XML documents without having to implement a lot of code.
- XML is platform independent, and thus it greatly contributes to the portability of a program's data.
- XML is extensible. With the usage of namespaces it is possible to freely extend a document without breaking the compatibility to the old format.

More information on XML can be found at the World Wide Web Consortium ([http://www.w3.org/]) or in XML related literature like [5].

2.3.4 CMF

Surprisingly, the core component of this project is not a program or a module, but the definition of a data structure. CMF, the "Compact Menu Format", is a description of a menu hierarchy as a variable sized data structure for use with *mexc*. Special attention to the design of this structure has been paid, as changes to the format have an impact on both *mlx* and *mexc*.

The main factor of influence on the design of CMF, the output of the *mlx* compiler, have been the last three requirements given in section 2.3.2. Understanding CMF is a big step towards understanding the *mexc* library, therefore a detailed analysis of CMF is given in section 4.7.

2.3.5 Menu interpreter

The goal of the second part of *mlx/mexc*, is to implement a library to ease the creation of interactive applications for embedded systems. The library, *mexc*, is meant to take the menu description, display the menu, and handle all the details of interacting with the user. Thus, allowing the developer to concentrate on the actual algorithm of the program. Thereby, *mexc* was designed and written to meet the following requirements, which mainly have their origin in the fact, that the target platform is limited in the amount of available memory.

- The library needs to be small.
- It needs to use the stack sparingly.
- It must not rely on system services, such as dynamic memory allocation.
- It must not write to the menu description as that could be stored in a read-only memory area.
- The library must not be dependent on the hardware of the output and input devices.

The core of the library is about the menu, thus the data structure that describes the menu is of great importance to the implementation of *mexc*. Therefore, requirements to the library must also be considered by the menu data structure and the program which creates it.

To understand the implementation of *mexc*, it is important to understand CMF, the structure of the menu description in binary format, as outputted by *mlx*. What CMF makes so special, is that beside describing a menu hierarchy, it also defines memory locations where *mexc* or a program will access and store values associated with the menu. This information releases *mexc* from requiring dynamic memory allocation services, and enables an application designer to store the menu description on a read-only memory module.

Another keystone of *mexc* is to be independent on the hardware of the output and input devices over which the library communicates with the user. Therefore, it has been decided to leave the implementation of hardware dependent routines on the programmer of a concrete embedded system.

2.3.6 Simulators

There are programs, called “simulators”, which link against *mexc* and provide a way to test and debug the library. The reason for the name is that they simulate a character based LCD, thus no real display module has to be attached to the development system. Currently, there are four different simulators.

gsim is a GTK+ based program, and is the most elaborate simulator. It can load a menu description from a file and runs on GNU/Linux, MS Windows, and FreeBSD systems. Beside a virtual keyboard it provides also a so-called “inspector” which allows the developer to view and edit the current state of a menu on the fly. This application has proven to be useful for menu developers to see and test a menu before it actually gets uploaded to the target system.

csim is a curses based simulator that can be run within a terminal and does not need a graphical workstation. It has been tested on GNU/Linux systems only. This simulator is based on the code of a simulator provided by Hubert Högl for his MEL/MEX project.

dsim is a DOS based simulator, and was written to run and test *mexc* on an Intel 8088 processor under the MS DOS operating system. It has been successfully compiled with the OpenWatcom C/C++ products. More information on these products is given in Appendix A. This is actually just a proof of concept program.

psim is another simulator which was written to test *mexc* on an embedded system running the Palm OS on a Motorola DragonBall VZ processor. This is also just a proof of concept program.

2.4 Working together

Now that we know what parts make up the *mlx/mexc* project, let's look at how these work together. Figure 2.1 shows the chain of dependency between the individual components when creating a program using the *mexc* library. As the diagram shows, *mlx* takes a *melx* document as its input and

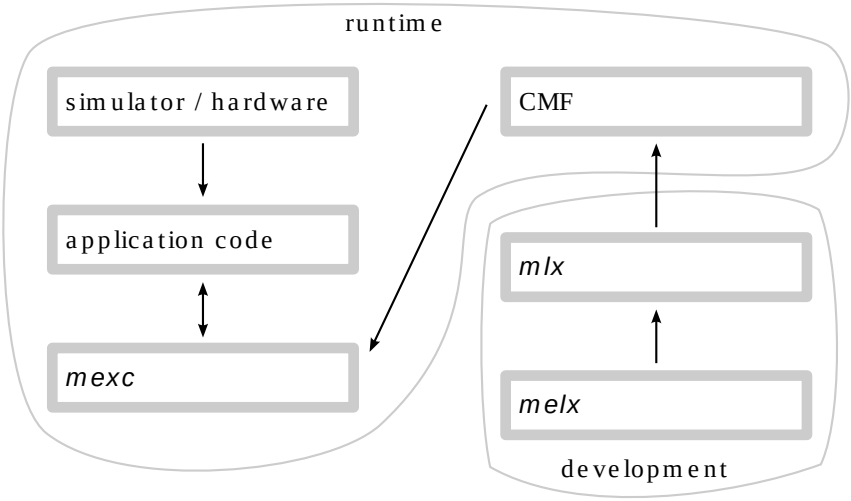

Figure 2.1: Components of a *mlx/mexc* application

outputs the binary menu description in CMF format, which in turn is used as the input to the *mexc* library. *mexc* is dependent on some application code which provides access to the actual hardware connected to the system. The application itself is dependent on the library of course. The big advantage is that the application code is not dependent on the menu description.

The two big bubbles indicate when the individual parts are used. While *mlx* is used purely as a development tool, the other parts in the "runtime" bubble are located on the target system.

There is a further detail about the way the components are working together. While having stated

that the application isn't dependent on a menu description, actually there is a dependency on the values that the menu provides. As figure 2.2 shows, an application wants to access these values and does so. Also *mexc* itself accesses these values. However, both use a different way to get hold of them. The latter uses the menu description, while the application code is directly provided with the addresses of the variables. More information on this is provided with an example in section 3.7.4.7. Even another principle is shown in the diagram. The menu description can be

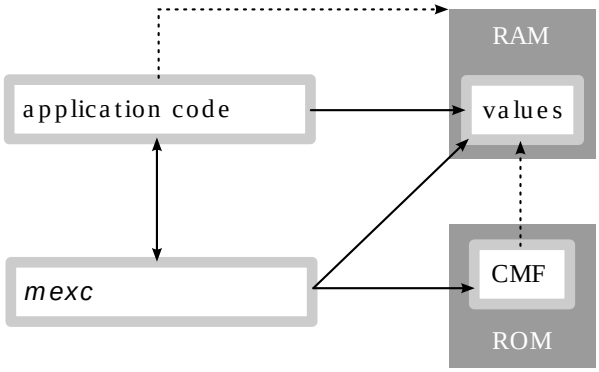

Figure 2.2: Access to values provided by a menu description

stored in a read-only memory region (ROM) as it is accessed only in reading mode. The values, on the other hand, must be located in writable memory (RAM). The base of this memory region is determined by the application code. The binary menu description only holds offsets within the provided memory region.

2.5 Menu Definition

Throughout this document we use the terms "menu", "menu table", "menu line", and "line component". Let's define what these terms address.

Wikipedia, the free encyclopedia, states, "a *menu* is a list of commands presented to the operator by a computer or communication system" [6]. A menu, as it is referred to in this document, is a structure holding lists of commands with relationships between each other. A *menu table* will always refer to a single list of commands. With a *menu line* we are referencing a single line displayed in a menu table. A menu line is a container for line components. A *line component* is the actual entity that is displayed in a menu line. There can be more components in a single menu line, and optionally they can be edited.

Figure 2.3 shows the structure of a menu table. Such a table, shortened with `mt` in the diagram, references menu lines (`ml`), which reference line components (`lc`). Thus, a menu table itself is a tree with a depth of 3. However, menu lines can also reference menu tables, in which case these are referred to as *submenus*. From a general point of view, looking at the menu table level only, the references between them – references to submenus – span also a tree. This tree is the whole menu.

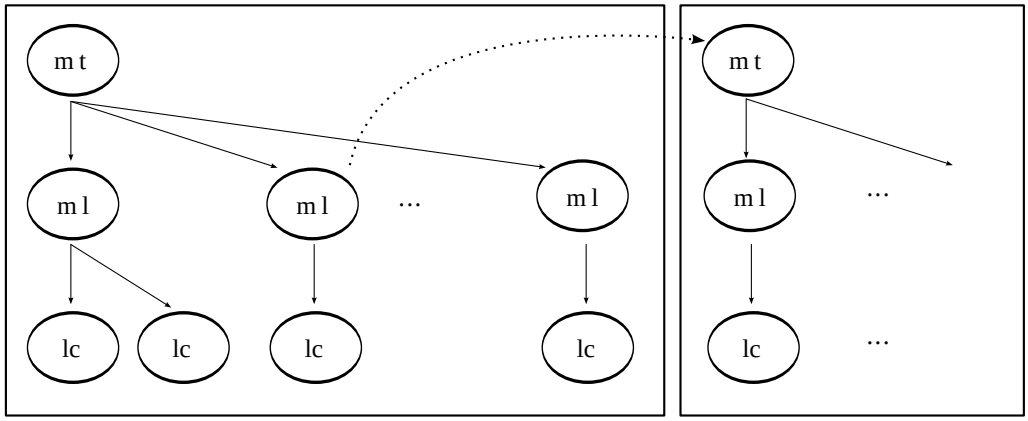

Figure 2.3: Structure of a menu table

Chapter 3

MEXC Documentation

This chapter is a slightly modified version of the official documentation for *mexc*. First it will explain how the library displays a menu and how the user can interact with it. Then it will give a detailed description of how line components are rendered and how the user can edit them. The rest of the chapter concentrates on programming related issues.

3.1 License

The *mexc* library is Free Software provided under the terms of the GNU Lesser General Public License (LGPL) [7]. The file `COPYING`, that is distributed with *mexc*, contains a copy of the GNU LGPL.

3.2 Introduction

mexc is there to display menus, navigate through them, and provide a way to let a user edit predefined input elements. Developers of embedded applications don't need to write such functionality over and over again. Together with the *mlx* compiler, which generates the byte code which *mexc* will interpret, all the developer has to do, is to describe the menu and provide custom callback handlers. However, the executor is not a self containing program and the developer needs to provide *mexc* with a few things about the environment to make it work.

Typically, a user interface for an embedded system consists of a LCD and a small keyboard connected to it. *mexc* was written for such a configuration and assumes that the display has at least 2 lines. Typical LCDs have sizes of 16x2, 16x4, 20x2, and 20x4. *mexc* can control such displays, but other sizes with at least two lines and 14 columns are possible, too. With respect to the keyboard, *mexc* expects it to have at least 5 keys. Four of them are interpreted as direction keys and one as ENTER (TAB). *mexc* can also interpret ten additional keys with the meaning of numbers ranging from 0 to 9, and an eleventh POINT key. The design of such a keyboard has been much

in vogue on modern mobile phones for some years. The direction keys, including ENTER, are often implemented with a small joystick on those devices. Figure 3.1 shows a 16x4 display with the expected keyboard and the additional eleven keys.

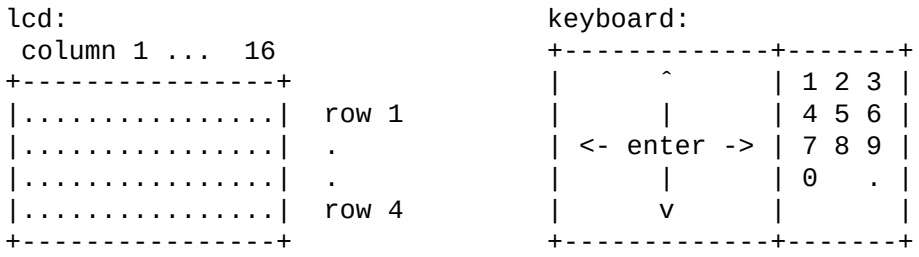

Figure 3.1: A 16x4 LCD and keyboard design

The executor does not know anything about the hardware. It doesn't know what display or keyboard controller is attached to the system, and it doesn't want to. *mexc* was designed to be general purpose as much as possible. It is entirely written in C and needs to be linked with a predefined set of functions to make it display things and react to key presses. But more on this later.

First we will explain how *mexc* actually displays a menu and how it navigates the user through it. After defining how to enter passwords and edit input fields, thereby introducing each, we will look at the library from a programmer's point of view.

3.3 Display layout

mexc uses a fixed layout on the screen. The first line is called *header line* and displays various information about the current state of the user interface. The rest of the lines is used to display the data of the currently opened menu table.

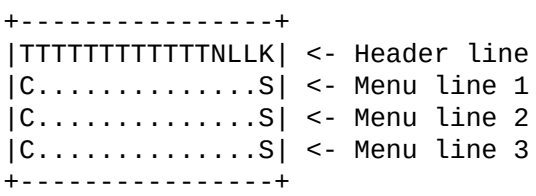

Figure 3.2: Display layout [8]

Figure 3.2 shows how a four line display is used. Comparing the display layouts, there have been no changes from Mr. Högl's original MEX. As indicated by the letters, the screen is divided into various fields:

T This field displays the title of the currently opened menu table. It has no specific length, but it gets as much columns assigned as possible. That is the width of the LCD minus 4 (for the

'NLLK' fields). The title is displayed left-aligned and cut if it's too long to fit into the field completely.

N This field holds the so-called *navigation character* and tells the user whether the current menu line is editable or not. Either a colon (':'), when there is at least one editable component, or an asterisk ('*'), in the case of a *read-only* menu line, is displayed.

L The L-field shows the line number of the current menu line.

K The K-field displays either a blank (' '), or a plus sign ('+'), or an exclamation mark ('!').

- The blank is displayed when all available menu lines of the currently opened menu table are visible on the screen. On a four line display this would be the case in a menu table with only 3 or less lines.
- A plus sign says that there are more menu lines than those currently displayed on the screen. They can be made visible by scrolling the menu table up or down.
- An exclamation mark indicates that there are more menu lines than those on the screen, but that the cursor is on the last possible line in the current menu table. To see the other menu lines the user needs to scroll upwards.

C By default, the first column of each menu line is reserved for the cursor. If it's not empty, it indicates that the appropriate menu line is currently active. *mexc* can be configured to use this column for menu data and draw the current menu line with inverted colors, thus the C-field is actually optional.

S The last column of each menu line is also reserved. It displays the submenu indicator. Typically, if there is a submenu it will show a "greater than" sign ('>'). However, there can be a 'P' instead, which has to be interpreted as "there is a password protected submenu".

... The rest, marked with dots in figure 3.2, is used for displaying menu data. Because of the reservation of the first and last column, the menu data has fewer space available than a display may offer.

In the example[1] shown in figure 3.3, the title of the currently opened menu table reads 'Preferences'. Following the title there is an asterisk which indicates that the current menu line is not editable. The current line is marked with the tilde character ('~') in the first column. It's the 5th line in the menu table as noted by the number following the asterisk. On the right of the 5 there is an exclamation mark telling us that no more menu lines follow the current line. This means that the user will have to scroll upwards to see the other menu lines.

The '>'s and 'P's at the end of the menu lines show that for each there is a submenu. The submenu from the last line is protected by a password. The user will have to enter it before *mexc* will grant access to the submenu.

[1] The original example which was introduced by Mr. Högl originates from the documentation of MEX[8], it was slightly modified here.

```
+----------------+
|Preferences * 5!|
| Sensors...    >|
| Parameter...  >|
|~System...     P|
+----------------+
```

Figure 3.3: A display example

3.4 Surfing around

Using the four direction keys of the keyboard, it is possible to move freely within the menu hierarchy. By pressing UP or DOWN the cursor can be moved to the previous or next menu line respectively. If the current menu line has a submenu indicator, pressing RIGHT will cause *mexc* to step into the submenu. *mexc* will of course ask for the password, if any, and then open the menu table only if the input was correct. The LEFT key brings the user back out of a submenu to where he was before.

With a numerical keyboard *mexc* can provide a very fast way of moving through a menu. Pressing a NUMBER key, while not editing a component, causes *mexc* to jump directly to the *nth* menu line of the currently opened menu table; in this context zero has a value of 10. If the target menu line has a submenu, *mexc* will immediately enter it, optionally a password may be requested. If there is no submenu but editable components in the target line, *mexc* will prepare everything to let the user change the first editable component. Pressing the POINT key, while not editing a component, will always drop the user to the first line of the top-level menu table. Because of the limitation of 10 lines to be directly addressed, the menu programmer should put frequently accessed menu lines at the beginning of the menu table.

It should be noted that *mexc* can return to the top level menu table by itself if the user doesn't press any key for a specified number of seconds. This number is specified by the programmer of the menu and can be up to 255 seconds, which is a little bit more than 4 minutes.

3.5 Entering a password

Whenever the user is required to enter a password, *mexc* will open an input field in the header line of an appropriate length. For a five characters long password, '`Pwd:.....`' will be displayed in the header line, the rest of it cleared, and the cursor, a blinking black box, placed on the first dot. On any subsequent key press, a dot ('`.`') is changed into an asterisk ('`*`') and the cursor shifted by one to the right, thus indicating how many password characters have already been entered. For passwords being longer than the width of the display the user can still enter the password, but the cursor will stay at the right edge if it's already there.

```
   3
1  0  2
   4
```

Figure 3.4: Numeric values on direction keys

While the cursor is in the password input field, the direction keys plus ENTER of the attached keyboard get the following meaning. The LEFT key is mapped to a 1, the RIGHT key to a 2, the UP key to a 3, the DOWN key to a 4, and the ENTER key to a 0 as shown in figure 3.4. Additionally, the NUMBER keys from a numerical keyboard can be interpreted, too. The POINT key gets not interpreted, and pressing it will insert the 'invalid character' into the password buffer. A programmer of a menu should consider that passwords should consist only of digits. Otherwise *mexc* will never successfully validate it.

When all characters have been correctly entered, then immediately after pressing the last character the actual action happens. In the other case, when the user entered a bad password, the header line will flash for a second and the attempt to perform an action is aborted.

3.6 Editing menu lines

Every menu line is built up of so-called *line components*. Some of these components can be edited and provide input elements to a program. In this section we will look at the navigation within a single menu line and at each component. We will see how the various components are displayed, what values they can hold, and how they are edited.

3.6.1 Navigating within a line

The navigation within a menu line happens via the ENTER key. When the current menu line doesn't contain editable components, the N-field in the header line is set to '`*`', then pressing the ENTER key has no effect. On the other hand, when there are editable components, pressing ENTER causes the so-called *horizontal cursor*, a blinking blackbox, to appear and jump to the first editable field. Pressing ENTER again will move the cursor to the next editable component and so on, until the horizotal cursor disappears.

While the cursor is visible, *mexc* is in the so-called *edit state*. In this case, the other keys beside ENTER are specially interpreted and cannot be used to navigate through the menu. How these keys are interpreted depends on the component being edited.

3.6.2 Number fields

A number field can contain a float, a signed, or an unsigned integer number. This number is displayed right aligned within a fixed width on a screen. A specification of the available number types currently supported is given in the following list.

`|dd-int.........42|` This is an unsigned one-byte integer with a value range of 0 .. 99. There is no sign in front of the number. The component consumes exactly two characters on the screen even for numbers with only one digit.

`|ddd-int........123|` This is an unsigned one-byte integer with a value range of 0 .. 255. There is no sign in front of the number. The component consumes exactly three characters on the screen even for numbers with only one or two digits.

`|hh-int..........E6|` This is an unsigned one-byte integer with a value range of 0 .. 255 (0xFF) displayed in hexadecimal format. There is no sign in front of the number. The component consumes two characters on the screen. For values less than 15 a zero is put in front of it.

`|sdd-int........-42|` This is a signed one-byte integer with a value range of -99 .. +99. The sign is always displayed, however, this can be configured at compilation time. The component consumes three characters of a menu line.

`|sddd-int......-123|` This is a signed one-byte integer with a value range of -128 .. +127. The sign is always displayed, however, this can be configured at compilation time. The component consumes four characters of a menu line.

`|DDD-int........567|` This is an unsigned two-byte integer with a value range of 0 .. 999. The component consumes three characters of a menu line. The sign is not shown.

`|DDDD-int......5243|` This is an unsigned two-byte integer with a value range of 0 .. 9999. The component consumes four characters of a menu line. The sign is not displayed.

`|DDDDD-int....12345|` This is an unsigned two-byte integer with a value range of 0 .. 65535. The component consumes five characters of a menu line. The sign is not displayed.

`|HHHH-int......FDE9|` This is an unsigned two-byte integer with a value range of 0 .. 65535 (0xFFFF). The value is displayed in hexadecimal format and consumes four characters of a menu line with optional leading zeros. The sign is not shown.

`|SDDD-int......-576|` This is a signed two-byte integer with a value range of -999 .. +999. The sign is always displayed, however, this can be configured at compilation time. The component consumes four characters of a menu line.

`|SDDDD-int....-5243|` This is a signed two-byte integer with a value range of -9999 .. +9999. The sign is always displayed, however, this can be configured at compilation time. The component consumes five characters of a menu line.

`|siif-float...+12.4|` This is a IEEE 754 float with a value range of -99.9 .. +99.9. Note that there is always only one fraction digit displayed. The sign is always displayed, however, this can be configured at compilation time. The component consumes five characters of a menu line. The float value to be displayed is rounded to one fraction digit accuracy.

`|siiif-float.+123.4|` This is a IEEE 754 float with a value range of -999.9 .. +999.9. Note that there is always only one fraction digit shown. The sign is always visible, but this is configurable at compilation time. The component consumes six characters of a menu line. The float value to be displayed is rounded to one fraction digit accuracy.

Editing one of the above specified numbers can be done with all four direction keys. Initially, the horizontal cursor is placed at the first character of the number. This can be a digit or the sign. By pressing LEFT or RIGHT the horizontal cursor is shifted in the appropriate direction to the previous or next character of the number. Editing a float, the cursor jumps over the decimal point. When the cursor is on a sign, pressing UP or DOWN toggles the sign either to '+' or to '-'. When the cursor is on a digit, pressing UP or DOWN will in- or decrease the digit's value. Thereby, pressing UP on a '9' will change it to '0', and pressing DOWN on a '0' will change it to '9'.

Optionally, numbers can be edited using the NUMBER keys. Pressing such a key will input the appropriate digit at the current cursor position and move the cursor to the next character. While editing float values, pressing the POINT key will cause the cursor immediately jump behind the decimal point and set the digits in front of it to zero if the cursor actually was before the decimal point.

Pressing the ENTER key causes the cursor to leave the component.

3.6.3 Counter fields

A counter field displays either a float or a signed two-byte integer. The number is rendered right-aligned within a fixed width on the screen. The value range of a counter is limited by the menu programmer who defines the range. What counters makes really different from normal numbers is the way they are edited.

While editing, the horizontal cursor is placed and kept on the last character of the displayed number. The value of the counter is increased with the UP key and decreased with the DOWN key. By which value the counter is altered is defined in the provided menu byte code. Incrementing or decrementing outside the specified range is not possible by the user. Pressing LEFT, RIGHT, the NUMBER keys or the POINT has no effect.

3.6.4 Time fields

A time component can occur in two different formats. The short format has no 'seconds'. Editing time components is done in two steps for the short format and in three steps for the long format. Each part of a time component is edited like an integer counter with special ranges. Initially the 'hours' are edited using the UP and DOWN keys. It can be in- or decreased in the range of 0 .. 23. Pressing ENTER will move the horizontal cursor to the 'minutes' . Pressing there UP or DOWN will in- or decrease the minutes in the value range of 0 .. 59. For a long time component pressing ENTER again will move the cursor to the 'seconds' which are equally edited as the 'minutes'. For a short time component pressing enter while editing the 'minutes' will stop editing the component.

```
|long:.....19:42:03|
|short:.......19:42|
```

As can be seen in the example, time is displayed in '24H' format. Numbers smaller than 10 are prefixed with a zero, so each part has exactly two digits. *mexc* puts colon ':' between each part of

the time. Thus, a short time consumes 5 characters, while a long time needs 8 characters.

3.6.5 Date fields

The date field is very similar to the time component. It consists of three parts: year, month, and day. Each part is an integer counter and each is edited on its own. Initially, the horizontal cursor is placed on the year part which can be increased by one with the UP key and decreased with the DOWN key. The value range for the year begins with 0 and ends with 9999 for the long and 99 for the short date type. After pressing ENTER, the cursor jumps to the month part which can also be in- and decreased by one. A month counts from 1 up to 12. Pressing ENTER again moves the cursor to the day part which is edited the same way as the other parts. The value range for a day starts with 1 and ends with 31. Pressing LEFT, RIGHT, the NUMBER keys or POINT while editing a date field has no effect.

```
|long:.....2005-10-17|
|short:......05-10-17|
```

A date is displayed in the 'YYYY-MM-DD' format, optionally with the year part being shortened to two digits. Thus, this line component takes 10 or 8 characters of a line depending on the format. *mexc* uses a dash ('-') to separate the parts of a date. It should be noted that *mexc* itself does not check for invalid dates, e.g. 2005-02-31. It is on the programmer that uses this library to do so.

3.6.6 Switch fields

A switch displays a bit field of length *n* and consumes an equal number of characters in a line. It represents *n* bits which can be toggled upon editing. The horizontal cursor is initially placed on the first bit. Using the UP and DOWN keys, the bit under the cursor can be toggled *on* or *off*. Using the LEFT and RIGHT keys, the cursor can be positioned at the previous and next bit respectively. Pressing one of the NUMBER keys or the POINT has no effect, except forcing the help string to be displayed. Here are two examples for switch fields:

```
|PORT-1:   **.*.***|
|Mask: 101011101101|
```

The characters representing the *on* or *off* state can differ from switch to switch. They are not hardcoded in *mexc*, but are specified by the programmer of the menu and stored in the byte code separately for each switch.

There is something special about the switch field. Whenever a key press occurs, *mexc* will display a help string, which is associated with the bit under the cursor, in the header line. This help string will disappear after a certain number of seconds. This number is defined by the programmer of the menu.

3.6.7 Option fields

An option field displays a string from a string list which is defined in the byte code. The string is rendered right-aligned in a width which is defined by the longest string in the list. Pressing the UP or DOWN key will cause the component to browse through the string list and display the previous or next string. Thereby, the list is treated like a ring. When the first string is displayed, the previous one is the last in the list. And if the last string is currently displayed, the next one becomes the list's first. Pressing LEFT, RIGHT, the NUMBER keys or POINT has no effect.

3.6.8 Strings

Beside displaying constant strings, *mexc* also provides a way to let the user edit a string. It should be noted that the length of a string cannot be altered. All the user may change are the characters within the string.

Having only the small 5 key keyboard, changing a character is done the following way. Pressing UP or DOWN changes the character at the current cursor position. With LEFT and RIGHT the cursor can be shifted by one character into the appropriate direction. Pressing ENTER will leave the component and stop editing the string.

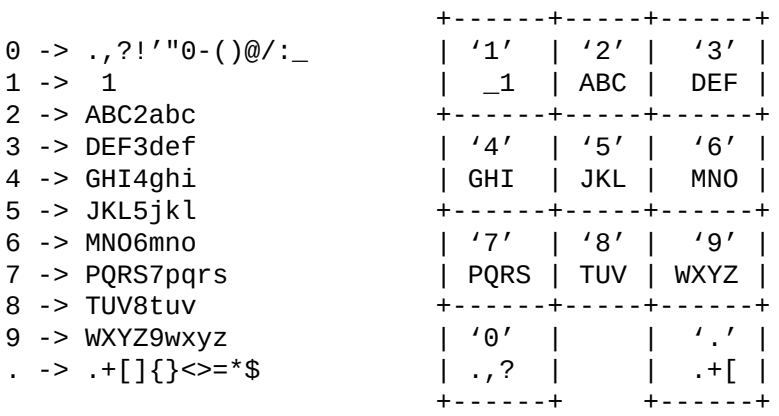

Figure 3.5: Input string lists and suggested layout for the numeric keyboard

Having the additional numeric keyboard available, editing a string is much more comfortable. *mexc* tries to imitate the way strings are entered on mobile phones. Each key on the numeric keyboard has an associated list of characters which can be browsed through by repeatedly pressing the same key in a small period of time. At the current cursor position, the string is assigned the currently selected character. Waiting for a short while or pressing another key will make the cursor jump to the next position. This way a skilled user can insert a new text in a fast way with only one finger. To help the user learning the keyboard layout, *mexc* will display the current character list in the header line. While repeatedly pressing a key, and thus choosing the next character, the displayed help string in the header line will rotate, so the currently selected character is always at

the beginning.

Figure 3.5 shows what character lists are assigned to each key from the numerical keyboard. The grid sketch on the right is a suggestion for a layout of the numerical keyboard with possible labels.

3.6.9 Triggers

A trigger is nothing more than a "soft-button" on the screen. To activate the button the user needs to press any key except ENTER. Pressing ENTER will leave this component and will not activate the soft-button. Triggers occur in two manners, normal and password protected. When entering the edit state, the horizontal cursor is placed at the '`X`' or '`P`'.

```
|Reset System Values: [X]|  <-- normal trigger
|Reboot Whole System: [P]|  <-- password protected trigger
```

As shown in the example, password protected triggers are displayed as a '`[P]`'. Activating them requires the user to enter a correct password before the action is carried out. How to enter a password is described in section 3.5.

3.7 Programming with *mexc*

This part of the documentation is directed to programmers who want to use *mexc* for their own applications. Firstly, we will take a look at what has to be done to get the library working properly. Then we will explain the compilation and configuration details. After introducing the public API, we will finally have a look at how a simple program that uses *mexc* can be written.

3.7.1 What *mexc* needs

When using *mexc*, the first thing to understand is what the library needs to be provided. As it has been concepted to work with many kinds of hardware, the developer implementing a program for a specific system has to write a set of hardware dependent routines which *mexc* will use. Providing the library with such routines, the programmer has great power at hand to customize the look and feel of the final application. The functions can be split into four categories: display accessing, key press fetching, waiting and C string utilities.

mlcd_* *mexc* requires routines that start with the prefix "`mlcd_`" and provide access to the connected display. They provide functionality for writing a string at a specified position, clearing parts of the display, controlling visiblity and position of the cursor and telling *mexc*, how many lines and columns of the display it may use.

mfpkey_get This routine provides *mexc* with an inteface to the attached keyboard. The keyboard is completely unknown to the library and can be some virtual buttons on a touchscreen, for

example. All the details of the hardware are hidden to the library in the implementation of this function.

msleep In many situations, *mexc* needs to wait for a little time. To be sure this is done efficiently and accurately the library relies on the programmer to implement a sleeping routine possibly using hardware dependent features.

str* There are two string utility functions, namely `strcpy` and `strlen`, which *mexc* uses to deal with C style strings. The implementation of them is trivial and they are not hardware dependent at all, but there are optimizations that we should consider.

Of course, *mexc* needs to be fed with the menu byte code it should interpret, but we will look at this later. Now let's dive into the details of the required functions.

3.7.1.1 Display accessing

Having the source code of *mexc* at hand, a glance into the file called `mlcd.h` will reveal what display accessing routines *mexc* exactly needs. Let's look at each and define what effects they are expected to perform.

`void mlcd_cu_off (void)`
Calling this routine should immediatelly hide the horizontal (LCD) cursor – it is often a blinking blackbox when visible. Upon entering the main loop, *mexc* hides the cursor and shows it only when the user is about to edit a component.

`void mlcd_cu_on (void)`
This routine should make the horizontal cursor visible again. It is the counterpart to the previous function.

`void mlcd_cu_gotoxy (unsigned char x, unsigned char y)`
While the previous functions control the visibility of the cursor, this routine controls the cursor's position. The `x` and `y` parameters specify the column and the line the cursor should be positioned at.

`void mlcd_clrchr (unsigned char n)`
This routine is expected to clear *n* columns to the right starting at the current cursor position. Actually, *mexc* could implement this by writing *n* space characters, however, a hardware dependent implementation of this routine can be much more efficient. This routine may or may not change the current cursor position, *mexc* doesn't require a specific behaviour.

`void mlcd_clrln (unsigned char n)`
This routine should clear a whole line on the display. The line is indexed by *n* – zero is assumed to be the index of the first line. As with the previous function, also this one may or may not change the current cursor position.

`unsigned char get_mlcd_lines (void)`
: This routine is called upon *mexc*'s initialization and supposed to return the number of lines the library will use. An example given below will explain this in more detail.

`unsigned char get_mlcd_cols (void)`
: This is the counterpart to the previous routine and should return the number of columns *mexc* will use on the display. An example given below will explain this in more detail.

`void mlcd_wrstrxy (unsigned char x, unsigned char y, char * str)`
: *mexc* uses this function to print zero terminated strings on the display. `x` and `y` specify where on the display the string should be printed. The cursor is expected to be left behind the written string.

```
void mlcd_wrstrxymax (unsigned char x, unsigned char y,
                      char *str, unsigned char n)
```
: This function is similar to the previous one, but the passed string doesn't need to be zero terminated necessarily. It should print at most *n* characters but respect a zero terminator, if there is one. The function is expected to leave the cursor behind the written string.

```
void mlcd_wrchrxy (unsigned char x, unsigned char y,
                   unsigned char c)
```
: This function is used by *mexc* to put a single character on the display at the specified position. It is expected to leave the cursor behind the written character.

`void mlcd_invertln (unsigned char n)`
: The "invert-line" function is actually optional and, when available, used by *mexc* to highlight the currently selected menu line. All the routine is expected to do, is to redraw the specified line with inverted colors. When the same line is inverted twice, it should display normally. Inverting is meant to be temporarily only. Whenever *mexc* writes to an inverted line the new characters are expected to be displayed normally, not inverted.

  Being able to use this function, *mexc* won't reserve the first column for the current line indicator as described in section 3.3. Thus, there is one more column for the menu data. As mentioned, this routine is optional and the programmer must decide whether to implement it or not. Mainly, this decision depends on the display being used and its capabilities. The library needs to be compiled with the `CONFIG_ENABLE_INVERTLN` preprocessor definition to make it use the function.

mexc ensures that all printed strings fit within the rectangle defined by `get_mlcd_lines` and `get_mlcd_cols`. This rectangle can be smaller than the actual screen. Due to the fact that the cursor positioning happens completely via the `mlcd_*` functions, the application programmer has the possibility to position the display rectangle occupied by *mexc* anywhere on the screen.

Of course the library can occupy the whole screen, but let's discuss the following more complicated scenario. Let's say we have a 40x20 display, but we want *mexc* to use only a 20x5 area in the right corner at the bottom as shown in figure 3.6. To make *mexc* address the proper screen area all we need to do, is to provide a suitable implementation of the `mlcd_*` functions.

`get_mlcd_lines` and `get_mlcd_cols` will simply return the constants 5 and 20 respectively. The other routines, which position the cursor, need to add a constant offset to the passed coordinates before moving the cursor. In our scenario they will simply add 15 to each `y` argument and 20 to each `x` argument.

There is one more thing that should be pointed at. As mentioned, the output routines are assumed to leave the cursor behind the written character or string. When *mexc* writes a string which ends exactly on the last column of the screen area assigned to the library, there is the question "What to do with the cursor?". In this situation, *mexc* doesn't require any specific behaviour and it is on the implementation of the `mlcd_*` routines to put the cursor somewhere.

3.7.1.2 Keyboard interface

To get access to the keyboard *mexc* uses only one function. This routine is declared in the file `mfpkey.h` as follows.

```
unsigned char mfpkey_get (void);
```

The advantage of using this routine is that *mexc* itself has no idea about the keyboard hardware. It could be even a small joystick attached to the system. This routine is responsible to fetch a key press and supply *mexc* with a constant defined in the same header file as the function's declaration. The important thing to note here is, that `mfpkey_get` is called by the library whenever it needs a key press to interpret, but there is no way into the other direction. The code cannot send any key press event to the library. Fortunately, *mexc* calls this function quite often. So with a small keyboard buffer no key press should get lost.

`mfpkey_get` is expected not to block the library by waiting for a key press. If there is nothing to serve *mexc* with, it should immediatelly return with `MFPKEY_NONE`. The library will idle for a short while and try again. *mexc* may perform a specific action upon idling for a defined interval, therefore it is important not to block it.

The header file defines quite a few constants. These are understood by *mexc* and have to be returned by `mfpkey_get`. The first six defintions are required to let the user interact with the menu.

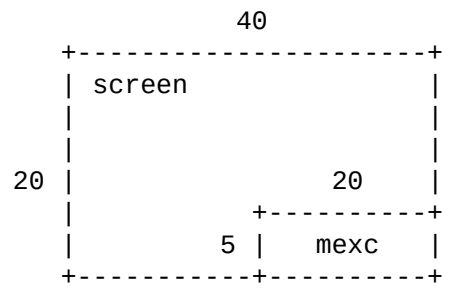

Figure 3.6: Display arrangement for an example application

Serving `MFPKEY_NUMPAD_*` definitions is optional. They allow the user to navigate through a menu more quickly and edit a component with more comfort, but they are not necessary. When the user isn't editing a component and the library gets the `MFPKEY_QUIT_MEXC_LOOP` definition, it will jump out of its main loop (`mexc_loop`) and return to the caller.

3.7.1.3 Sleeping

Beside the already discussed routines one more needs to be linked with the library. This function will allow *mexc* to put itself into sleep mode for a specified number of milliseconds.

```
void msleep (unsigned short msec);
```

A simple implementation would just loop until the time is over. On systems which already provide a task sleeping service the code could call such a system function. A clever but more complicated code could perform some background task while *mexc* is idling. It could for example look whether a key press occured and put it into a keyboard buffer queue.

3.7.1.4 String utilities

Further on, *mexc* uses the two routines `strcpy` and `strlen` which are not included in the library. These are often included in the development environment libraries or even included as built-ins by the compiler.

For example, `gcc`, when not passed the `--no-builtin` option, replaces calls to `strlen` on constant strings with the actual length of the strings at compilation time. This results in an optimazation of speed and code size. If this function is used only in conjunction with constant strings, this case is true for *mexc*, then there will be no call to `strlen` at runtime and the function's code needs not to be linked to the library.

However, if the development environment doesn't bring the two functions one needs to implement and link them to the library.

3.7.2 Memory requirements

Using the `size` program, we can examine the size of the *mexc* library. However, this size is greatly dependent on the utilized compiler, the optimization options of it and the use of preprocessor definitions being discussed in section 3.7.3. Using the GNU C compiler, the size of *mexc* for a 32-bit Intel platform should vary between 10KB[2] and 20KB[3] with `asserts` disabled.

[2]The exact compilation command to produce the result was:
`gcc -Os -fomit-frame-pointer -DHAVE_STRING_H -DCONFIG_DISABLE_FLOAT -c *.c`

[3]The exact compilation command to produce the result was:
`gcc -DHAVE_STRING_H -DCONFIG_NUMPAD_KEYBOARD -c *.c`

The other question is, how much stack *mexc* needs. The required stack size is dependent on the target architecture, the used compiler, and its optimization options. Further on, the required stack size also depends on the number of columns *mexc* will use on the screen and the depth of the menu structure itself, too. Using the GNU debugger, tests with *mexc* on a 32-bit Intel platform with a 20 columns display and a menu structure of depth 3 have shown, that the library needs around 550 bytes of stack.

It also needs to be considered, that the binary menu description needs some space, too. To quickly find out how much memory a concrete menu consumes, the `--binary` option of the *mlx* compiler can be used. It will produce the menu as a binary file which size can be easily determined by system services.

3.7.3 Compiling the interpreter

mexc is known to compile smoothly with GCC[4] and was successfully tested on Intel architectures as well as on a Motorola DragonBall VZ processor. Using the OpenWatcom C/C++ compiler tools[5], *mexc* was successfully run on the Intel 8088 processor. There are a couple of things that can be configured at compilation time and we will look at them in this section.

The provided `Makefile` can be used to build *mexc*. The following listing shows how the library can be built by hand.

```
~mexc% ls *.c
cmf.c  mexc.c  mutils.c  pstring.c
~mexc% gcc -c *.c
~mexc% ls *.o
cmf.o  mexc.o  mutils.o  pstring.o
~mexc% ar rcs libmexc.a *.o
~mexc% ls libmexc.a
libmexc.a
```

Listing 3.1: Creating the library

Of course, optimization options can be passed to `gcc` or the compiler of choice. For example, using the `gcc` options `-fomit-frame-pointer` and `-Os` at the same time, it is known to reduce the needed stack size of *mexc* by almost a half on modern desktop computers.

There are various preprocessor definitions that configure the library's behaviour. They don't have any substitution value and can be defined on the command line using `gcc`'s `-D` option. In the following list we will introduce each and also explain the effects.

`HAVE_ASSERT_H` The source code for *mexc* is studded with calls to `assert` to quickly find bugs during development. The function itself is declared in the system header file `assert.h`. However, some development environments don't provide such a header file, thus causing

[4]The official web page for the GCC project can be found at [http://gcc.gnu.org/].
[5]The official web presentation of Open Watcom can be found at [http://www.openwatcom.org/].

problems at compilation time. Therefore, when HAVE_ASSERT_H is not defined at compilation time, the header file is not included by the code and all calls to the `assert` function get removed on the fly. Actually, they get substituted with nothing by the preprocessor.

HAVE_STRING_H *mexc* uses two functions of the standard C library: `strcpy` and `strlen`. They are declared through the system header file `string.h`. On some systems this file may not be available, and therefore including this file is protected with the HAVE_STRING_H definition. Only if it is defined at compilation time the code will include the system header. However, not defining it doesn't prohibit *mexc* from using the two string functions. In this case, the programmer needs to provide and link them to *mexc*.

CONFIG_NUMPAD_KEYBOARD As already mentioned in the introduction, *mexc* is capable of interpreting an additional numerical keyboard. If such a keyboard is not available on the target system, the code responsible for handling those key presses can be disabled at compilation time. This results in a smaller size of the library.

CONFIG_ENABLE_MLCD_INVERLN See description of `mlcd_invertln` in section 3.7.1.1.

CONFIG_DISABLE_FLOAT Defining this flag at compilation time will disable all code dealing with the `float` data type. This flag comes from a test on an Intel 8088 processor. If the target platform doesn't support floating-point operations this flag should be defined. Of course it will reduce the size of the library at the cost of not being able to deal with `float` numbers.

To compile *mexc* with support for the numerical keyboard but not for floats the following command line would be used:

```
~mexc% gcc -c -DHAVE_STRING_H -DCONFIG_NUMPAD_KEYBOARD \
          -DCONFIG_DISABLE_FLOAT *.c
```

Listing 3.2: Compiling the library with customization options

While the definitions shown above control whether some features will be included or excluded from the code, the following definitions provide customization of features included in it. These are defined in the file `mconfig.h` and must not be removed, but can be changed to reflect the requirements.

MEXC_MAX_LINE_LEN is used only when *mexc* is *not* compiled with `gcc`. The GNU C compiler has a very nice feature called "Arrays of Variable Length" which enables the library to allocate such arrays on the stack. When this feature is not available, the code must assume a fixed length for its buffers. It is important that MEXC_MAX_LINE_LEN is equal to or greater than the value returned by `get_mlcd_cols`, otherwise buffer overflows will make the code run incorrectly, and possibly cause endless loops.

MEXC_MAX_PWD_LEN has the same background as the previous option and is used only when compiling *not* with `gcc`. It defines the length of the password buffer and must not be smaller than the longest password in the menu definition to avoid buffer overflows.

MEXC_MAX_DISP_CONTEXT_DEPTH's value is important not to be smaller than the maximum depth of the menu hierarchy. When entering a submenu, *mexc* stores the current display context in a table and increases the current context level. When returning from the submenu, the library restores the last display context and decreases the level. The value of MEXC_MAX_DISP_CONTEXT_DEPTH gives the number of possible entries in the context table. One entry stores three pointers, so on a 32-bit platform an entry will take up 12 bytes.

If this value is too small, *mexc* will display the error message "err: menu too deep" and refuse to enter the submenu when the user reaches the table space limit.

MEXC_LC_PASSWORD_STRING defines the string to be displayed as a password protected trigger line component. The length of the string should be non-even as *mexc* tries to position the horizontal cursor upon activating the component into the mid of the displayed label.

MEXC_LC_TRIGGER_STRING has the same meaning as the previous item with the exception that it is displayed for normal triggers – triggers that are not password protected.

With MEXC_FORCE_SIGN_ON_SIGNED_NUMBERS being defined as non-zero, *mexc* will always display a sign on signed numbers. Thus a posivite signed number is preceeded with a plus (+). This behaviour can be turned off by specifying a zero.

MEXC_FIRST_PRINTABLE_CHAR and MEXC_LAST_PRINTABLE_CHAR define an interval in the character code table. Characters in this interval, including both ends, can be entered when editing a string with the UP and DOWN keys.

MEXC_BLINK_INTERVAL defines the number of seconds after which a blinkable component should be erased on the screen and after the same interval redrawn again, thus, letting the component visually blink.

MEXC_ASK_PWD_PROMPT is the string that *mexc* will display as a prompt in front of the password input field. See section 3.5.

MEXC_FLASH_DELAY is the number in milliseconds a flash will last. Sometimes the user presses a key which is not suitable in the current situation. *mexc* warns the user about it with a short flash in the header line.

MEXC_GET_KEY_DELAY; As described in section 3.7.1.2, *mexc* calls mfpkey_get to fetch a key press. However, when there is nothing, MFPKEY_NONE gets returned and *mexc* will sleep for MEXC_GET_KEY_DELAY milliseconds before trying to fetch again. The smaller the value of the definition the quicker *mexc* will response to key presses. The current implementation of *mexc* restricts the value not to be smaller than 4.

MEXC_SHOW_ERRMSG_DELAY is another interval in milliseconds. It defines how long an error message will be displayed.

MEXC_BLACK_BOX_CHAR should be the character code of a black box character. It is used to produce the flash in the header line. If no such character is available any other can be used, too.

`MEXC_CUR_LINE_INDICATOR_CHAR` is used only if `CONFIG_ENABLE_MLCD_INVERLN` is *not* defined. It defines the cursor character displayed in the first column of a menu line as described in section 3.3.

`MEXC_SUBMENU_INDICATOR_CHAR` defines the character to be displayed at the right border of a menu line if there is a submenu.

`MEXC_SUBMENU_PWD_INDICATOR_CHAR` defines the character to be displayed at the right border of a menu line if there is a password protected submenu.

`MEXC_LAST_LINE_INDICATOR_CHAR` defines the character to be displayed in the K-field as described in section 3.3 when the cursor is on the last line of a menu table.

`MEXC_MORE_LINES_INDICATOR_CHAR` defines the character to be displayed in the K-field as described in section 3.3 when there are more lines in the menu table than visible in the screen area.

`MEXC_MLINE_EDITABLE_CHAR` defines the character to be displayed in the N-field when the current menu line contains editable components.

`MEXC_MLINE_READ_ONLY_CHAR` defines the caracter to be displayed in the N-field when the current menu line has no editable component.

`MEXC_TIME_SEPARATOR_CHAR` is the character to be put between the hours, minutes, and the seconds of a time component.

`MEXC_DATE_SEPARATOR_CHAR` is the character to be put between the year, month, and the day of a date component.

`MEXC_FILLER_CHAR` is the character to be put where something is missing. It should be the blank character to interact smoothly with the `mlcd_clrchr` function.

`MEXC_PASSWORD_CHAR` defines the character that should be echoed when entering a password.

Creating different applications for different platforms will probably require to configure the library for each platform and application. Because of this and the fact that there is quite a lot to be configured there is no complete building and installation system.

3.7.4 Writing a program

From a programmer's point of view, using *mexc* is quite simple. However, there are several steps that need to be done.

- First of all the *mexc* interpreter needs to be compiled and possibly customized. This is discussed in section 3.7.3.

- As *mexc* is expected to be linked with a predefined set of functions, the second step is to create these routines.
- Next, the binary menu image which *mexc* will interpret is needed. For this step the *mlx* compiler has been written.
- Finally, everything is prepared to write a complete program.

We will now give an overview of the *mexc* API and look at an example a little bit later.

3.7.4.1 Data types

The API is exported through the header file `mexc.h` which includes the declaration of four public functions. The used data types are defined in `mtypes.h`. Here are the appropriate excerpts:

```
28 typedef unsigned char   uchar;
   typedef char            schar;
30 typedef unsigned short  uint2;
   typedef short           sint2;
32 typedef unsigned char * addr_t;
```

Listing 3.3: mtypes.h / 28–32

```
37 typedef void  (*fcncbp) (uchar, addr_t);
```

Listing 3.4: mtypes.h / 37

While all data types are just synonyms for those already existing in the C languages, `fcncbp` needs a short explanation. It is a pointer to a function with two parameters and no return value. The first argument of the function has to be of type `unsigned char` and the second a pointer to an `unsigned char`.

3.7.4.2 mexc_init

```
uchar mexc_init (addr_t mcode, addr_t ram, fcncbp def_cb_handler);
```

Initialization of the library happens with a call to `mexc_init`. Beside initializing the library's globals, it will verify the passed menu byte code and intialize all menu variables in RAM. With an exception to the `get_mlcd_*` functions, none of the display accessing routines gets called at this moment.

`mexc_init` will return zero to indicate that everything went alright. Otherwise, it will return one of the following constants which are defined in `cmf.h`.

`CMF_INIT_BAD_ID` indicates that the byte code to interpret isn't in CMF format.

`CMF_INIT_UNSUPPORTED_VERSION` indicates that the byte code version isn't supported by the library. `CMF_MAJOR_VERSION` and `CMF_MINOR_VERSION` defined in `cmf.h` show the supported version.

`CMF_INIT_BAD_BYTE_ORDER` indicates that *mexc* and the byte code don't match the same endianness. Often, this error comes from specifying the wrong argument to the `--endian` option of the *mlx* compiler or not using the option at all.

The three expected arguments to `mexc_init` have the following meaning:

`mcode` must be a pointer to the menu binary image. This parameter must not be `NULL`.

`ram` must be the address of a writable memory area. This parameter may be `NULL` if there are only constant strings in the whole menu.

`default_cb_handler` After a line component has been edited by the user, *mexc* will notify the application by calling a handler function. By default, it will invoke the function passed as the third argument to `mexc_init`. This parameter may be `NULL`.

3.7.4.3 mexc_loop

```
void mexc_loop (void);
```

A call to this function will start the main loop. It will display the top-level menu table, wait for key presses, and interpret them. It is necessary that `mexc_init` has already been called before. `mexc_loop` will not return as long as it hasn't fetched the `MFPKEY_QUIT_MEXC_LOOP` key press.

3.7.4.4 mexc_set_callback_handler

```
fncbp mexc_set_callback_handler (fncbp * fo, fncbp fn);
```

As already mentioned in section 3.7.4.2, the third parameter to `mexc_init` is the address of a function to be called whenever any component has been edited. However, callback handlers for individual components can be installed by using `mexc_set_callback_handler`. Its parameters are:

`fo`; the addresses of the memory block holding the address of the handler to be called. Usually one will pass a `CALL_*` definition for the appropriate component from the header file outputted by the *mlx* compiler.

`fn`; the address of the function to call when the appropriate component has been edited.

Usually, calls to `mexc_set_callback_handler` occur after initializing *mexc* and before running its main loop. The installed handler is called with two arguments, the first being a numerical value representing the type of the edited component, and the second being a pointer to the component's current value. Definitions for each type that *mexc* understands can be found in the file `cmf.h`.

The returned value is the address of the previously installed callback handler or `NULL` if there was none before.

3.7.4.5 mexc_enable_line

```
void mexc_enable_mline (unsigned char *addr, uchar val);
```

This routine provides a convenient way of enabling and disabling dynamic menu lines. *mexc* simply hides disabled menu lines. Currently, calling this routine will cause the interpreter to return to the top-level menu table and hide the appropriate line if the user is not editing a component.

`addr` specifies the address of the boolean 'enable' value declared by *mlx* in the generated menu header file. It is associated with a concrete menu line.

`val` is the new state of the menu line and will be stored where the first argument points to. Any other value than zero will enable a menu line.

mexc assumes there is always at least one visible line in a displayed menu table. For example, entering a submenu with all menu lines disabled will crash the interpreter!

3.7.4.6 mexc_redraw

```
void mexc_redraw (void);
```

Having the `mexc_loop` started and the user currently not being editing a component, invoking this function will simply redraw the screen area occupied by the library.

3.7.4.7 An example

Figure 3.5 provides a skeleton for an application using *mexc*. At first, `mexc.h` must be included. It makes the public API available. Including `menu.h`, the generated menu header file, imports the declarations of `g_mlx_menu` and `__MLX_RAM_BASE__` which are used upon initialization of *mexc*. If the initialization fails the program simply aborts. Otherwise, it starts the main loop which will display the top-level menu table and react upon key presses.

```
#include <mexc.h>
#include "menu.h"

int main ()
{
  if (mexc_init (g_mlx_menu, __MLX_RAM_BASE__, NULL))
    return 1; /* error occured */

  /* ... mexc callback installation */

  mexc_loop ();

  return 0;
}
```

Listing 3.5: Skeleton of an application

Following the initialization of *mexc*, there is room to install custom functions which are to be called after components were edited. Let's assume the following code snippet being in `menu.h`.

```
#define dd_integer ((unsigned char *)(__MLX_RAM_BASE__ + 0x00))
#define CALL_dd_integer ((fcncbp *)(__MLX_RAM_BASE__ + 0x04))
```

With the following statement between `mexc_init` and `mexc_loop` a custom function, here named `on_edited_cb`, would be called after the user edited the `dd_integer` component.

```
mexc_set_callback_handler (CALL_dd_integer, on_edited_cb);
```

The custom callback needs to be of type `fcncbp` as explained in section 3.7.4.1. In our example, the first argument can be ignored as we connect exactly one component to the callback. The second argument is a pointer to the current value of the component and optionally needs to be casted to the proper data type pointer. Here is a demonstration:

```
void on_edited_cb (unsigned char type, unsigned char * value)
{
  assert (value == dd_integer); /* from menu.h */
  assert (type == 0); /* or LC_TYPE_UCHAR_DD from cmf.h */
  printf ("current value changed to %d\n", *value);
}
```

Listing 3.6: Accessing menu variables in callbacks

3.8 Simulator

During the development of *mexc*, a simulator was needed to test and debug the code. `gsim` is a GTK+-2.0 based program which implements the required `mcld_*`, `msleep`, and `mfpkey_get`

routines. The program runs on MS Windows, various GNU/Linux distributions and FreeBSD; other operating systems have not been tested yet. It has proven that the simulator is very useful when writing menu definitions. One can immediately see, on the development platform, what the menu will look like.

3.8.1 Compiling it

To compile the simulator, the provided `Makefile` or `Makefile.win32` should be used. Of course, the appropriate makefile should be checked for valid paths and the `CFLAGS` makefile variable.

```
~gsim% make
usage: make [gsim|ggsim|menu|clean]
~gsim% make gsim
[...]
~gsim% ls -F gsim
gsim*
```

Listing 3.7: Creating the simulator

There are four targets, two of them – `gsim` and `ggsim` – are actually simulators. `gsim` is the character based LCD simulator, while `ggsim` is graphics based. The latter one is an experiment to show *mexc* with the `CONFIG_ENABLE_MLCD_INVERTLN` configuration.

Under MS Windows the simulator and *mexc* are known to compile smoothly with tools available by the MinGW project. The GTK+ library 2.0 or higher is required. When using `Makefile` also the `pkg-config` program will be needed.

3.8.2 Using it

To start the simulator, a filename containing the binary menu image must be specified on the command line. Invoking `gsim` without this parameter will make it return with an error.

```
~gsim% ./gsim
Usage: ./gsim [-z zoom | -c columns | -l lines | -i | -k] <menu>
```

Listing 3.8: Command line arguments of the simulator

The binary menu image to be passed to `gsim` needs to be in a binary file. The *mlx* compiler can generate such a file when passed the `--binary` option. Let's examine the other parameters.

`-z` awaits a numeric argument, the zoom factor, and causes the simulated LCD to be displayed as many times larger as specified. Default: 1.

`-c` awaits a numeric argument and sets the number of characters to fit into one line. Default: 20.

`-l` awaits a numeric argument and sets the number of lines on the LCD. Default: 4.

`-i` will popup the 'inspector' window which disassembles the binary menu image and represents it in tree view. It allows to change the current value of line components. Editable cells have a red background.

`-k` will popup a virtual keyboard.

The simulated LCD itself is a small green window with '`-c`' columns and '`-l`' lines. To navigate through the displayed menu the virtual keyboard window or the directions keys can be used. `MFPKEY_RTAB` is mapped on ENTER of the real keyboard.

Chapter 4

MLX Documentation

4.1 License

mlx is Free Software provided under the terms of the GNU General Public License (GPL) [9]. The file `COPYING`, that is distributed with *mlx*, contains a copy the GNU GPL.

4.2 Background

The idea of *mlx* has began with the M-Language by Hubert Högl. In his introduction to the M-Language he writes:

> "The M-Language is a notation for describing menu-directed user interfaces which are to be displayed on small LCD-panels (16x4 or 20x4). ML independently describes a) menu hierarchy and b) the contents of each menu line. The menu line contents can be constructed by using many different building blocks, e.g. strings, integer i/o-fields, switches, options, soft functions keys flexible horizontal whitespace and many more. ML ML was designed to fit the needs for embedded system design. A compiler for ML written in Python generates a block of tagged binary data which is self-containing and can easily be included into ones own embedded application."[10]

With *melx* the M-Language's intention doesn't change in any way. In fact, *melx* and its compiler, *mlx*, is just a rewrite with some improvements of Mr. Högl's work. The main change is the format of the language. While the M-Language looks a little bit like a Lisp program, a *melx* file is an XML document which must validate against the `melx.dtd` – this DTD is listed later in section 4.10. Looking closely at both languages, we will notice that there is little difference between them in structure.

4.3 Bird's-eye view

Let's look at what steps it takes to use *mlx*.

1. Firstly, we will create a description of a menu interface which is intended to be displayed on some small LCD-panels connected to an embedded system. To define the menu hierarchy we can use our favorite editor.
2. Having the menu description, we will use *mlx* to compile it into a compact binary format. By default *mlx* produces two files `a.h` and `a.c`.
3. Now the generated C file can be compiled and linked to an interpreter and/or a menu displaying program.

4.4 Requirements

mlx requires a standard Python distribution of version 2.3 or higher. If we'd like *mlx* to validate the source files, the `xmlproc` modules, which are part of the `python-xml`[1] package, are needed.

4.5 Introduction to *melx*

melx is the language which the *mlx* compiler understands, it describes a menu in its full hierarchy. Simply spoken, the language is a predefined set of elements and attributes organized in XML format. The exact structure of a valid *mexc* file is defined by the Data Type Definition (DTD) listed in section 4.10.

Let's step through a minimal menu definition to show what the basic elements are and what a *melx* file looks like.

```
1  <?xml version='1.0' encoding='US-ASCII'?>
2  <!DOCTYPE melx SYSTEM 'melx.dtd'>
```

As in each XML document, the first line must always be given. `encoding` can of course have another value. But both *mlx* and *mexc* currently do not support multibyte characters. The `doctype` declaration in the second line is optional, but important for the validation which is strongly recommended to help finding errors. The system identifier, that comes after the keyword `SYSTEM`, must be the path of a file holding the DTD definition. If this path is relative it is dissolved of the directory the *melx* document is located at. Thus, in this example, `melx.dtd` and the *melx* document must be in the same directory.

As stated by the `doctype` declaration, the top level element has to be `melx`. The DTD says that a `melx` element must have one child element called `description`, at least one child element with the name `menu`, and zero or more children called `line-format` in this order.

[1] The home page of this project is at [http://pyxml.sourceforge.net/].

```
3   <melx>
4     <description>
5       <delay-to-top value='120'/>
6       <delay-password value='10'/>
7       <delay-help value='2'/>
8       <top-menu ref='m-top'/>
9     </description>
10    <menu id='m-top' title='First Menu'>
11      <line ref='l-line-01'/>
12    </menu>
13    <line-format id='l-line-01'>
14      <string value='This is a string.'/>
15    </line-format>
16  </melx>
```

The above shown code satisfies the requirements. First there comes the `description`, then all the `menu`s, and finally all the `line-format`s. Let's look at each component in more detail and explain their meanings.

Note: If not otherwise mentioned the following restrictions apply.

- Most elements are empty. This means that they don't contain character data or other elements, and thus, can be shortly written as <*element* ... /> instead of <*element* ...></*element*>.
- Strings and passwords cannot be longer than 255 characters. Multibyte characters are currently not supported.
- Values of `id` attributes must be unique within the document. No other `id` attribute can have an already used identifier.
- All values which are expected to be numbers must be in decimal or in hexadecimal notation. Numbers in hex format must begin with "`0x`".
- The characters '&', '<' and '>' cannot be entered directly, but must be coded with their predefined entities which are '&', '<', and '>'. Further, *mlx* does not parse strings for '\x' sequences like the C compiler. Nevertheless, a similar construct already exists in XML and is called "notation for character reference" (e.g. 'ÿ').

4.5.1 Description element

The `description` element consists of four child elements which must be defined in the order shown in the example above. The following list explains their meanings.

delay-to-top ... must be empty. It must have exactly one attribute with the name `value` that holds numbers within the range of 0 to 255. Any other content for this attribute is considered as an error. The intention of this information is to give the number of seconds after which

the menu displaying program should return to the top-level menu table if the system idles. A zero defines the infinity, in other words, the program should never return to the top-level menu table automatically.

delay-password ... must be empty and have exactly one attribute named `value`. That attribute's content must be a number in the range of 0 to 255. It declares the number of seconds to wait, while the system idles, before aborting a password request. In other words, when the user is about to enter a password, but waits for more than *x* seconds, the interpreter should abort the input.

delay-help ... has the same format as both previous elements. Its meaning is to define a number of seconds after which a displayed help string has to disappear.

top-menu ... is also an empty element with one required attribute named `ref` which references a menu table to be displayed at the top level. The value of `ref` must be the same as an `id` value of a `menu` element.

4.5.2 Menu element

`menu` elements define collections of lines which are to be displayed as a menu table. Here is a list of valid attributes of a `menu` element.

id must be defined for each menu element and is a unique identifier so a menu table can be referenced.

title is an optional attribute and defaults to the empty string. It is the title of the menu table which the interpreter may display somewhere on the screen.

password can be given to protect the menu table. A user should be allowed to view this menu table only if he knows the correct password.

Each `menu` element must have at least one child. There are two elements which can be arbitrary mixed.

line This element simply references the definition of an advanced menu line. The `ref` attribute must therefore be identical with the `id` of a `line-format` element. The optional `submenu` attribute can reference a menu table to be entered when the user tells the interpreter to do so.

Dynamic menu lines can be implemented through the optional attribute `enable-vname`. It takes a valid C identifier which will be available in the generated header file and is a synonym for the address of an allocated byte in RAM. This byte denotes whether the menu line is currently enabled or not. Enabled menu lines behave normal, while disabled menu lines should be grayed out and should not respond to user events. *mexc*, for example, doesn't display disabled menu lines at all.

const-string-line This element was introduced to be a handy shortcut for:

```
    ...
    <line ref='some-id'/>
    ...
</menu> <!-- end of a menu -->
...
<line-format id='some-id'>
  <string value='some-string'/>
</line>
```

Thus, a `const-string-line` does not have a `ref` attribute, but `value` instead. The string defined through `value` will be the only thing to be displayed in the menu line, and it will be constant. Read about the string line component to learn more about constant strings. A `const-string-line` element has the same optional attributes `submenu` and `enable-vname` with the same meanings as a `line` element. Additionally, a `blink` attribute is understood and causes the displayed string to blink.

4.5.3 Line-format element

The element `line-format` is the description of the structure of a single menu line. A menu line consists of at least one line component – the components are discussed in a moment. The order of the components in which they appear in a line is defined by the order they are defined in a `line-format` element. There is nothing exciting about the element. All it must have is an `id` attribute with a unique identifier so the line descriptor can be referenced.

4.5.4 Line components

Finally, let's turn to the small entities called '*line components*'. There can be as much line components in each menu line as we want, however, we have to consider that they need to fit into a single line on the LCD. Each line component needs some space on the display. How much space it actually consumes, is finally determined by the interpreter or the program which does the drawing on the device. Nevertheless, we will give a length for each line component that the interpreter is encouraged to reserve. Fortunately, the *mexc* interpreter does respect our proposals.

4.5.4.1 Common attributes

There are optional attributes which are supported by almost all currently implemented line component elements, so let's discuss them first. If not otherwise stated , each line component element has the following attributes with an appropriate meaning and default value.

blink This attribute must be either 1 or 0. Any other value is considered to be an error. If it is set to 1, then the line component is assumed to blink with a fixed interval on the display. The interval is defined by the interpreter and cannot be set by the *mlx* compiler. Default: 0.

edit Also this attribute can hold either 1 or 0. If set to 1, the interpreter should provide a way of allowing the user to change the value of the line component. The *mexc* library, for example, provides editing of all components with a 5-button keyboard. Additionally, the interpreter library is assumed to call a custom callback handler to notify the application of a change when the user finished editing the component. For some line components it doesn't make sense to declare them non-editable. Default: 0.

update If a user changes the value of a component, then the library will update the display. But what happens if a program code (e.g. a callback or a parallel thread) changes the value? The program code doesn't have to know about a display at all, but the change needs to be reflected on it. Therefore, we can tell the library whether it should update a component periodically and how often it should do so. This attribute holds a number which defines the seconds to wait before the current value of the component should be redrawn. The number must be between 0 and 255, both values including. A zero cancels this feature, and in simple words, it means '*don't periodically update*'. Default: 0.

Note: It doesn't make sense to define an `update` value other than zero without specifying the `vname` attribute. Without `vname`, an application won't know where a component's current value in RAM is and, thus, won't be able to access and change the current value. The *mlx* compiler checks for that and prints a warning.

vname The current value of a line component is always stored somewhere in RAM. The location of the memory block is computed during compilation of the *melx* source. To make these memory blocks accessible to an application which maybe linked to an interpreter library and which does not know about the byte code (the addresses are stored there), *mlx* writes the addresses to a generated header file as '#define *name address*' where *name* is substituted with the value of the `vname` attribute and *address* with the computed address. In addition, *mlx* allocates a memory block where the address of a callback handler is stored. The interpreter library is assumed to call this handler after the user has edited a component. The address of the memory block where the routine's address is stored gets defined in the header file under the name *'CALL_' + vname*, so an application is able to register custom callbacks. As *vname* is used in a C header file, its value must be a proper C identifier. *mlx* checks for that. Default: the empty string.

4.5.4.2 Integer

An `integer` element defines a component that displays several types of integer numbers. Depending on the `type` attribute, the data type, the value range, and the displaying width differ.

```
<integer type='dd' value='42' edit='1'/>
```

type The required `type` attribute of this element determines some further details about the component. The *bytes* column in the following table shows the number of bytes the integer is stored in (this is the memory block available under `vname`). The *chars* column gives

the number of characters needed to display the component. The other columns should be self-explanatory.

type	*bytes*	*interpretation*	*value-range*	*chars*	*comment*
`dd`	1	unsigned	0 .. 99	2	
`ddd`	1	unsigned	0 .. 255	3	
`hh`	1	unsigned	0 .. 0xFF	2	hex
`sdd`	1	signed	-99 .. +99	3	sign
`sddd`	1	signed	-128 .. +127	4	sign
`DDD`	2	unsigned	0 .. 999	3	
`DDDD`	2	unsigned	0 .. 9999	4	
`DDDDD`	2	unsigned	0 .. 65535	5	
`HHHH`	2	unsigned	0 .. 0xFFFF	4	hex
`SDDD`	2	signed	-999 .. +999	4	sign
`SDDDD`	2	signed	-9999 .. +9999	5	sign

value This is the default value of the integer component. It must be in the correct range which depends on the value of `type`. This attribute is required.

blink, edit, update, vname See 4.5.4.1.

4.5.4.3 Float

The line component introduced with the `float` element is meant to present a number as a float. Currently there are two types which differ only in the format they are meant to be displayed.

```
<float type='siif' value='12.4'/>
<float type='siiif' value='-123.4'/>
```

type This is an optional attribute which defaults to `siif`, beside this it can also be set to `siiif`. The type says nothing about the value itself, but about how to display it: 's' is meant to be the sign, 'i' digits and 'f' the fraction digit. An interpreter should display the float as specified with a dot or a comma between the last digit and the first fraction digit.

value This specifies the component's initial value and is required. It has to be a float.

blink, edit, update, vname See 4.5.4.1.

Depending on the `type` attribute this component should take up 5 or 6 characters on the display.

4.5.4.4 String

The string component is probably the most frequently used component at all. Very often it is used to display static information that will never change during the execution of a program. Therefore, this component is somewhat special.

If the component is not declared to be editable, the attributes `update` and `vname` have no effect. In this case, the current value of the string will never be copied to RAM. Thus, a program which links to an interpreter library will not be able to access the string. Only the library will extract the string out of the byte code and display it.

Strings are always stored the Pascal way. This means they are not zero terminated, but their length is stored in the first byte. The string itself begins in the second byte. The fact that the length is saved in a one-byte cell is limiting a string's length to 255 characters.

```
<string value='a constant string'/>
<string value='an editable string' edit='1' blink='1'/>
```

value This is the string to be displayed. If the string is editable, this is the string's default value. This attribute is required.

blink, edit, update, vname See 4.5.4.1.

The length of the string determines how much space is needed on the display.

4.5.4.5 Counter

The counter component is a number on the display which, when edited, can be increased or decreased by a defined value. Depending on the type, the counter handles *signed two-byte integers* or *floats*.

```
<counter type='integer' edit='1'
         value='42' min='-100' max='100' step='2'/>
<counter type='float' edit='1'
         value='12.4' min='10.0' max='15.0' step='0.5'/>
```

type This attribute is required and determines the data type to use. It can be *integer* or *float*.

value This attribute is required and sets the initial value of the counter. It must be between `min` and `max`, including both values.

min This attribute is required and sets the lower boundary of the counter. It must be smaller than `max`.

max This attribute is required and sets the upper boundary of the counter. It must be greater than `min`.

step This attribute is required and is the value by which to increase or decrease the counter.

blink, edit, update, vname See 4.5.4.1.

How much space is needed on the LCD to display the component is determined by the attributes `min` and `max`. The longest of those two sets the width of the component. For example, the first counter in the code shown above would need 4 characters as the counter can take up a value of -100.

4.5.4.6 Switch

A switch component is a binary field whose bits can be set *on* or *off*. By defining '*' for the *on*-state and '.' for the *off*-state, a switch would typically be drawn as `*..**.**` on an LCD.

To define such a component, we will use the `switch` element with `switch-item`s as children. The number of the children is limited to 32 but must be at least 1. The `switch` is one of few elements that is not empty. The example given below defines the above discussed field.

```
<switch on-char='*' off-char='.' edit='1'>
  <switch-item info='Budweiser' value='1'/>
  <switch-item info='Dobrovar' value='0'/>
  <switch-item info='Steiger'  value='0'/>
  <switch-item info='Eger' value='1'/>
  <switch-item info='Plzen' value='1'/>
  <switch-item info='Martiner' value='0'/>
  <switch-item info='Gambrinus' value='1'/>
  <switch-item info='Smaedny Mnich' value='1'/>
</switch>
```

Attributes of a `switch` are ...

on-char This attribute is optional and defines the character to use for the *on*-state. Default: '*'

off-char This attribute is optional and defines the character to use for the *off*-state. Default: '.'

blink, edit, update, vname See 4.5.4.1.

Attributes of a `switch-item` are ...

info This attribute is required and defines an help string to be displayed when user edits the appropriate bit.

value This attribute is required and defines the initial state of a bit. It can either be 1 (*on*) or 0 (*off*).

How much space on the LCD this component needs depends on the number of specified bits. For each bit it should take up one character.

4.5.4.7 Option

An option is a component to let the user choose one item out of a fixed list of alternatives. The code shown below, for example, defines an option to let the user choose a name of a month.

```
<option edit='1' default='o-03'/>
  <option-item value='Jan' id='o-01'/>
  <option-item value='Feb' id='o-02'/>
  <option-item value='Mar' id='o-03'/>
  <option-item value='Apr' id='o-04'/>
  ...
</option>
```

As shown, the `option` element is not empty and must contain at least one `option-item` element, but no more than 255. `option` elements have the following attributes:

default This attribute is required and is a reference to an item which should be displayed by default. The value must be the same as the value of an `id` attribute of one `option-item` within the appropriate `option`.

blink, edit, update, vname See 4.5.4.1.

The `option-item` has two required attributes:

value This string is to be displayed when the item is selected.

id A unique identifier.

The length of an `option` on the display is determined by the longest `option-item`'s value.

4.5.4.8 Time

The time component is there to display a time with or without seconds and may look like `12:42` in the 'short' format or like `12:42:37` in the 'long' format. The delimiter character printed between the single time parts is not set by *mlx* but provided by the interpreter.

```
<time type='long' hours='12' minutes='42' seconds='37'
      vname='CURRENT_TIME_ADDR' update='1'/>
<time type='short' hours='12' minutes='42' seconds='0'
      edit='1' vname='ALARM_ADDR'/>
```

Beside the time specific attributes, this example also shows the usage of the two attributes `update` and `vname`. With the first time component, a program can periodically update the variable at the address CURRENT_TIME_ADDR and the interpreter will update/redraw the component every

second, when it is on the screen. The second time component, for instance, can serve as an input field to let the user define an alarm.

The data type for the time value is a three or two bytes structure respectively. The first byte holding the hours, the second byte holding the minutes and for the long time format the third byte holding the seconds.

The following attributes are defined for the time element:

hours This attribute is required and is the 'hours' component of the time. It must be a number between 0 and 23, including both.

minutes This attribute is required and is the 'minutes' component of the time. It must be a number in the range of 0 to 59.

seconds This attribute is required and is the 'seconds' component of the time. It must be a number in the range of 0 to 59.

type This attribute is optional and defaults to 'short'. It can also be set to 'long' and indicates whether the time component will have seconds or not.

blink, edit, update, vname See 4.5.4.1.

The length in characters of a short time should be 5, of a long time it should be 8.

4.5.4.9 Date

A date is similar to the time component. It could be displayed as `2005-10-24` in the long format or `05-10-24` in the short one. In the end, it depends on the interpreter. The main difference is that a 'short' date cannot hold values greater than 255 for a year, while in the long format, it can be up to 9999. As with the time component, also for dates, *mlx* has no influence on the delimiter character between the year, month, and day parts.

```
<date type='short' year='2005' month='10' day='24'/>
<date type='long'  day='1' month='1' year='2004'/>
```

The following attributes of a `date` element are defined:

day This attribute is required and must be a number between 1 and 31, including both values.

month This attribute is required and must be a number in the range of 1 to 12.

year This attribute is required and must be a number in the range of 0 to 9999. Specifying 1999 for a short date is possible, however the compiler assumes 99 is meant.

type This attribute is optional and defaults to 'short'. Beside its default value, it can be set to 'long'. This attribute denotes the size of the year.

blink, edit, update, vname See 4.5.4.1.

The length of the displayed date component should be 10 characters for a long type and 8 for a short type.

4.5.4.10 Trigger

A trigger differs from all the components mentioned above. It actually doesn't display any information at all, but is meant to be a "soft-button" which, when activated, calls a routine to perform some action.

```
<string value='Reset: '/>
<trigger vname='reset_system'/>
```

The shown code snippet would probably be displayed as '`Reset: [X]`' in a line. The interpreter is responsible for providing a way to activate the trigger, when the user wants to do so. The following attributes of a `trigger` element are defined:

vname This required attribute has the same meaning as described in 4.5.4.1 with the exception that only the `CALL_` definition will result in the generated header file.

password With this optional attribute set, an interpreter should request for this password before calling the installed handler routine.

blink This attribute is optional and has the same meaning as described in 4.5.4.1

It depends on the interpreter how many characters a trigger takes up. The *mexc* interpreter library displays a trigger as '`[X]`', password protected triggers as '`[P]`'. Thus, it takes up three characters.

4.5.4.11 Horizontal fill

An `hfill` element is actually a fictional component and resolves into a constant string. Its purpose is to provide a way to put fixed or variable sized gaps between the other components. For example, if we want to have a line with right-aligned components, we would use an `hfill`.

```
<line-format id='foo'>
  <string value='temp:'/>
  <hfill char='.'/>
  <integer type='dd' value='20' update='10'
           vname='cur_temperature'/>
</line-format>
```

In the example above, *mlx* would compute how many characters to put between the given string and the integer, so that the integer is aligned at the right edge of the display. An interpreter would draw the line as '`temp:.....20`' on a 16 characters wide LCD.

We can even specify more than one `hfill` element in a `line-format`. In this configuration, *mlx* tries to distribute the free (character) space to the specified gaps equally.

```
<line-format id='bar'>
  <string value='X'/>
  <hfill char='-'/>
  <string value='Y'/>
  <hfill char='.'/>
  <string value='Z'/>
</line-format>
```

On a 16 character wide LCD the given example would produce a '`X------Y.......Z`' printed line. If we carefully count the characters, we will notice that the dashes count one more than the dots. This is because the 13 remaining characters, which are to be equally distributed, cannot be divided without a remainder. *mlx* prefers the last gap and assigns the rest of the free characters to it.

The `hfill` element understands two optional attributes.

char This attribute defines the character to fill the gaps with. Default: a blank (' ')

count With the `count` attribute the number of characters to be put into a gap can be defined. The default value, zero, causes the gap to grow as much as possible.

The width of this element is variable or fixed (via `count` attribute), but it can also be zero if there is no space to distribute, so it is not guaranteed that there will be a gap between two components.

4.6 Command line options

Before we can use the *mlx* compiler, we need to understand its command line options. The command `python mlx.py --help` at the shell prompt will print a list of options recognized by the program. We should get a listing as show in figure 4.1.

Only long format options are supported, and two arguments are required to the compiler. Let's step through the command line parameters to define their meanings.

--help This option prints a listing as shown in figure 4.1.

--version This option prints the version number of the compiler and exits the program.

--dont-validate This option causes the compiler not to validate the source file. Given this option, the compiler will not use the *xmlproc* modules from the "`python-xml`" package, but modules from the standard library that are installed with the default Python distribution. This

```
~mlx% python mlx.py --help
usage: mlx.py: [options] mem-origin <filename>
options are:
  --align                 do align addresses
  --endian={little|big}   use specified byte-order (def: big)
  --awidth=n              address bus width in bytes (def: 2)
  --mem-optimization      do optimize memory layout

  --no-header             do not produce the header a.h file
  --binary                produce a.bin file instead of a.c
  --output base           'base' as name for output filenames

  --max-line-width=n      num characters in one line (def: 18)
  --max-title-width=n     num characters for title (def: 16)

  --dont-validate         do not validate the source file
                          xmlproc required for validating

  --help                  print this text, then exit
  --version               print the version, then exit
```

Figure 4.1: Command line parameters of *mlx*

enables the use of the compiler on machines without the "`python-xml`" package installed. However, if the source file isn't properly structured the compiler's result is undefined.

--max-line-width This option takes a numerical argument and tells the compiler to print warnings about menu lines which are longer than the given number. As we learned about the `hfill` element, the compiler can right-align components or stretch them from each other. This requires a line width to be given. By default it is 18[2].

--max-title-width As with the previous option, also this one takes a numerical argument and tells the compiler to print warnings about menu titles which are longer than the given value. By default it is 16[3].

--output This option requires an argument. By default, the *mlx* compiler will output two files named '`a.h`' and '`a.c`'. With the `--output` option the '`a`' of the filenames can be changed to a custom name.

--binary By default, the compiler will output the generated byte code as an array in a C file. Using this option, the compiler will generate a binary file instead of the C file. The binary file will be appended the '*.bin*' suffix and contains the raw bytes of the binary menu image. It is useful for applications that want to the load byte code dynamically. The `gsim` simulator,

[2] *mexc* uses 18 characters of a 20 character wide display for menu data. The remaining two columns are reserved, one for the current line indicator and the other for the submenu indicator.

[3] *mexc* uses 16 characters for the title on a 20 column display. The remaining 4 columns are reserved to display some other information.

which was written as a part of this project, can run different byte codes without the need of being recompiled.

--no-header This option suppresses the generation of the header file.

--align For each line component, with the exception of constant strings, the compiler allocates a memory block of an appropriate size somewhere above `mem-origin`. We can tell the compiler to align memory blocks with a size of two bytes on even addresses and memory blocks with a size of 3 or more bytes on addresses divisible by 4. Some hardware may require this.

--mem-optimization This option has only an effect if it's used together with the `--align` option. When optimizing, the gaps, that originates from aligning memory blocks, get filled by other memory blocks. This option can cause a considerable reduction of memory consumption at the cost of the fact that a single line's components don't have to have increasing addresses for their current values. In most cases this should be no problem.

--awidth This option is important to be properly set and defines the width of an address. It is the size of a pointer and can be determined with the C `sizeof` operator, which depends on the compiler and the target platform the produced byte code will be used at.

--endian This option causes the compiler to output values, which are stored in more than one byte (e.g. short or floats), either in little- or big-endian. It is important to know on which platform the byte code will be interpreted.

mem-origin This is a required command line parameter and has to be a numerical value (must begin with '`0x`' for hexadecimal format) and tells the compiler to allocate memory blocks above this address. This will often be the address of the beginning of RAM on the target system.

filename This is the path to the *melx* source file.

4.7 CMF - Compact Menu Format

The "Compact Menu Format" is a definition of the byte code that *mlx* produces. Its current version is 0.4. It is a compact binary representation of a menu hierarchy specified by an *melx* source. The binary format is very similar to Mr. Högl's PMF (Portable Menu Format) [11], however, CMF differs to make the code more compact and efficient.

The following sections reflect many things already discussed in section 4.5 and will be interesting to programmers who want to implement a CMF parsing program.

4.7.1 Notation

In the following description we will use the convention to enclose terminal symbols in braces `< .. >`. Within these braces a terminal symbol is separated by a colon followed by a number

which denotes the size in bytes of the symbol. With `<foo:2>`, for example, we have a terminal symbol which takes up two bytes. Sometimes we will have brackets with an interval or a single number behind a terminal definition. A `<foo:2>[3,0]`, for instance, denotes that we refer to the bits 3, 2, 1 and 0 of the terminal symbol 'foo' which has a size of two bytes. All other symbols, namely those not enclosed in `< .. >` braces, are non-terminal symbols. To indicate zero or more repetitions of a symbol, it is enclosed in curly braces `{ .. }`.

Some terminal elements (e.g. strings or passwords) are noted as `<element:n+1>`. The '`+1`' indicates that the element is a string, and strings are stored the Pascal way in CMF. They are not zero terminated, but their first byte, referred to as the 'length byte', which holds the value `n` as an one-byte unsigned integer, is followed by `n` bytes holding the string's characters. Thus a string actually takes up `n+1` bytes in CMF.

4.7.2 Overall structure

```
byte-code ::=
      prolog menu-table { menu-table }

prolog ::=
      <pmf-id:3>
      <major-version:1>
      <minor-version:1>
      <delay-to-top:1>
      <delay-clr-help:1>
      <delay-password:1>
      <byte-order-mark:2>

menu-table ::=
      <menu-title-string:n+1> (*) menu-line { (*) menu-line }

menu-line ::=
      <ldtag:1> line-opts (*) line-comp { (*) line-comp }

line-comp ::=
      <lctag:1> line-comp-opts
```

Figure 4.2: Overall structure of CMF

The definition in figure 4.2 shows the structure of a CMF formated menu description. A '`(*)`' in the definition is an indicator for an optional padding zero byte. Sometimes such bytes are necessary to make the following structures specially aligned. Currently, there are two alignment rules that apply to the byte code.

- A `menu-line` is always aligned on a non-even offset within the byte code. Padding zero bytes are put in front of it to make the offset of a `<ldtag:1>` not divisible by 2.

- A `line-comp` is always aligned on an even offset within the byte code. A padding zero byte may precede the structure to make the offset of a `<lctag:1>` divisible by 2.

There is no indication which menu table is actually the first to be displayed. By convention, the first menu table – the top-level menu table – to be displayed is the `menu-table` following immediately the `prolog`.

4.7.3 prolog

pmf-id ... is an array of 3 characters. This array is filled with 'C', 'M', and 'F' (or in numbers 0x43, 0x4D, and 0x46) in this order. A parsing program must check for this to ensure it has the right binary data.

major-version ... denotes the major version number of the byte code.

minor-version ... denotes the minor version number of the byte code.

delay-to-top ... is an unsigned-byte integer and gives the number of seconds an interpreter should wait before it is supposed to return to the top-level menu table. A value of zero indicates the interpreter should never return to the top-level menu table automatically.

delay-clr-help ... is an unsigned-byte integer and gives the number of seconds for how long a help string should be displayed. A value of zero indicates a help string should not be cleared automatically.

delay-password ... is an unsigned-byte integer and gives the number of seconds after which a password query should be aborted when the user makes no input. A value of zero indicates an interpreter should infinitely wait for the password.

byte-order-mark ... is an unsigned two-byte integer with the fixed value of 0xFEFF. However, accessing it via an array of bytes it can have two values: 0xFFFE in case the number was stored in big-endian, or 0xFEFF in case it was stored in little-endian. A parsing program can easily determine whether it supports the proper byte-order with the following code:

```
unsigned short * p = (unsigned short *)&byte_code[8];
if (*p != 0xFEFF)
  ; /* wrong byte order determined */
```

Listing 4.1: Checking the byte–order mark

4.7.4 menu-line

A menu line begins with an unsigned one-byte integer (`<ldtag:1>`) which decides what fields are available from the line options. Following the `line-opts` there is always at least one line component. Let's look at the bits of an `ldtag`.

```
if <ldtag:1>[1,0] == 00    # not first and not last line
    line-opts ::= <next-line-ofs:2> <prev-line-ofs:2>
if <ldtag:1>[1,0] == 01    # first line but not last
    line-opts ::= <next-line-ofs:2>
if <ldtag:1>[1,0] == 10    # last line but not first
    line-opts ::= <prev-line-ofs:2>
if <ldtag:1>[1,0] == 11    # is last and is first line
    # nothing for line-opts in this case (maybe submenu)
    line-opts ::=
if <ldtag:1>[2] == 1       # dyn-enable feature
    line-opts ::= ... <var-addr:2>
    # var-addr is appended to the previous options
if <ldtag:1>[3] == 1       # has submenu
      line-opts ::= ... <smenu-abs-ofs:2>
      # submenu-abs-ofs is appended to previous options
if <ldtag:1>[3,4] == 11   # submenu password
    line-opts ::= ... <password-str:n+1>
    # password is appended to the previous options
```

Figure 4.3: Definition of `line-opts` in CMF

bitmask	meaning if the appropriate bit is set in `<ldtag:1>`
`0x01`	is first menu line (no preceding menu lines)
`0x02`	is last menu line (no following menu lines)
`0x04`	menu line can be dynamically enable/disabled
`0x08`	menu line points to a submenu
`0x10`	submenu is password protected

Depending on these bits the structure of a `line-opts` must be dynamically assembled. Accessing values following this structure can be a little bit problematic because it has not a fixed size. However, this can be efficiently implemented with a lookup table holding the sizes for each case. The exact definition of the dynamic structure depending on the shown flags is shown in figure 4.3.

The '...`-ofs`' fields, namely `next-line-ofs` and `prev-line-ofs`, are offsets to the previous or the next menu line respectively. They count from including the `ldtag` and contain the optional padding zero byte. This means, in the case of a next menu line, we need to add the `next-line-ofs` to the address of the current `ldtag` to obtain the address of the next.

The optional `var-addr` field is an offset into RAM and points to an unsigned one-byte integer that should be used as a boolean value. This byte is often referred to as the '*enable byte*' and its value as the '*enable value*'. The interpreter has to initialize this byte and interpret its value accordingly.

`smenu-abs-ofs` holds an absolute offset to the submenu. An absolute offset is counted from including the first byte of the first menu table. This is the first menu table located directly behind the `prolog`. Thus, to access a submenu we need to add to `smenu-abs-ofs` the address of the `prolog` and its size.

To access the first line of a `menu-table` we need to look at the absolute offset of the byte immediately following the menu title. Because all menu lines are aligned on non-even offsets, we must increase the offset of the byte following the menu title by one if it is even, and thus, jump over a padding zero byte. When accessing menu lines on the base of '`...-ofs`' fields, there is no need to worry about the padding zero byte.

4.7.5 line-comp

A line component is the actual entity that is displayed in a menu line. Currently there are 23 different types of line-components which are to be implemented by an interpreter and there is still place for another eight components. The type of a component is defined by the first five bits of the `lctag` and determines the actual byte code structure. The following table gives an overview of the available components which are explained in more detail later, each on its own. The *mask* column suggests how to display the components.

`<lctag:1>[4,0]`	type	mask	comment
`0x00`	uchar	dd	unsigned one-byte integer
`0x01`	uchar	ddd	-
`0x02`	uchar	hh	hex
`0x03`	char	sdd	signed one-byte integer
`0x04`	char	sddd	-
`0x05`	uint2	DDD	unsigned two-byte integer
`0x06`	uint2	DDDD	-
`0x07`	uint2	DDDDD	-
`0x08`	uint2	HHHH	hex
`0x09`	int2	SDDD	signed two-byte integer
`0x0a`	int2	SDDDD	-
`0x0b`	float	SII.F	ieee-754 float
`0x0c`	float	SIII.F	-
`0x0d`	counter	DD..D	signed two-byte int counter
`0x0e`	fcounter	SII.F	ieee-754 float counter
`0x0f`	time-long	HH:MM:SS	-
`0x10`	time-short	HH:MM	-
`0x11`	date-long	YYYY-MM-DD	-
`0x12`	date-short	YY-MM-DD	-
`0x13`	switch	".*..**."	max. 32 items
`0x14`	option	"..."	opt1, opt2, ...
`0x15`	string	"..."	pascal strings
`0x16`	password	"[P]"	-
`0x17`	trigger	"[X]"	-
`0x18 - 0x1f`	-	-	-

There are three further flags in `<lctag:1>`. The following table explains their meanings.

<lctag:1>[5]	description
0	(read-only) component should not be editable by the user
1	(read-write) component should be editable by the user

<lctag:1>[6]	description
0	(dont-blink) component should not blink on the display
1	(do-blink) component should blink on the display

<lctag:1>[7]	description
0	(not-last) after this component there is another one
1	(last) this component is the last in the menu line

In the following sections each line component's byte code structure is given with some hints. Nearly all structures begin with the following fields:

`<update:1>` A one-byte unsigned integer to define an interval in seconds in which an interpreter should redisplay the current value of the component. Of course, an interpreter can update it only if it is currently displayed. An interval of zero disables the automatic updates.

`<call-addr:2>` A relative pointer (an unsigned two-byte integer) to a memory block where a handler's address is installed. This handler should be called after the user edited a component. The pointer being relative is explained in a moment.

`<var-addr:2>` A relative pointer (an unsigned two-byte integer) to a memory block where the current value of the component is stored. How large the memory block is and how it has to be interpreted is determined by a component's type.

A *relative pointer* is just like a relative path. An interpreter will be given a base address and it is supposed to add this address to all relative pointers to actually access memory at the right location.

As mentioned, a `line-comp` structure is always stored on an even offset within the byte code. When accessing a `line-comp` we must jump over optional padding zero bytes. We'll just fetch a component's offset as usual but increment it by one, in case it is not even.

4.7.5.1 uchar 'dd'

```
if <lctag:1>[4,0] == 0x00
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:1>
```

`default` and the corresponding memory block, which is accessible by the `var-addr` relative pointer, are unsigned one-byte integers. `default` is the component's initial value. An interpreter is supposed to display this component as a two-digit decimal number without a sign ('+'). The size of the code for this component including the `lctag` is 7 bytes.

4.7.5.2 uchar 'ddd'

```
if <lctag:1>[4,0] == 0x01
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:1>
```

This component is the same as "uchar dd" with the exception that it is supposed to be displayed as a three-digit decimal number.

4.7.5.3 uchar 'hh'

```
if <lctag:1>[4,0] == 0x02
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:1>
```

This component is the same as "uchar dd" with the exception that it is supposed to be displayed as a two-digit hexadecimal number.

4.7.5.4 char 'sdd'

```
if <lctag:1>[4,0] == 0x03
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:1>
```

`default` and the corresponding memory block are signed one-byte integers. `default` is the components initial value. An interpreter is supposed to display the component as a two-digit decimal number with a sign in front of it ('+/-'). The size of the code for this component including the `lctag` is 7 bytes.

4.7.5.5 char 'sddd'

```
if <lctag:1>[4,0] == 0x04
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:1>
```

This component is the same as "char sdd" with the exception that it is supposed to be displayed as a three digit number with a sign.

4.7.5.6 uint2 'DDD'

```
if <lctag:1>[4,0] == 0x05
   <update:1>
```

```
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

`default`, which is the component's initial value, and the corresponding memory block are unsigned two-byte integers. An interpreter is supposed to display the component as a three-digit decimal number without a sign ('+'). The size of the code for this component including the `lctag` is 8 bytes.

4.7.5.7 uint2 'DDDD'

```
if <lctag:1>[4,0] == 0x06
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

This component is the same as "uint2 DDD" with the exception that it is supposed to be displayed as a four-digit decimal number.

4.7.5.8 uint2 'DDDDD'

```
if <lctag:1>[4,0] == 0x07
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

This component is the same as "uint2 DDD" with the exception that it is supposed to be displayed as a five-digit decimal number.

4.7.5.9 uint2 'HHHH'

```
if <lctag:1>[4,0] == 0x08
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

This component is the same as "uint2 DDD" with the exception that it is supposed to be displayed as a four-digit hexadecimal number.

4.7.5.10 int2 'SDDD'

```
if <lctag:1>[4,0] == 0x09
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

`default`, which is the component's initial value, and the corresponding memory block are signed two-byte integers that should be displayed as three-digit decimal numbers with a sign in front of it. The size of the code for this component including the `lctag` is 8 bytes.

4.7.5.11 int2 'SDDDD'

```
if <lctag:1>[4,0] == 0x0a
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:2>
```

This is the same as the "int2 SDDD" component with the exception that it should be displayed as a four-digit number with a sign.

4.7.5.12 float 'SII.F'

```
if <lctag:1>[4,0] == 0x0b
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:4>
```

`default`, the initial value, and the corresponding memory block are IEEE-754 single precision (32-bit) floating point numbers. They should be displayed with a sign followed by two digits before the decimal point and one digit after the decimal point. The size of the code for this component including the `lctag` is 10 bytes.

4.7.5.13 float 'SIII.F'

```
if <lctag:1>[4,0] == 0x0c
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <default:4>
```

This component is the same as 'float SII.F' with the exception that it should be displayed with one more digit before the decimal point.

4.7.5.14 counter

```
if <lctag:1>[4,0] == 0x0d
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <min:2>
   <max:2>
   <step:2>
   <default:2>
   <field-width:1>
```

`default`, the counter's initial value, the corresponding memory block, `min`, `max`, and `step` are signed two-byte integers. A counter should be in-/decremented by `step` within the range [`min`; `max`]. `field-width` is an unsigned one-byte integer and gives the width in characters needed to display the counter within the specified value range. The size of the code for this component including the `lctag` is 15 bytes.

4.7.5.15 fcounter

```
if <lctag:1>[4,0] == 0x0e
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <min:4>
   <max:4>
   <step:4>
   <default:4>
   <field-width:1>
```

This is essentially the same as a 'counter' with the exception that `default`, the corresponding memory block, `min`, `max`, and `step` are IEEE-754 single precision (32-bits) floating point numbers to be displayed as a 'float SII.F' component. The size of the code for this component including the `lctag` is 23 bytes.

4.7.5.16 time-long

```
if <lctag:1>[4,0] == 0x0f
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <hours:1>
   <minutes:1>
   <seconds:1>
```

`hours`, `minutes` and `seconds` are unsigned one-byte integers. The memory block, which holds the current value, is a data type of three bytes with the first byte being the hours, the second byte being the minutes and the third byte being the seconds of the time. In C we would describe the data type with a struct `cmf_time_t` as shown in listing 4.2.

```
struct cmf_time_t {
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
};
```

Listing 4.2: Definition of cmf_time_t

The size of the code for this component including the `lctag` is 9 bytes.

4.7.5.17 time-short

```
if <lctag:1>[4,0] == 0x10
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <hours:1>
   <minutes:1>
```

This component is the same as “time-long” but without seconds. The size of the code for this component including the `lctag` is 8 bytes.

4.7.5.18 date-long

```
if <lctag:1>[4,0] == 0x11
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <day:1>
   <month:1>
   <year:2>
```

`day` and `month` are unsigned one-byte integers, while `year` is an unsigned two-byte integer. The corresponding memory block, which holds the current value, has a size of 4 bytes and the C structure definition as shown in listing 4.3.

```
struct cmf_date_t {
  unsigned char day;
  unsigned char month;
  unsigned short year;
};
```

Listing 4.3: Definition of cmf_date_t

The size of the code for this component including the `lctag` is 10 bytes.

4.7.5.19 date-short

```
if <lctag:1>[4,0] == 0x12
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <day:1>
   <month:1>
   <year:1>
```

This is the same as a “date-long” with the exception that the year is an unsigned one-byte integer. The size of the code for this component including the `lctag` is 9 bytes.

4.7.5.20 switch

```
if <lctag:1>[4,0] == 0x13
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <length:1>
   <nswitch:1>
   <on-char:1>
   <off-char:1>
   <default:4>
   <string:n+1>
   ...
   <string:n+1>
```

`length` is an unsigned one-byte integer that gives the size in number of bytes of this component including the `lctag`. `nswitch` is also an unsigned one-byte integer and defines the number of valid switches/bits and help strings. The maximum can be 32. `on-char` and `off-char` are both characters to be displayed for a bit in the appropriate state. The string array at the end of the struct defines a help string (Pascal style) for each bit. `default`, which is the switch's initial value, and the corresponding memory block are 4-byte arrays with the first 8 switches/bits in the first byte, the next 8 switches/bits in the second byte and so on. Due to the fact that some platforms don't support 32-bit data types, this component has not been implemented as an unsigned four-byte integer. Nevertheless, we can easily access each bit with the C code given in listing 4.4.

```
unsigned char * mask = &mem_block_of_switch[0];
for (i = 0; i < 32; i++)
{
   byte_index = i / 8
   bit_index = i % 8
   if (mask[byte_index] & (1<<bit_index))
      ; /* bit is set */
   else
      ; /* bit is not set */
}
```

Listing 4.4: Accessing each bit of a switch component

The size of the code for this component including the `lctag` is variable and defined in the `length` field.

4.7.5.21 option

```
if <lctag:1>[4,0] == 0x14
   <update:1>
   <call-addr:2>
   <var-addr:2>
   <length:1>
```

```
<nopts:1>
<field-width:1>
<default:1>
<string:n+1>
...
<string:n+1>
```

`length` and the string array at the end of the component's code have the same meaning as within a "switch". `nopts` is an unsigned one-byte integer giving the number of options. `default`, the initial value, and the corresponding memory block are unsigned one-byte integers being indexes into the string array. `field-width`, an unsigned one-byte integer, gives the width in characters of the longest string. The size of the code for this component including the `lctag` is variable and defined in the `length` field.

4.7.5.22 string

```
if <lctag:1>[4,0] == 0x15
   if <lctag:1>[5] == 0     # line-comp is read-only
      # constant string
      <string:n+1>
   else                     # line-comp is read-write
      <update:1>
      <call-addr:2>
      <var-addr:2>
      <string:n+1>
```

The string component is somewhat special. If the component is not editable, then the value, a Pascal string, is following immediately the `lctag`, and the size of the component's code is the length of the string plus two (length byte of the string + `lctag`). In this case, there is no relative pointer and no corresponding memory block.

If the string is editable, the corresponding memory block has the length of the string plus one (the length byte). The size of this component in this case is also variable and computed as 'length-of-the-string + 7'.

4.7.5.23 password

```
if <lctag:1>[4,0] == 0x16
   <unused:1>
   <call-addr:2>
   <string:n+1>
```

This component is a password protected trigger. The 'is-editable' flag in `<lcdtag:1>` is always set. But instead of changing the appearance of the component, an interpreter should ask for a password, verify it, and if it was correct, call the installed callback handler which address is stored at the location the relative pointer `call-addr` points to.

It is assumed that this component will be displayed as a '`[P]`'. Its code size in bytes including the `lctag` is the length of the password plus 5.

4.7.5.24 trigger

```
if <lctag:1>[4,0] == 0x17
   <unused:1>
   <call-addr:2>
```

This component is essentially the same as a "password" with the exception that there is no password to be requested. It is assumed to be displayed as a '`[X]`'. This component's code size in bytes including the `lctag` is 4.

4.8 m2melx.py

For programmers, who are already familiar with the M-Language and who want to switch to *melx*, this section may be of interest. As *melx* was introduced to replace the M-Language, a script has been written to convert M-Language documents to *melx* documents, and thus, allow developers a quick movement towards the *mlx* compiler.

The converter script named `m2melx.py` takes an M-file and creates an semantically equal menu definition in the *melx* language. The `m2melx` script can be started with or without parameters. In the later configuration, the converter expects its input from stdin and prints the result to stdout. If started with the '`-h`' option, the message shown in listing 4.5 will be printed. As listed there, it is possible to specify an input file and a filename where to write the result.

```
~mlx% python m2melx.py -h
usage: python m2melx.py [-h] [-o <output-filename>] [<input-filename>]
```

Listing 4.5: Command line options of m2melx

Note: The converter does no error checking, and assumes that the menu description given in the M-Language is correct. If it isn't, the result of the converter is undefined.

4.9 Writing extensions

In this section we will look at how *mlx* can be extended with custom line components. The compiler was written in a manner that makes it not too hard to integrate a programmer's own components. However, some experience with DTDs, SAX, and Python programming is required. An understanding of CMF, which is described in section 4.7, is essential.

Due to the compactness of the output format, there is place for only eight new line components. However, this should be enough. Before starting to make changes to *mlx*, we should investigate whether it's worth the trouble at all. We should try to realize our idea with an already implemented component, because we need to consider that, beside *mlx*, also the byte code interpreter needs to be extended, too.

Throughout this section we will introduce an example component called "checkbox" that actually could be realized with an `option`. However, our `checkbox` will produce fewer bytes. It will

have the four attributes *blink*, *edit*, *update*, and *vname* as described in section 4.5.4.1. Of course it will have a default *value*. We will assume that an interpreter will display the component as '`(x)`' (checked) or '`(o)`' (unchecked), and thus use only three characters for it on the screen. When used a lot in a menu definition, the new component will save a considerable amount of memory in comparison with an `option`. The byte code for a `checkbox` will be the same as for "uchar dd" which is described in section 4.7.5.1.

4.9.1 Extending the language

At first we need to extend the *melx* language which is defined through `melx.dtd`. A copy of this DTD is given in section 4.10. The XML parser, precisely spoken the event handler, used by *mlx* is written in a manner that makes it simple to handle empty elements, however, nesting them is also possible. To introduce a new component in *melx* we need to add an element definition to the DTD. Listing 4.6 shows what we would append to `melx.dtd`.

```
<!ELEMENT checkbox EMPTY>
<!ATTLIST checkbox %common-lcomp-attrs; value CDATA (1|0) "0" >
```

Listing 4.6: Definition of a checkbox element

The trick with the shown definition is that it uses the `%common-lcomp-attrs;` attribute entity already defined in `melx.dtd`. By using this entity, the new element gets attributes that are common to almost all components. Additionaly, a *value* attribute that can hold either '`1`' or '`0`' was introduced to the `checkbox`. It has a default value, and thus the menu programmer will not need to explicitly specify this attribute.

To use the new element, it must be made available as a child of `line-format`. The element's name must be put into the list of valid children of the container. In our example, the definition of `line-format`, after inserting `checkbox`, would look like in listing 4.7.

```
<!ELEMENT line-format
   (hfill|integer|string|counter|option|switch|
    time|float|date|trigger|checkbox)+ >
<!ATTLIST line-format id ID #REQUIRED >
```

Listing 4.7: Extended line–format with checkbox

Now we are allowed to write *melx* documents with `checkbox`es. The compiler will not complain about the new element when validating, however, it will still do nothing with it, but quietly ignore it. The next step is to write a byte code generator for `checkbox`es and then couple it with the parser.

4.9.2 Writing a byte code generator

The source code of *mlx* has a file called `cmf.py` which implements the byte code generators for all components. In this file we can find the class `Lc` which is the parent of all generators and which we will use to inherit our new class from.

Before we begin to implement the inherited class, we should make the following change to `Lc`. It holds a dictionary called `ident_id_map` which represents a mapping between logical names and IDs. These IDs are the `lctag`'s first five bits as described in section 4.7.5. In our example, we will add a '*checkbox*' to the dictionary with the next free ID as shown in listing 4.8.

```
ident_id_map = {'uchar-dd' : 0x00,
                ...
                'trigger'  : 0x17,
                'checkbox' : 0x18 } # new
```

Listing 4.8: Extended ident_id_map with checkbox

Now we need to subclass `Lc`. We will call the new class `LcCheckbox`, its implementation is show in listing 4.9, but let's first take a look at the meaning of the methods to be implemented:

`__init__` In the constructor of the subclass the first thing to do is to call the constructor of the base class with appropriate parameters.

`alloc_addrs` As the documentation of `Lc` says, this method gets called before the byte code generation and provides a chance to the component to let *mlx* know that it needs some memory space in RAM. When this method is called, a component only registers an allocation request. After all components have registered their requests, *mlx* begins to compute the addresses, and then they can be retrieved. In our `checkbox` example, the component will register a one-byte data type for the current value and a function-address data type for the address of a handler which is to be called after the component has been edited.

`bc` When this method gets called, the byte code generation process is active. This method is assumed to return a list of bytes that represents the byte code for a component. The implementation should always use the passed `emitter` object to output the byte list. Addresses of memory blocks, which were registered in the `alloc_addrs` method, can now be retrieved via the passed `allocator` object.

`str_len` This method is intended to answer the following question: "How many characters does this component consume in one line at most?". Having this information, *mlx* can warn the user if a line contains to many components which will not completely fit into it. In the `checkbox` example, this method will simply return the constant `3`.

```
class LcCheckbox (Lc):
  def __init__ (self, value, blink, writable, update, vname):
```

```
    # 'value': the default value of the component
    # 'writable': should the user edit the component?
    Lc.__init__ (self, blink, writable, update, vname)
    self.default = value

  def alloc_addrs (self, allocator):
    self.reg_vname = \
             self.vname or allocator.generate_new_name ()
    put_into_header = self.vname and True or False
    # register a one byte block for the current value
    allocator.reg_var (self.reg_vname, 'unsigned char', \
                       1, None, put_into_header)
    allocator.reg_cb (self.reg_vname, None, put_into_header)

  def bc (self, emitter, allocator):
    return emitter.uchar (self.lctag ('checkbox'))        + \
       emitter.uchar (self.update)                        + \
       emitter.uint2 (allocator.cbaddr (self.reg_vname))  + \
       emitter.uint2 (allocator.varaddr (self.reg_vname)) + \
       emitter.uchar (self.default)

  def str_len (self):
    return 3
```

Listing 4.9: Implementation of class LcCheckbox

There are some things which appear to be magic but simply happen in the base class. The call of the base class constructor makes certain member variables available, namely `self.vname`, `self.writable`, `self.update`, and `self.blink`. They are set to its equivalent constructor parameters. There is one special member variable called `self.last` which is set to `False` by default and indicates whether a component is the last one in a menu line. Generally, we don't need to access this variable. The call to the `self.lctag` method returns the `lctag` with the proper component ID and flags. This works because we have inserted the string '*checkbox*' together with the ID into the `ident_id_map` dictionary and the flag variables are available to the base class.

4.9.3 Extending the parser

Finally, the parser needs to be extended and everything is done. In the file `handler.py` we will find the class `MelxHandler` that handles SAX events upon parsing the XML input file. For our example, we need to handle the beginning `checkbox` tag. When `MelxHandler` is called to handle this start tag, it passes the request to the `do_start_checkbox` method if it can be found. Looking at the already implemented `do_start_integer` method, we can use it as a template for the new component and end up with something like shown in listing 4.10.

```
def do_start_checkbox (self, rname, attrs):
```

```
  lc = cmf.LcCheckbox (atoi (attrs['value']), \
                       *self.def_lc_attrs (attrs))
  self.__cur_lf.append (lc)
```

Listing 4.10: Implementation of do_start_checkbox method

That's all! We only need to create an instance of the new line component class and append it to the component list of the current menu line which is represented through the `self.__cur_lf` variable.

4.9.4 Summary

Now, that we have *mlx* working with our own extension, let's summarize the steps.

1. Extend the language by inserting an element definition into `melx.dtd` and extending the `line-format` element.
2. Write a component class in `cmf.py` which subclasses `Lc` and generates the byte code for the new component.
3. Extend the `MelxHandler` class in `handler.py` with a `do_start_`*`element-name`* method which creates an instance of the new component and inserts the object into the component list of the current menu line (line-format).

So far so good. Now we will probably want to extend the interpreter library or implement a program that can handle the new component beside the others.

4.10 melx.dtd

Here is the content of `melx.dtd` which defines the input language to *mlx*.

```
<!ENTITY % blink-attr "blink (1|0) '0'">
<!ENTITY % edit-attr  "edit  (1|0) '0'">
<!ENTITY % update-attr "update CDATA '0'">
<!ENTITY % vname-attr "vname CDATA #IMPLIED">
<!ENTITY % common-lcomp-attrs
   "%blink-attr; %edit-attr; %update-attr; %vname-attr;">
<!ENTITY % enable-vname-attr "enable-vname CDATA #IMPLIED">

<!ELEMENT melx
   (description, menu+, line-format*) >

<!ELEMENT description
   (delay-to-top, delay-password, delay-help, top-menu) >

```

```
<!ELEMENT delay-help EMPTY >
<!ATTLIST delay-help value CDATA #REQUIRED>

<!ELEMENT delay-password EMPTY >
<!ATTLIST delay-password value CDATA #REQUIRED >

<!ELEMENT delay-to-top EMPTY >
<!ATTLIST delay-to-top value CDATA #REQUIRED >

<!ELEMENT top-menu EMPTY >
<!ATTLIST top-menu ref IDREF #REQUIRED >

<!ELEMENT menu (const-string-line | line)+ >
<!ATTLIST menu id       ID    #REQUIRED
               title    CDATA #IMPLIED
               password CDATA #IMPLIED >

<!ELEMENT line EMPTY >
<!ATTLIST line ref     IDREF #REQUIRED
               submenu IDREF #IMPLIED
               %enable-vname-attr; >

<!ELEMENT const-string-line EMPTY >
<!ATTLIST const-string-line value CDATA   #REQUIRED
                            submenu IDREF #IMPLIED
                            %blink-attr;
                            %enable-vname-attr; >

<!ELEMENT line-format
   (hfill|integer|string|counter|option|switch|
    time|float|date|trigger)+ >
<!ATTLIST line-format id ID #REQUIRED >

<!ELEMENT integer EMPTY >
<!ATTLIST integer
  %common-lcomp-attrs;
  type (dd|ddd|hh|sdd|sddd|
        DDD|DDDD|DDDDD|HHHH|SDDD|SDDDD) #REQUIRED
  value    CDATA #REQUIRED >

<!ELEMENT string EMPTY >
<!ATTLIST string
  %common-lcomp-attrs;
  value    CDATA #REQUIRED>

<!ELEMENT counter EMPTY >
<!ATTLIST counter
  %common-lcomp-attrs;
  type     (integer|float) #REQUIRED
```

```
  value    CDATA #REQUIRED
  min      CDATA #REQUIRED
  max      CDATA #REQUIRED
  step     CDATA #REQUIRED >

<!ELEMENT switch (switch-item+) >
<!ATTLIST switch
  %common-lcomp-attrs;
  on-char  CDATA "*"
  off-char CDATA "." >

<!ELEMENT switch-item EMPTY >
<!ATTLIST switch-item
  info  CDATA #REQUIRED
  value (1|0) #REQUIRED >

<!ELEMENT option (option-item+) >
<!ATTLIST option
  %common-lcomp-attrs;
  default  IDREF #REQUIRED >

<!ELEMENT option-item EMPTY >
<!ATTLIST option-item value CDATA #REQUIRED
                      id    ID    #REQUIRED >

<!ELEMENT time EMPTY >
<!ATTLIST time
  %common-lcomp-attrs;
  hours    CDATA        #REQUIRED
  minutes  CDATA        #REQUIRED
  seconds  CDATA        #REQUIRED
  type     (short|long) "short" >

<!ELEMENT float EMPTY >
<!ATTLIST float
  %common-lcomp-attrs;
  value    CDATA          #REQUIRED
  type     (siif|siiif)   "siif" >

<!ELEMENT date EMPTY >
<!ATTLIST date
  %common-lcomp-attrs;
  day      CDATA        #REQUIRED
  month    CDATA        #REQUIRED
  year     CDATA        #REQUIRED
  type     (short|long) "short" >

<!ELEMENT trigger EMPTY >
<!ATTLIST trigger vname CDATA    #REQUIRED
```

```
                    password CDATA #IMPLIED
116                 %blink-attr; >

<!ELEMENT hfill EMPTY >
<!ATTLIST hfill char  CDATA " "
120                 count CDATA "0" >
```

Listing 4.11: The Melx Data Type Definition

Chapter 5

Implementation

This chapter discusses the implementation of *mexc* and *mlx*, and will require an understanding of the binary menu description in CMF format which is described in section 4.7. The analysis given here is not meant to be a description of each line of the source code, but merely provide enough information to get new programmers picking up the code and making changes to it, and possibly contributing to further development of the project. We will concentrate on the basic concepts, look at the structure of the sources and discuss some tricky lines of code.

5.1 The Menu Interpreter

From the beginning on it has been tried to build a layered design of the code to make it clear and easy to maintain, although there have been made compromises, as the code should run under restricted circumstances. Figure 5.1 outlines the dependencies between the source files of *mexc*.

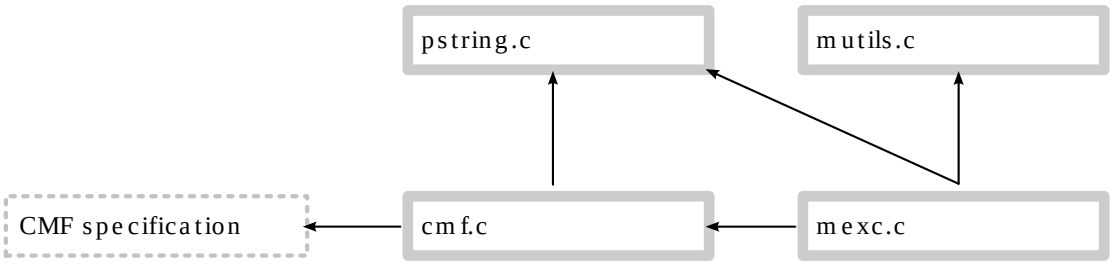

Figure 5.1: Source code dependency of *mexc*

Each source file implements a set of functionality that is used by the others. The CMF specification is the description of the structure of a binary menu image. As shown, `cmf.c` relieves `mexc.c` of having to know the details of the menu description.

Now, let's step through the source files and show what functionality they provide. All file names in this section are relative to the source code directory for *mexc*. See Appendix B.

5.1.1 The `cmf` Sub-Library

To abstract from the fact that a CMF formated binary menu image is actually an array of bytes, and to provide a more comfortable way for using a concrete CMF structure, general purpose code has been implemented in `cmf.c` and `cmf.h`. It has the advantage that changes in the structure needs to be reflected only to this sub-library and don't necessarily affect code using the `cmf` module.

Talking about `cmf.c` and `cmf.h` as a library isn't correct, as the code isn't released as such. However, it could. The present text will refer to the `cmf` code as a sub-library to indicate that it actually is a library, but as part of another library, namely *mexc*.

The header file `cmf.h` defines the public API of this sub-library. It consists out of numerous constants, C macros, and function declarations as well as of data structures that reflect parts of the menu byte code. Code that uses the `cmf` module should never access the byte code directly but use the `cmf` API.

All public functions of the `cmf` API begin with the prefix '`cmf_`'. As the sub-library accesses the binary menu image in read-only mode, the word '`get`' would be redundant and therefore has been omitted in function names. The exported function can be categorized as follows.

- Both functions, `cmf_init` and `cmf_init_ram`, are special in the form that they are used for initialization of the library and RAM variables defined through CMF. They operate on the whole CMF structure. Another function that operates on the whole structure is `cmf_first_mtable` which returns a pointer to the top-level menu table.
- Functions that begin with the `cmf_mtable_` prefix are related to a given menu table only and expect such an argument. They need to be passed a pointer to a menu table in the byte code.
- Further, functions starting with `cmf_lcomp_` concentrate on a line component and therefore expect as an argument a pointer such a structure in the byte code.
- There are a few more functions with start with the prefix `cmf_`*comp-name* and are meant to be used with the appropriate line components.

These functions are accompanied by the following set of C macros:

- Macros starting with the prefix `LDTAG_*` are meant to retrieve information from the line descriptor tag associated with each menu line. They need to be passed the value of the tag.
- Macros starting with the prefix `LCTAG_*` are meant to retrieve information from the line component tag associated with each line component. They need to be passed the value of the tag.

Further the following byte code related definitions are provided by `cmf.h`:

- `LC_TYPE_*` definitions provide names for the numerical IDs of the currently understood line components.

- `CMF_MAJOR_VERSION` and `CMF_MINOR_VERSION` are dedicated to reflect the supported CMF byte code version.

`cmf.h` also defines a structure for each supported line component. Such a `struct` is introduced with a name `lc_`*comp-name*`_t`. These structures reflect the byte code structures as they have been defined in section 4.7. The advantage of them is that they can be "laid over" the menu byte code and provide a very clear way to access it. As the type of a line component is determined at runtime, the use of the `union` called `lc_t`, which includes all line component structures, has proven to be comfortable and simplifies the code at the same time. An example follows.

```
addr_t mtable;
addr_t mline;
lc_t * lc;

/* menu_code is the array holding the menu byte code */
if (cmf_init (menu_code))
  return; /* initialization failed */

/* get the top-level menu */
mtable = cmf_first_mtable ();
/* there must be at least on menu table */
assert (mtable != NULL);

/* get the menu's first line */
mline = cmf_mtable_mline (mtable);
/* each menu must have at least one menu line */
assert (mline != NULL);

/* get the line's first component */
lc = (lc_t *)cmf_mline_lcomp (mline);
/* a menu line must have at least one line comp */
assert (lc != NULL);

if (LCTAG_TYPE (lc->common.lctag) == LC_TYPE_TIME_SHORT)
  printf ("%02d:%02d\n", lc->lc_time.hours, lc->lc_time.minutes);
else if (LCTAG_TYPE (lc->common.lctag) == LC_TYPE_UCHAR_DD)
  printf ("Default value: %d\n", lc->lc_uchar.def);
```

Listing 5.1: Example for accessing a line component

The advantage of using the `lc_t` union is that we don't need to declare pointers for all possible line components, but leave the dirty work to the compiler. It should be noted, that the returned pointers of the `cmf` routines are pointers into the binary menu image. If that image is stored in a read-only memory region, we must reference the pointers for reading only, not writing.

The implementation of the sub-library is fairly straight forward and well documented in the source code itself, but let's look at some points which may not be obvious immediately.

- As the individual line components in the byte code are not separated by a delimiter byte, but merely following each other, it is necessary to know the size of the byte code for each component to be able to jump from one to the next. However, some components consists of a variable number of bytes and their exact sizes must be determined at runtime.

  The function `cmf_lcomp_length`, which returns the size of the byte code for a given line component, uses a lookup table with the appropriate sizes. Whenever a field in this table has the value of zero, the component is disassembled and its size computed.

- As explained in section 4.7.4, following the `ldtag`, which is associated with each menu line, there is a variable number of fields. These fields' existence is determined by flags set in the `ldtag`. To efficiently access these fields, or those that follow, the `ml_submenu_ofs` lookup table has been defined. It holds the offsets from the `ldtag`, not including it, up to the optional submenu field. Considering the first three flags of the `ldtag` as a number, they can be used to index the lookup table.

  This method relies on the order of the flags and their association with the fields following `ldtag`. However, CMF has been designed to provide this possibility of determining the offsets and will not change this in future versions. Thus, this method, isn't a hack[1] as it may look like at the first glance.

- `cmf_init_ram` initializes all memory regions specified by the menu byte code. Therefore, it must traverse the menu table tree and iterate over all line components. This is accomplished by traversing the tree structure recursively in depth-first order. When only a limited amount of memory is available, recursion is critical. However, the implementation of `cmf_init_ram` pays attention to not allocating too much space on the stack, and, in general, the depth of a menu usually isn't to deep to cause serious problems.

5.1.2 Handling Pascal-style Strings

The CMF specification introduces strings that are not zero-terminated, as used in the C language, but preceded with a length byte. `pstring.c` implements a small set of functions for handling Pascal-style strings. They all start with the prefix `pstr_`.

`PSTR_LEN` and `PSTR_STR`, both being C macros, simply return the length and the pointer to the first character of strings respectively. Although their definition is trivial and doesn't save a lot of work, there are two reasons why to use them:

a) Firstly, they introduce names for the operations. Further, they also indicate on which object they operate. This leads to more readable code.

b) Should there be changes to the data type of the length value, for example increasing its size to a two-byte integer, only changes to the macros would be necessary.

Both macros must be passed a pointer to a Pascal-style string.

[1] We refer to the word "hack" by its original meaning as stated in The Jargon File [12].

`pstr_copy` is equivalent to `strcpy`, but operates on Pascal-style strings. It simply copies its first argument to the place where the second argument points to. Having a length byte, which can be directly accessed, usage of the `memcpy` or `memmove` functions[2] would be ideal and result in fast operation. However, as the code is targeted at embedded systems where such system functions probably won't be available, this routine copies the string one byte after another.

`pstr_to_r_cstr` ("Pascal to right-aligned C string") is a service routine to copy the characters of a Pascal-style string to a buffer and zero-terminate them. Furthermore, the resulting string will be right-aligned within a width that the caller must specify along with the character to use for padding. As with the previous function, it was decided against the use of system services like `memcpy` or `memmove`.

5.1.3 Utility Functions

General purpose utility functions has been put into the file `mutils.c`. Mainly, it implements "number to string" conversion functions.

```
uchar *utorstr (uchar *buffer, schar w, uchar hex, uchar fill, uint2 n);
```

`utorstr` converts the unsigned two-byte integer `n` into a right-aligned C string representation. `buffer`, a pointer to the memory region that will receive the string, must be at least `w` bytes long. `w` defines the width in which the resulting string will be right-aligned. Padding is done using the `fill` character. With `hex` being non-zero, the number is outputted in hexadecimal format. The returned pointer is `buffer`.

The implementation of this functions uses a trick on the hexadecimal output. For example, if we should convert a 12 to its hexadecimal representation, we could do the following consideration: *as 12 is greater than 9, subtract 10 from 12, and add the result to the character code of A to finally receive the character C.* On the other hand we could also think like this: *as 12 is grater than 9, add the ASCII code of the digit 7 to it, to receive the character C.* The second method is obviously faster, but it relies on the fact that the ASCII code of the digit 7 is smaller by 10 than the ASCII code of the uppercase letter A.

```
uchar *storstr (uchar *buffer, schar w, uchar fs, uchar fill, sint2 n);
```

`storstr` provides essentially the same service for signed numbers like `utorstr` for unsigned numbers. There is one exception. `storstr` cannot format its output in hexadecimal notation. Instead it can be told, to force putting a sign in front of the number. When `fs` is non-zero, also positive numbers will be accompanied by the sign.

```
uchar *ftorstr (uchar *buffer, schar w, uchar fill, float f);
```

[2] See `man memcpy` and `man memmove`.

`ftorstr` converts a float number to a C string. The output format uses a `sii.f` template, with `s` being the sign, `i` the digits before the decimal point, and `f` a digit behind the decimal point. Depending on the `w` parameter to this function, the `i` in the template can grow. The `fs` parameter controls whether a sign will be outputted for positive floats.

The implementation uses some arithmetic and type conversion. It rounds the number to one digit after the decimal point to display numbers like 42.29999999 as 42.3.

```
uchar get_switch_bit (uchar *mask, uchar bit);
uchar flip_switch_bit (uchar *mask, uchar bit);
```

`get_switch_bit` and `flip_switch_bit` provides access to bits of an array as introduced by a switch line component. They expect a byte array and an index which addresses one bit within the byte array. Figure 5.2 shows a switch bitmask along with the indexes of the bits. The

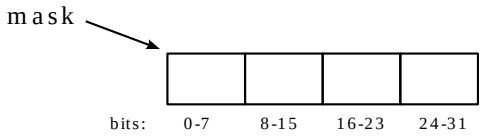

Figure 5.2: Bitmask of a *switch* line component

implementation of both functions uses bit arithmetic operators. In case of `flip_switch_bit`, it toggles a bit within the mask before it returns the bit's new value. `get_switch_bit` just returns the current value of a bit without altering the bitmask.

5.1.4 The Engine

The file `mexc.c` implements the core of the library and can be considered as its engine, most functions of it are documented by using doxygen style comments. The module is dependent on functionality provided with the already discussed files and the functions to be implemented by an application programmer as discussed in section 3.7.1.

5.1.4.1 Global Data

At the beginning of the source file there are global data definitions which we will discuss next. However, why is there global data at all, when good software design teaches us to avoid it. Of course, it would be possible to avoid globals by allocating it on the stack and pass it as function parameters to the code that operates on it. However, this would lead to the following disadvantages.

- The code would get worse readable, as a lot of parameters would get passed around only to be outreached to other functions further down the execution path.

- The size of needed stack space would considerably grow as arguments to functions are put on the stack. As the code is targeted at low memory systems, this is an essential consideration.

Usage of globals in this project has *not* been established to serve as an optimization, although it is more efficient to access data directly than referencing it with a pointer. Arguing with the advantage of efficiency would definitely be a "premature optimization"[3]. All global variable names start with the prefix `g_` to indicate their globality and are declared `static` to restrict their visibility to `mexc.c`. The following paragraphs discuss some important global variables.

Display Context

As CMF doesn't define references to parent menus and because *mexc* restores the state of the display, termed *display context*, when the user jumps back from a submenu, the context must be stored before the user dives into a submenu. This is done in the `g_dc` array accompanied by `g_cur_dc_idx`. The display context array is used as a stack. `g_cur_dc_idx` indexes the currently active entry. Therefore, `g_dc[g_cur_dc_idx]` was a frequently used statement in the code, which has been replaced by the `g_cur_dc` pointer to reduce code size. Each time `g_cur_dc_idx` is altered, the code makes sure that `g_cur_dc` points to the current table entry. Therefore we can assume that '`g_cur_dc == &g_dc[g_cur_dc_idx]`' is always true.

A display context itself, an entry in the `g_dc` array, is a structure of type `disp_context_t`. Having the three pointers from a display context, which is shown below in listing 5.2, the display can be restored to the state before a submenu was entered.

```
53 typedef struct {
54   addr_t mtable;    /* current menu table */
55   addr_t cur_mline; /* current menu line */
56   addr_t top_mline; /* menu line at the top the screen */
57 } disp_context_t;
```

Listing 5.2: mexc.c / 53–57 (the display context)

It is important to consider, that the implementation cannot rely on dynamic memory allocation, and therefore, a fixed amount of entries for the `g_dc` array is allocated in the program's data section. The size of the array is finally determined by number substituted by the preprocessor definition `MEXC_MAX_DISP_CONTEXT_DEPTH`.

Blinking Components

Each time after the code went through all the menu lines to be displayed – this is in `disp_draw_mlines` – the `g_do_blink` global variable gets set to one of the following values with the appropriate meaning.

[3]The statement "Premature optimization is the root of all evil." originates from Tony Hoare and has become famous by Donald Knuth. It is a frequently used idiom in software development to darkly hint at problems which results from optimizing at a too early stage. See also [13].

- A zero denotes that currently there is no component on the screen that should be blinking.
- `DO_BLINK_DRAW` indicates that there is at least one component on the screen which should blink, and that the next time the component will be drawn it should be visible.
- `DO_BLINK_ERASE` indicates that there is at least one component on the screen which should blink, and that the next time the component will be drawn it should not be visible.

Editing Components

Whenever the "edit state" is entered – this is when the user edits a component and the horizontal cursor is visible on the screen – the global variable `g_editing` is set to non-zero, otherwise it is zero.

Visibility of the Cursor

When the LCD cursor, a blinking black box, is currently set to be visible, then the global variable `g_cursor_visible` is set to non-zero, otherwise to zero. Setting the cursor's visibility should be done by using the two macros `PUT_CURSOR_ON` and `PUT_CURSOR_OFF` to ensure not forgetting to set also the global variable.

```
102 #define PUT_CURSOR_OFF() do {                    \
103                           g_cursor_visible = 0; \
104                           mlcd_cu_off ();       \
105                          } while (0)
106 #define PUT_CURSOR_ON()  do {                    \
107                           g_cursor_visible = 1; \
108                           mlcd_cu_on ();        \
109                          } while (0)
```

Listing 5.3: mexc.c / 102–109 (cursor visibility macros)

Both macros expand into a `do .. while` loop whose body will be executed exactly once. This ensures that the two instructions inside the loop always stay together, even in a situation like shown in listing 5.4.

```
if (foo)
  PUT_CURSOR_OFF ();
else
  PUT_CURSOR_ON ();
```

Listing 5.4: Example for using cursor visiblity macros

Already at compilation time it is clear, through the usage of the constant 0, that the loop will never repeat, modern compilers will not produce code that reflects a loop, and thus no overhead will be caused.

Automatic Update

As specified in CMF, individual line components can request to be periodically updated, and even specify the number of seconds after which their current value should be redrawn. Currently, *mexc* redraws all menu lines on the screen, when the `g_update_delay` global variable is set to non-zero. Strictly speaking, it redraws the lines after that many seconds as specified by `g_update_delay`. This variable gets set to the minimal requested update period while the code parses all menu lines to be displayed. When there are no components on the screen that should be periodically updated, `g_update_delay` is set to zero. Thus, it is ensured that *mexc* will not redraw the menu lines more often than necessary.

5.1.4.2 Initialization

Initialization of the *mexc* library is done with a call to `mexc_init`, its public interface is discussed in section 3.7.4.2. As first, the routine initializes the `cmf` sub-library with a call to `cmf_init`. This makes the sub-system available upon successful return. In the other case – when the sub-system initialization fails – `mexc_init` simply returns with the error code from the sub-library as shown in listing 5.6.

```
179 rv = cmf_init (mcode);
180 if (rv)
181   return rv;
```

Listing 5.5: mexc.c / 179–181 (initializing cmf sub–library)

After a few global variables have been set, the dynamic variables located in RAM, which are associated with and defined in the given menu description, are initialized by `cmf_init_ram`. From this point on, the application and the library can access those variable.

```
188 /* initialize the ram-variables */
189 cmf_init_ram (cmf_first_mtable (), ram, default_cb_handler);
```

Listing 5.6: mexc.c 188–189 (initializing RAM variables)

Thereafter, the number of lines and columns to be used by *mexc* on the screen is fetched by calling the `get_mlcd_lines` and `get_mlcd_cols` functions and stored in `g_lcd_lines` and `g_lcd_cols` respectively. It is important not to call any other display related routines at this point, as the application programmer should not be restricted to having to initialize the display before calling `mexc_init`.

Finally, global variables that represent the display context are properly initialized and the display context itself is set to be invalid as there is still no menu table open.

5.1.4.3 Opening Menu Tables

Having introduced the display context array `g_dc` earlier, now we will see where this array is used. There are three functions accompanied by a helper function called `dc_open_mtable` to operate on `g_dc`.

The `dc_open_mtable` function initializes the current display context entry in `g_dc` with a given menu table. As already mentioned, this entry can be referenced with the `g_cur_dc` pointer. After having initialized the context's variables, the whole screen is redrawn by using the public function `mexc_redraw`.

The `goto_top_menu` function is responsible for jumping out of any submenus, thereby deleting all saved display contexts, and open the top-level menu table by calling `dc_open_mtable` with the appropriate menu table as its argument. However, when the currently open menu table is already the top-level table and the table's first menu line is at the top of the screen, no redrawing of the screen has to be done, as this is the initial state. To force redrawing, a non-zero value needs to be passed to `goto_top_menu`. As a side-effect this routine puts the cursor off, if it is visible.

The `goto_submenu` function expects a menu line as its argument, not a menu table, as the optional submenu password is stored as part of a menu line. After assuring that the given menu line is associated with a menu table and optionally verifying the password, the submenu is entered. Thereby a new display context from the `g_dc` array is activated and initialized by calling `dc_open_mtable` as shown in listing 5.7.

```
1099 if (g_cur_dc_idx < (MEXC_MAX_DISP_CONTEXT_DEPTH-1))
1100   {
1101     g_cur_dc_idx++;
1102     g_cur_dc++;
1103     dc_open_mtable(cmf_mline_submenu (mline));
1104   }
```

Listing 5.7: mexc.c / 1099–1104 (Opening a new display context)

It is important that the code assures not to increase `g_cur_dc_idx` to a value equal to or greater than `MEXC_MAX_DISP_CONTEXT_DEPTH` to avoid corruption of global data by writing beyond the end of the `g_dc` array.

The `go_dc_back` function is the counterpart to the previously discussed one. It throws away the current display context by decrementing `g_cur_dc_idx` and `g_cur_dc`. Thereby, the lastly used context becomes active again and is finally put on the screen. Listing 5.8 shows how easy it is to restore an old display context due to used data type, a stack[4].

[4] `g_dc` is an ordinary array of a fixed size, but it's the way the array is used that makes it being regarded as a stack.

```
1020 if (g_cur_dc_idx > 0)
1021   {
1022     g_cur_dc_idx--;
1023     g_cur_dc--;
1024     mexc_redraw ();
1025   }
```

Listing 5.8: mexc.c / 1120–1125 (Restoring an old display context)

5.1.4.4 Thin Layer over `cmf`

`mexc.c` implements three functions on top of the `cmf` sub-library to provide the same functionality but respect the "enable value"of a menu line in a special manner. This value and its purpose is described in section 4.7.4.

Because *mexc* is targeted at small displays, it simply ignores disabled menu lines by not drawing them on the screen. Whenever the next or previous menu line is fetched via `mexc_mline_next` or `mexc_mline_prev` respectively, the returned menu line, if any, is guaranteed to be enabled at the time of the function call. The third function, `mexc_mtable_mline`, fetches the first enabled menu line of a menu table.

With these three routines the rest of code in `mexc.c` doesn't have to know about dynamic menu lines. In fact, when dynamic lines were introduced, the implementation of the three routines were added and all calls to `cmf_mline_next`, `cmf_mline_prev`, and `cmf_mtable_mline` were replaced with calls to their `mexc_` equivalents. No more work has been necessary to extend the library.

5.1.4.5 Getting Key Presses

Key presses are fetched with the `mfpkey_get` routine which is described in section 3.7.1.2. When there are no key presses that function immediately returns `MFPKEY_NONE`. However, it would be desirable to have a function that waits for a key press and then returns it to the caller. This is achieved with the `get_key` function. When its argument is not zero, this function returns `MFPKEY_NONE` after *n* seconds of no keyboard activity.

The implementation of `get_key` gives a perfect example of programming for embedded systems. Listing 5.9 shows an easy to understand implementation, however, it requires `howlong` to be a 32-bit integer to make the code work. But this requirement cannot be accepted, as the code is targeted at platforms which do not have to support such a data type.

```
do {
  key = mfpkey_get ();

  if (MFPKEY_NONE != key)
    break;
```

```
  else
    {
      msleep (MEXC_GET_KEY_DELAY);

      howlong += MEXC_GET_KEY_DELAY;
      if (timeout && howlong >= timeout*1000)
        break;
    }
} while (1);
```

Listing 5.9: Possible implementation of get_key loop

The implementation in `mexc.c` is somewhat trickier, but needs only two one-byte integers to achieve the same result. Figure 5.10 shows how it is done. As we stated, `howoften` is a one-byte integer and this can cause a problem when `MEXC_GET_KEY_DELAY` is smaller than 4, as the division would result in a value greater than 255 (modern compilers will compute the division at compilation time as both operands are constants). This problem can be simply solved by making `howoften` a two-byte integer. However, it is very unlikely that a smaller value than 4 will be specified for the definition. Further on, modern compilers like `gcc` will warn us when there is a problem with the comparison.

```
620 do {
621   key = mfpkey_get ();
622
623   if (MFPKEY_NONE != key)
624     break;
625   else
626     {
627       msleep (MEXC_GET_KEY_DELAY);
628
629       if (timeout && ++howoften == 1000/MEXC_GET_KEY_DELAY)
630         {
631           howoften = 0;
632           howlong++;
633         }
634
635       if (timeout && howlong == timeout)
636         break; /* key is MFPKEY_NONE */
637     }
638 } while (1);
```

Listing 5.10: mexc.c 620–638 (Implementation of get_key loop)

5.1.4.6 Displaying Matters

When the whole screen needs to be (re-)drawn, calling `mexc_redraw` is appropriate. This command divides into two parts, drawing the header line with `disp_draw_hdr` and drawing the menu

lines with `disp_draw_mlines`. Both functions operate on the current display context available through `g_cur_dc`.

`disp_draw_hdr`'s implementation consists of drawing the title of the currently opened menu table and drawing some information on the screen as described in section 3.3. The latter is realized with the `disp_draw_hdr_info` function which is called also from other routines. The core point of the separation is to reflect the need for redrawing the information fields, but not the title. The implementation of `disp_draw_hdr_info` consists mainly of assembling a string which reflects the state of the current menu line, and finally drawing it into the upper right corner of the screen.

`disp_draw_mlines` is responsible for drawing as many menu lines on the screen as possible. As a secondary target, this function updates some global variables; we will look at this in a moment. Although the primary target is drawing, no calls to drawing routines can be found inside the function's body. To draw a menu line, it is necessary to disassemble it and draw the individual parts. This complicated task is accomplished by the `disp_draw_mline` function which is examined in a short while. Thus, `disp_draw_mlines` concentrates on its secondary target, and directs the drawing requests to the menu line parsing routine. While the function iterates over the menu lines to be drawn on the screen, it collects some information and sets three global variables to reflect the requests defined in the menu. Listings 5.11 and 5.12 show the realization.

- Even before the iteration over the lines, `g_update_delay` is set to zero, because we initially assume there are no line components on the screen to be automatically updated. When the individual menu lines are disassembled in `disp_draw_mline` the global gets appropriately set from the values of the individual line components.
- Further, while iterating, `disp_draw_mline`'s return value is evaluated, which tells whether the passed menu line contains blinkable line components. If the return value is non-zero a local variable `do_blink` is set to `DO_BLINK_DRAW`. This local variable is evaluated after the loop ends.
- `g_cur_lcd_line` gets set to the index of the line on the display in which the current menu line is displayed. This global is later used to determine whether scrolling the menu table is necessary when navigating through it.

```
688 g_update_delay = 0; /* assume we do not need to update */
689
690 mline = g_cur_dc->top_mline;
691 for (lineno = FIRST_MENU_LINE_IDX; lineno < g_lcd_lines; lineno++)
692   {
693     /* update g_cur_lcd_line */
694     if (mline == g_cur_dc->cur_mline)
695         g_cur_lcd_line = lineno;
696
697     if (disp_draw_mline (mline, lineno))
698         do_blink = DO_BLINK_DRAW;
699
700     if (mline)
```

```
701         mline = mexc_mline_next (mline);
702   }
```

Listing 5.11: mexc.c / 688–702 (determining whether to blink)

The evaluation of the `do_blink` local variable is given in listing 5.12. It simply sets `g_do_blink` to zero if there are no components on the screen that should blink. If there are such components, the value gets toggled from `DO_BLINK_DRAW` to `DO_BLINK_ERASE` and vice versa.

```
709 if (!(g_do_blink && do_blink))
710   g_do_blink = do_blink;
711
712 if (g_do_blink)
713   g_do_blink = ~g_do_blink;
```

Listing 5.12: mexc.c / 709–713 (setting g_do_blink)

The already mentioned `disp_draw_mline` function is quite long because it implements the rendering for each line component to the screen. Though, there is little to say about it. To be efficient, this routine allocates a buffer of length `g_lcd_cols` and renders all components of the given menu line into it, mainly using routines discussed in sections 5.1.3 and 5.1.2. Finally, after all components are rendered in the buffer, it is written all in once onto the screen by calling `mlcd_wrstrxy`. If the passed menu line pointer is NULL, then `mlcd_clrln` is used to wipe out everything that was before in the display line index by the passed `lineno` argument. As a side effect this function sets the `g_update_delay` global variable. Listing 5.13 shows the code. `update_val` is the minimal update interval beside zero, that was extracted from the visited line components. When all components of the given menu line have an update value of zero, `set_update` is zero, otherwise non-zero. The global `g_update_delay` gets set to a new update value only when the new value isn't zero and is smaller than `g_update_delay`. Because `disp_draw_mlines` sets the global to zero, the variable will be zero if there are no components on the screen which should be periodically updated. On the other hand, when there are such components, the variable will define the minimal update interval among components to be updated.

```
976 if (set_update && (!g_update_delay || (g_update_delay > update_val)))
977   g_update_delay = update_val;
```

Listing 5.13: mexc.c / 976–977 (setting g_update_delay)

5.1.4.7 Editing Line Components

Almost over a half of the code in `mexc.c` deals with editing line components, however, no great complexity hides behind it.

The function `edit_cur_mline`, which is called when the user requests to do so, is the first point to look at. After entering the function, the global variable `g_editing` is set to non-zero to

indicate the current state of the library. Before `edit_cur_mline` returns to the caller, this global is set to zero again. A loop over the line components of the current menu line is started. Whenever an editable line component is found, a specific function which can handle the component's type is called and takes control over editing the specified component. We will continue to refer to these special routines as "edit functions". After an edit function returns, the callback handler associated with the component gets called, if there is one, and the loop continues until it reaches the last component. However, there is one exception to the described flow of execution! When an edit function returns a non-zero value, looping over the rest of the components is aborted, and the `goto_top_menu` functions is called before `edit_cur_mline` returns. To understand the reason of this behavior we need to know what it means when an edit function returns a non-zero value. It simply signalizes that a timeout occurred and that the library should return to the top-level menu table.

There are a couple of edit routines which produce no side effects on global data except of putting the cursor *on* and *off* using `PUT_CURSOR_ON` and `PUT_CURSOR_OFF`. Their implementation is straight forward and should not be hard to understand.

5.1.4.8 The Main Loop

`mexc_loop` is the library's main loop. Once started, the routine returns only when it fetches the `MFPKEY_QUIT_MEXC_LOOP` key press. The function `mexc_init` must have already been called when `mexc_loop` starts execution. At this point also the display must have been initialized.

The first steps of the function are to use `PUT_CURSOR_OFF` to hide the cursor, and to initialize the current display context with the top-level menu table using the discussed `dc_open_mtable` routine. Then the loop itself is started. Inside it, a key press is fetched, and the appropriated action is carried out. Fetching a key press is done using `get_key` which takes a number of seconds to wait as long as no key press occurs. The interesting part of the main loop is determining the value that is to be passed to `get_key`, and handling the timeout. As listing 5.14 shows, `delay` is set to the minimum of three values. Zero is handled in a special way, all three value have to be zero to get `delay` set to 0.

```
223 delay = g_prolog->delay_to_top;
224 if (g_update_delay)
225   delay = delay ? MIN(delay, g_update_delay) : g_update_delay;
226 if (g_do_blink)
227   delay = delay ? MIN(delay, MEXC_BLINK_INTERVAL) :
          MEXC_BLINK_INTERVAL;
```

Listing 5.14: mexc.c / 223–227 (Determining timeout value)

When `get_key` returns with `MFPKEY_NONE`, a timeout occurred. The function waited for `delay` many seconds but got no key press to report. Handling the timeout event is simple due to the preparations. When it's time to go to the top-level menu table, the code calls `goto_top_menu`. Otherwise, when there are components to blink or to be updated, the the menu lines are redrawn using `disp_draw_mlines`.

5.2 The Menu Compiler

mlx, the menu compiler, is written in two layers. The first processes the compiler's input, an XML document, and creates a data structure, a tree, which is need when the second layer computes the output, a binary data described by CMF. These two layers are controlled by a small application which provides a command line interface.

Due to the nature of object orientation, in which manner *mlx* is written, the data structure created by the first layer is actually represented as the second layer. However, this doesn't change the design. In the following discussion we will look at the two layers and also the small application controlling the steps necessary to produce the program's output. We won't step into each detail, but provide enough information to get an idea of how to read the *mlx* source code.

All files names in this section are relative to the source code directory of *mlx*. See Appendix B.

5.2.1 Byte Code Generating Layer

As it will help to understand the first layer, let's discuss the second one first. This layer is completely implemented in the file `cmf.py` and consists of 16 classes. While three classes are purely used to implement helper objects, the rest of them reflects all parts of a menu hierarchy. Let's introduce the helper classes first.

`CmfError` serves to report errors. It subclasses `Exception` as suggested in [14]. Whenever an error within the `cmf` module occurs, an exception of this type is raised. Catching this exception is up to the caller of the appropriate `cmf` code.

`ByteListEmitter` implements a byte code generator for low level data types, namely integers, floats, and strings. It is this class, that is responsible for producing Pascal-style strings. An object of this class is initialized with one boolean value and tells whether the produced byte code for the data types is to be in big- or little-endian. Listing 5.15 shows how this class can be used and what it produces.

```
~mlx% python
>>> import cmf
>>> be = cmf.ByteListEmitter(True)   # big-endian
>>> le = cmf.ByteListEmitter(False)  # little-endian
>>> be.int2(-1234)
[251, 46]
>>> le.int2(-1234)
[46, 251]
>>> be.uint2(0xFEFF)
[254, 255]
>>> le.uint2(0xFEFF)
[255, 254]
>>> be.string("hello, world")
[12, 104, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100]
```

```
>>> le.string("hello, world")
[12, 104, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100]
>>> be.float4(12.4)
[65, 70, 102, 102]
>>> le.float4(12.4)
[102, 102, 70, 65]
```

Listing 5.15: Using ByteListEmitter class

`MemAllocator` isn't complex as it may seem at the first glance. An object of this class abstractly represents a continuous memory region in which variables have to be stored. The user of an object of this class firstly feeds the allocator with variables to be put into the memory region, this is also called registering variables, and finally lets the object compute the variables' offsets within the region. Due to this approach, optimization to the layout of the memory addresses can be applied. Both algorithms, non-optimizing and optimizing, for computing the final offsets are implemented in the `alloc` method. The optimization algorithm is quite easy, although its realization is somewhat tricky. Listing 5.16 outlines the basic idea.

```
repeat as long as there are variables to process:
  * get next variable to process
  * is there a memory gap into which the variable fits?
    -> yes:
      * put the variable into the gap
      * remove this gap from the 'gaps list'
    -> no:
      * allocate space for the variable at an aligned
        offset at the current end of the memory
      * is there now a new gap due to the alignment?
        -> yes:
          * put the gap into the 'gaps list'
```

Listing 5.16: Memory optimization algorithm

The rest of the classes is used to create the hierarchy of the menu and produce the proper byte code. Such as `ByteListEmitter` can produce byte code for basic data types, the classes to be introduced can produce the byte code for more elaborate data like a line component or a menu line. Thereby, the individual parts of the menu hierarchy are covered by individual classes.

`Cmf` represents the whole menu. It is the class that outputs a menu description as specified in section 4.7. The complexity is divided and partly shifted down towards other classes. Thus, the implementation is pretty easy. It only knows how to produce the byte code of an CMF prolog as described in 4.7.2 and 4.7.3, and directs the rest of the work to the only menu table it references, a `Menu` object which represents the top-level menu table.

When the `Cmf.bc` method is called, the caller requests the byte code for the menu and the machinery to produce it is started. It is done in two cycles.

Firstly, a `MemAllocator` object is created, and the top-level menu table's `alloc_addrs` method is called. This will cause a complete population of the allocator object with variable requests. Then the addresses are computed by the object's `alloc` method.

```
303 self.menu.alloc_addrs (alloc)
304 alloc.alloc ()
```

Listing 5.17: cmf.py / 303 –304 (Generation of variable addresses)

The second step consists of producing the byte code itself. This is simply done by collecting the CMF prolog's parts into a list using an `ByteListEmitter` object and afterwards calling the top-level menu table's `bc` method. The returned result is appended to the produced prolog.

Finally `Cmf.bc` returns with the byte code along with the allocator object, as beside the addresses, this object holds also other valuable information that *mlx* will write into one of the generated files.

`Menu` is a more complicated class and presents a single menu table. It holds a list of menu lines, a title, and an optional password for the menu table it represents. As a menu table itself has no variables to allocate, its `alloc_addrs` method simply passes the request further down to each menu line of the table.

The complexity of the `Menu.bc` method originates from the fact that references – references are offsets within the byte code – are needed at a level at which even the length of the individual parts isn't known. Thus, the first thing to do in this method is to compute the length of all menu tables by calling `bc_len` on each. Then the byte code itself is generated according to the CMF specification. As far as a menu table is concerned, this mainly consists of producing the menu table's title, its menu lines, and optionally outputting padding zero bytes to align the menu lines on non-even offsets. The byte code of the menu lines is simply requested from the `MenuLine` class objects. Then, when the `Menu` object is the top-level menu table, the `bc` method of the other menu tables is called, and the returned byte code is appended to the already generated list of bytes.

The method `bc_len` simply accumulates the size of all parts of a menu table. This includes the table's title, the optional padding zero bytes, and the length of all menu lines inside a menu table.

`MenuLine`, a class for representing the container for line components, holds a list of those and a reference to an optional menu table, also referred to as submenu in this context. If a menu line should be dynamically disabled/enabled, its `enable_vname` member variable has a name for the variable to be allocated using a `MemAllocator`.

The `alloc_addrs` method registers the optional "enable" variable, if there is one. Then it passes the request to all line components of the menu line and also to the optional submenu.

The next method in the process of the byte code generation is the invocation of the `bc_len` method. It gets called by `Menu.bc_len`. The length of the byte code for a menu line

is simply computed by accumulating the length of all components of a line, this includes the fields shown in figure 4.3 on page 51 and the line components itself. The length of the line components is retrieved as the length of the list of bytes which each component object produces. Thus, at this point, the byte code for each line component is generated. To avoid generating the byte code again, later in the `bc` method, it is stored in the `__bc_len_cache` variable. This works, because line components don't need any references which are available only at the time of a call to the `bc` method.

Due to the work already done by the `bc_len` method, `MenuLine.bc`'s implementation is really simple. All it does is generating the bytes for fields shown in figure 4.3 and appending it to the already generated byte code stored in the `__bc_len_cache` member variable. This is the byte code for the line components of the line.

`Lc` is the base class for those implementing the individual types of line components. This class is not meant to be instantiated. However, as the Python language does support neither abstract classes like C++ nor interfaces like Java, `Lc` is a proper class providing just empty versions of the methods to be overwritten by subclasses. In fact, these method versions raise an exception whenever they are called, thus indicating the abstractness. Subclasses of `Lc` must overwrite three methods, namely `alloc_addrs`, `bc`, and `str_len`. Their meanings are explained in the documentation string of the `Lc` class and in section 4.9.2.

The advantage of `Lc` is to bundle common functionality of all line components to one class. The `lctag` method, has all necessary information to produce the `lctag` byte as introduced in section 4.7.5. The `ident_id_map` dictionary brings a little bit of clarity into the jungle of the IDs for line components and enables callers of the `lctag` method to pass a name instead of a numerical value. Thus, the numbers are concentrated at one place in the code and are not bound to a concrete implementation of a line component. This allows reasonable management to the logical order of line components and allows to optimize `switch` constructs in the C source code of the *mexc* library.

What we haven't mentioned yet, is the way a `Cmf` object with a menu hierarchy is created. This is the job of *mlx*'s first layer which is explained next.

5.2.2 Input Processing Layer

Processing the input of the *mlx* compiler is encapsulated in the file `handler.py` which provides the class `MelxHandler`. However, the files `nonvalhandler.py` and `valhandler.py` construct a thin layer over the implementation of `MelxHandler`, each on its own. Both files provide classes which are subclassed from `MelxHandler` and another class. Figure 5.3 shows an inheritance diagram. Depending on the `--dont-validate` option to *mlx*, only one of the subclassed handler classes is used and determines the functionality of the XML parser. As its name suggests `ValMelxHandler` validates the input document against a referenced DTD, while `NonValMelxHandler` doesn't. It may be surprising that both classes have been implemented in separate files and not put together into one. This has been done to avoid loading of modules

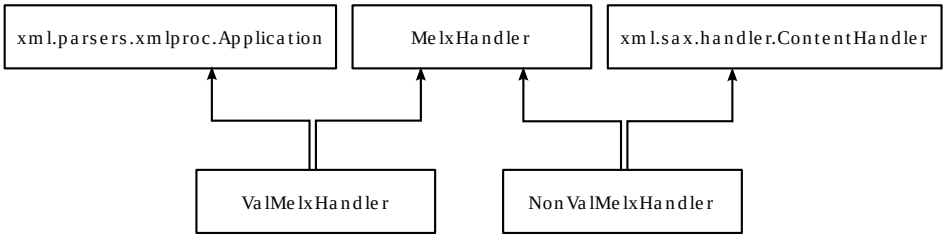

Figure 5.3: Inheritance of `ValMelxHandler` and `NonValMelxHandler`

which are not needed at runtime, and thus reduce the overhead of the loading, which can even on modern machines take a considerable amount of time.

Both classes, `ValMelxHandler` and `MelxHandler`, implement a subset of the SAX API, a method and specification for how to parse XML documents. It has been decided against DOM, another API specification for parsing XML documents, to reduce the memory footprint of *mlx*. As the data to be read is stored in a custom data structure, it is not necessary to store it also in a DOM tree.

However, as both classes have to provide the same core of functionality, all common code has been put into the `MelxHandler` class. Actually, the other handlers don't implement any functionality at all, they simple map calls from the appropriate parser to the `MelxHandler` class.

To understand the structure of `MelxHandler` we need to be aware of the fact that a SAX parser calls certain methods. As far as `MelxHandler` is concerned, there are three methods of interest.

- `handle_start_tag` gets called for each starting XML tag. Along with the name of the element, the method gets also the element's attributes.
- `handle_end_tag` is called for each ending XML tag. As there is no other information, only the name of the element is passed to the method.
- `set_locator` gets called with an object that is to be stored for later use. This object, a locator, can be asked to give the end position of a SAX event, e.g. in the start or end method. The handler uses it mainly for providing line numbers when reporting errors and warnings about the processed document.

When we closely look at the implementation of the first two methods, we will detect that they try to find an appropriate method and call it, if it is available. This is done using the powerful introspection tools[5] of Python. Listing 5.18 shows the `handle_end_tag` method. Thus these methods function only as a redirector of the work to be done.

```
260 def handle_end_tag (self, name):
261     # try to call a specific method
262     fn = 'do_end_' + name.replace('-', '_')
```

[5] A very good introduction to the `hasattr` and `getattr` functions is given in [15].

```
263     if hasattr(self, fn):
264         getattr (self, fn)(name)
```

Listing 5.18: handler.py / 260–264 (Using Python's introspection tools)

As we can learn from listing 5.18, in the case of an end tag, the method to be called must be named `do_end_element-name`. In the case of a start tag, it is `do_start_element-name`. Thereby, all dashes in the element's name are converted to underscores. Thus the real work of processing the compiler's input is handled by individual methods, if they are defined. This is an very elegant way which allows incremental development[6].

Although it would be possible, `MelxHandler` does not check the structure of the input stream, which is defined by the DTD `melx.dtd`. It simply relies on correct syntax of the parsed document. Using validating parsers, like `xmlproc`, checking the syntax is done by the parser, and the handler can concentrate on the actual work, creating a tree structure for use in the `cmf` module. The following paragraphs will explain the basic steps. Nevertheless, the study of the source code will be necessary to understand the handler.

When the "`mexc`" element starts, this is the root element of a menu description in the *melx* language, the handler's `do_start_melx` method gets called, which creates a `Cmf` object and stores it in the member variable `cmf`. This object needs to be filled with menu tables and those in turn with menu lines and so on. When the end tag of the element is fetched, `do_end_melx` is called, which assumes the object stored in the `cmf` member variable to be properly filled with the menu hierarchy as described by the input document. After doing some "post processing" of the created structure there is nothing more to do by the handler. We will look later at what "post processing" exactly does.

The next interesting element to be handled is "`top-menu`". Its arrival at the parser causes `do_start_top_menu` to be called. In this method, the `top_menu` member variable gets assigned a tuple of two values, a string that uniquely references something – in this case a menu table – and the current location in the input stream. Actually, we would like to store a `Menu` object in the member variable, but we haven't got the menu table's definition yet. Whenever, we cannot reference an object, because its description still hasn't been read, we do store a tuple instead of the object. This tuple is called a *pending reference* and has to be replaced by the real object later.

The methods `do_start_menu` and `do_start_line` go hand in hand with each other. The first creates a `Menu` object and stores it in the member variable `__cur_menu`. Additionally, the created object is also stored in the dictionary `__menus` along with the location of its definition in the source document. `do_start_line` creates a `MenuLine` object with two pending references and appends it to the current `Menu` object's `mlines` list – the current `Menu` object is referenced with the `__cur_menu` member variable. When a "`menu`" element ends, `do_end_menu` removes the reference to the current `Menu` object.

`do_start_line_format` is similar to the "`menu`" handling method. Under `__cur_lf` a new list is created and appended to the `__lfs` dictionary. Child elements of a "`line-format`" ele-

[6] By speaking of incremental development we want to say that the software can be developed and tested in small iterative cycles on real data. The full meaning of the term "incremental development" is explained in [16].

ment create instances of subclasses of `Lc` and append them to `__cur_lf`. `do_end_line_format` simply destroys the reference to the current line format list hold in `__cur_lf`.

Another interesting method is `do_start_hfill`. In the case of a non-zero specified "`count`" attribute, it simply creates an appropriate `LcString` object and appends it to the current line format list (`__cur_lf`). In the other case, no object but the pure character is appended to the format list, and additionally a reference to this line format list is appended to the `__lfs_with_hfills` list.

Having reached the end tag of the "`melx`" element, there is the following data gathered, which still needs to be processed in some way.

- `top_menu`; a pending reference to a `Menu` object regarded as the top-level menu table.
- `cmf`; a `Cmf` object. Its `menu` member variable is still `None`.
- `__menus`; a dictionary of `Menu` objects with the location of their definitions in the source file. Further, each `Menu` object's `mlines` member variable holds a list of `MenuLine` objects. If these objects' `lcs` member variable is a tuple, it is a pending reference to a list of line components. Also, if these objects' `submenu` member variable is a tuple, it is a pending reference to a `Menu` object regarded as a submenu.
- `__lfs`; a dictionary of lists of line components. Each list is accompanied by the location of its definition in the source file ("`line-format`" element).
- `__lfs_with_hfills`; a list of references to lists stored in the `__lfs` dictionary. These referenced lists contain an "hfill" which needs to be expanded.

`do_end_melx`, shown in listing 5.19, calls several methods which perform post processing tasks on the above listed data. The names of the calls speak for themselves. Actually, only the calls to `expand_hfills` and `fix_pending_refs` are necessary. The others can be regarded as optimization or error checking processing. After `do_end_melx` returns, the `cmf` member variable references a properly initialized data structure, which now can be passed to the `cmf` module to generate the byte code.

```
274 def do_end_melx (self, rname):
275     self.expand_hfills ()
276     self.concatenate_const_strings ()
277     self.check_line_widths ()
278     self.fix_pending_refs ()
279     if not self.has_errors():
280         self.check_for_endless_loops ()
```

Listing 5.19: handler.py / 274–280

5.2.3 The Controller

To combine the two layers into one application, the file `mlx.py` implements a command line interface for the user and performs all necessary work to produce the desired result using the `cmf` and handler modules.

Parsing command line parameters is solely done with the `getopt` module from the standard Python library. This module's `getopt` function supports the "same conventions as the UNIX `getopt()` function" [17]. It provides a standard behavior for reacting against badly specified parameters and allows the application to keep the code for evaluating the parameters small.

After calling `parse_file`, which results into a call to the first layer, the returned `Cmf` object, if not `None`, is directed to generate the byte code, this is the invocation to the second layer. If everything goes alright, the byte code generation routine returns a list of bytes along with an appropriate `MemAllocator` object. The object contains computed addresses of variables allocated for the menu description, theirs synonyms, and some other information about each. Depending on what the user specified at the command line, the byte code list and the `MemAllocator` object are used to produce the compiler's output. `write_c_header`, `write_h_header`, and `write_binary` are responsible for this. Their implementation is pretty easy.

The most interesting part of `mlx.py` is the implementation of `parse_file`. Python allows to dynamically load modules, and this in turn allows the routine to be written for use with different modules without the requirement of having all the modules installed. Only those, that are going to be used at runtime, are needed. If the routine is told to validate the input document, it tries to `import`[7] the `xmlval` module from the `xmlproc`[8] distribution. Then a parser object is created along with an object of the class `ValMelxHandler`. This object is registered to the parser, and finally parsing is started. In the other case, when a non-validating parser class from the standard Python library is used, the steps are the same with the exception of instantiating the `NonValMelxHandler` class and importing the appropriate modules instead of `xmlval`.

[7] `import` is Python's term for loading a module into one's own application.

[8] More information about `xmlproc` can be found at [http://www.garshol.priv.no/download/software/xmlproc/].

Chapter 6

Conclusion

The goal of this work, to create a programming framework for writing user interface applications targeted at embedded systems, and its realization have revealed a great insight into different domains of development. Design of binary data formats, understanding the XML technology, design of compiler like programs, and the design of libraries as well as writing source code for low memory systems are only a few subjects addressed by this work.

6.1 Summary of Achievements

With *mlx/mexc*, a complete rewrite and stable version of the original idea was created. It serves as a base for further development and improvements, and replaces its predecessor MEL/MEX.

Extensive documentation to each part of the project was a secondary target, and was successfully fulfilled to help new developers understand these parts, quickly adopt them to their own projects, and possibly contribute to *mlx/mexc* itself.

Giving the outcomes of this work to the world as Free Software is an important milestone of the *mlx/mexc* project. Although there may be good reasons for keeping a software closed source, the advantages of Open Source and Free Software for *mlx/mexc* are too significant to be ignored. On the one hand, it helps attracting attention by the fact that programmers are free to look at and change the source code to their own needs. On the other hand, it assures that improvements by individual programmers will be available to others.

6.2 Further Development

Although much efforts have flown in to the realized project, there is still enough room for extending it or contributing to it. Here are some thoughts about what would be desirable to have for the *mlx/mexc* project. They are ordered by their level of importance.

- Availability. Currently, the project is not hosted on any server which allows download of the sources. It would be essential to the success of the project to find a good place where it can be made available to the public. Platforms like [http://www.berlios.de/] or [http://sourceforge.net/], just to mention a few, would be preferable, as they are well known and provide great support for developing Open Source and Free Software.

- Reducing library's size. By surrounding all code dealing with an individual line component with preprocessor definitions like '`CONFIG_DISABLE_`*`COMPONENT_NAME`*', it would be possible to exclude this code from the library at compilation time. Thus the application programmer might be able to reduce the size of his application by excluding functionality that is not needed.

- Configuration system. It would be nice to have a configuration system at compilation time to let the application developer quickly choose what features to include in or exclude from the library. GNU Autoconf/Automake would be a standard choice for example, however, it would require a UNIX-like environment.

- Optimization. As with each project there is still enough room for overall optimization of the whole project.

- Touchscreen interface. The *mexc* library currently uses only a keyboard interface as an input device. In conjunction with a touchscreen it would be desirable to allow the user to tap on a menu line and let *mexc* activate it. Realizing this would concern only few changes to the file `mexc.c` of *mexc*'s sources.

- Namespaces. Currently *mlx* doesn't support namespaces. The elements and its attributes in the *melx* language are not defined under a specific namespace. Introducing one would concern changes to some parts of the *mlx* compiler and slight changes to `melx.dtd`. The benefit of namespaces would be the possibility to enrich menu descriptions with information from other applications without interfering with *mlx*.

- Installation. Currently, there is no installation script for the *mlx* compiler. It would be nice to have a distribution script, possibly using the "distutils" (see [http://docs.python.org/dist/dist.html]) to do a system wide or user local installation of the program and its modules.

- Graphics. *mlx/mexc* was designed to work mainly with non-graphics displays. However, as graphical LCD become cheaper and are increasingly used, it would be desirable to have the concept of *mlx/mexc* also with graphical features, like bitmaps and variable wide fonts. Such extensions would probably be best not implemented as extensions to *mlx/mexc* itself, but as another branch of the project.

- GUI *melx* editor. Having an editor with a graphical user interface tailored to creating *melx* sources could ease the creation of menu descriptions and would take away the burden of learning *melx*. Platform portable toolkits like GTK+, QT, Tk, or wxWidgets are preferable.

Appendix A

Utilized Software

With only a few exceptions, the *mlx/mexc* project and this documentation has been developed using Open Source and Free Software exclusively. The following sections will show which programs and libraries have contributed to the development of this project.

A.1 Development environment

The main development took place under the Debian 3.1 (Sarge) GNU/Linux system with OSI[1] certified Open Source software. Development of the DOS based simulator has been done under the Microsoft Windows XP operating system.

As far as development tools are concerned, free programs in the meaning of "free speech, not free beer"[2] have been preferred.

- The GNU C compiler, released under the GNU GPL [9], was used to compile the *mexc* library, the curses and GTK+ based simulators. GCC is available for many platforms. Its official home page is [http://gcc.gnu.org/].
- For debugging purposes GNU GDB, a powerful debugger, has been used. It is released under GNU GPL [9] and available from [http://www.gnu.org/software/gdb/gdb.html].
- To build native binaries for MS Windows with GCC, tools from the MinGW project have been used. This project is located at [http://www.mingw.org/] and partially released into public domain, under the GNU GPL [9], and under the GNU LGPL [7].
- Development of the Palm OS based simulator was done under use of `pilrc` and the `prc-tools` available for GNU/Linux and MS Windows systems. Both projects are released under the GNU GPL [9]. Also needed was the Palm OS SDK Version 4.0, which is

[1] "Open Source Initiative (OSI) is a non-profit corporation dedicated to managing and promoting the Open Source Definition for the good of the community, ..." as stated at [http://www.opensource.org/].

[2] The exact phrase is "Don't think free as in free beer; think free as in free speech." and is given in [18].

not released under a license approved by OSI, but provided under the "Palm OS software development kit software license agreement" available at [http://www.palmos.com/cgi-bin/sdk40.cgi]. The presented license must be accepted to download the SDK.

- For development of the DOS based simulator the OpenWatcom C/C++ compiler products have been used. These are released under the terms of the Sybase Open Watcom Public License which is approved by the OSI. The web presentation of Open Watcom can be found at [http://www.openwatcom.org/].
- Python, a modern and very popular programming language, was used to implement the *mlx* menu compiler. The Python interpreter is released under a GPL-compatible license and presented on the web at [http://www.python.org/].
- Subversion, a revision control system, has been used to keep track of changes during the development of all parts of the project. It is released under an Apache/BSD-style license, which is given at the projects home page, which is located at [http://subversion.tigris.org/].

The following third party libraries – libraries that are not shipped with the mentioned development tools – where involved in the development of some simulators and *mlx*.

- *mlx* optionally uses `xmlproc`, Python modules for parsing XML documents. It is released under a formal BSD-ish license and available from [http://www.garshol.priv.no/download/software/xmlproc/]. It is also distributed as part of the "XML package for Python" located at [http://pyxml.sourceforge.net/].
- GTK+, the GIMP toolkit, a library for creating graphical user interfaces, is used by `gsim`. This library has been ported to many platforms and is released under the GNU LGPL [7]. Its home page is located at [http://www.gtk.org].
- `csim` is implemented using the `ncurses` library. `curses`, the predecessor of `ncurses`, has a long history and actually is part of each UNIX-compatible system. For more information on `ncurses` see [http://dickey.his.com/ncurses/ncurses.html] or [19].

A.2 Typesetting and Drawings

This document as well as the stand-alone versions of the *mexc* and *mlx* documentation were created using teTeX ([http://www.tug.org/teTeX/]), a TEX distribution for UNIX compatible systems which consists only of Free Software. However, the index of this document, as produced by the tools of the teTeX distribution, was customized by a small self-written Python program (`fixidx.py`) which is distributed with this document's sources.

All graphical drawings of this document has been created with InkScape, an Open Source scalable vector graphics (SVG) editor which can export SVG to PostScript format for use with LATEX documents. InkScape is released under the terms of the GNU GPL [9] and is available from [http://www.inkscape.org/].

Appendix B

Source code

The printed edition of this document is supplemented by a CD-ROM with the source code for the *mlx/mexc* project. Beside the sources of *mexc*, *mlx*, and the simulators, there are also included the LaTeX sources of this documentation and its compiled version in various formats.

The directory hierarchy on the CD-ROM is structured as follows. `dist` denotes the mount point of the medium, thus there is no such directory on the CD-ROM.

`dist/`	root of the CD-ROM
`bin/`	pre-compiled versions of some simulators
`gsim.exe`	a version of the GTK+ based simulator for use on Win32 platforms
`dsim.exe`	a simulator for use on MS DOS platforms
`psim.prc`	a simulator for use on Palm OS platforms
`doc/`	documentation directory
`thesis.*`	compiled version of this document in PDF and PS formats
`mexc.*`	compiled stand-alone documentation of *mexc* in PDF and PS formats
`mlx.*`	compiled stand-alone documentation of *mlx* in PDF and PS formats
`packed/`	contains tar-gzipped source packages
`src/`	source code directory
`csim/`	sources for a curses bases simulator
`dsim/`	sources for a DOS based simulator
`gsim/`	sources for a GTK+ based simulator
`psim/`	sources for a Palm OS based simulator
`mexc/`	sources for the *mexc* library
`mlx/`	sources for the *mlx* menu compiler
`data/`	test input files for *mlx* compiler
`docs/`	LaTeX sources for this document

Appendix C

GNU Free Documentation License

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document ”free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice

grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last

time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all

as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special

permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

References

[1] Niall Murphy. *Graphics Libraries for Embedded Systems*. (Available from http://www.embedded.com/97/feat9708.htm).

[2] LinuxDevices.com. *Embedded Linux Graphics Quick Reference Guide*. December 2005. (Available from http://www.linuxdevices.com/articles/AT9202043619.html).

[3] Wikipedia. *Endianness*. December 2005. (Available from http://en.wikipedia.org/wiki/Endianness).

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau Teil 1*. Oldenbourg Wissenschaftsverlag GmbH, Rosenheimer Strasse 145, D-81671 München, 1999.

[5] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly & Associates, Inc., 2nd edition, 2002.

[6] Wikipedia. *Menu (computing)*. December 2005. (Available from http://en.wikipedia.org/wiki/Menu_%28computing%29).

[7] Free Software Foundation. *GNU Lesser General Public License*. February 1999. (Available from http://www.gnu.org/copyleft/lesser.html).

[8] Hubert Högl. *MEX - The Menu Executor*. April 2001. (Available from http://www.fh-augsburg.de/~hhoegl/da/da-22/mex.html).

[9] Free Software Foundation. *GNU General Public License*. June 1991. (Available from http://www.gnu.org/copyleft/gpl.html).

[10] Hubert Högl. *A 'Menu Language' introduction*. May 2004. (Available from http://www.fh-augsburg.de/~hhoegl/da/da-22/mel.html).

[11] Huber Högl. *The Portable-Menu-Format*. May 2004. (Available from http://www.fh-augsburg.de/~hhoegl/da/da-22/mel.html#SEC23).

[12] Eric Raymond. *Jargon File 4.4.7*. December 2003. (Available from http://catb.org/~esr/jargon/).

[13] Wikipedia. *Optimization (computer science)*. December 2005. (Available from http://en.wikipedia.org/wiki/Optimization_%28computer_science%29).

[14] Guido van Rossum. *Python Tutorial*. 2005. (Available from http://docs.python.org/tut/tut.html).

[15] Mark Pilgrim. *Dive into Python*. Apress, July 2004. (Also available from http://diveintopython.org/).

[16] Wikipedia. *Iterative and incremental development*. December 2005. (Available from http://en.wikipedia.org/wiki/Iterative_development).

[17] Guido van Rossum. `getopt` – *Parser for command line options*. September 2005. (Available from http://docs.python.org/lib/module-getopt.html).

[18] Sam Williams. *Free as in Freedom*. O'Reilly, Mai 2002. (Also available from http://www.oreilly.com/openbook/freedom/).

[19] Pradeep Padala. *NCURSES Programming HOWTO*. 2005. (Available from http://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/).

Index

G

H

I

K