# MEXC - The Menu Executor

Petr Novotník

December 19, 2005

**Abstract**

This document describes *mexc* the “Menu Executor”. Beside explaining the functionality of the library and how it interacts with the user, an explanation is given about how to bind *mexc* to one’s own programs. A description of the interpreted byte code is not given here, but can be found in the documentation of the *mlx* compiler.

## Contents

## 1 License

The *mexc* library is Free Software provided under the terms of the GNU Lesser General Public License (LGPL). The file `COPYING`, that is distributed with *mexc*, contains a copy of the GNU LGPL. It can be also retrieved from the web at [`http://www.gnu.org/licenses/lgpl.html`].

## 2 Introduction

The idea for a menu executor was born in the late nineties by Hubert Högl[1] during the development of an embedded system with a small alphanumeric display connected to it. While developing some programs for his system, he realized that it would be useful to have a library for displaying menus on small 20x4 or 16x2 character based displays and has begun to implement such a library called "MEX". *mexc* is a complete rewrite of Mr. Högl's work with a lot of changes and improvements.

*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 liquid crystal display (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 1 shows a 16x4 display with the expected keyboard and the additional eleven keys.

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.

[1]Have a look at [`http://www.hhoegl.com/mel/mel.html`].

```
lcd:                              keyboard:
 column 1 ...  16                 +-------------+-------+
+----------------+                |      ^      | 1 2 3 |
|................|  row 1         |      |      | 4 5 6 |
|................|  .             | <- enter -> | 7 8 9 |
|................|  .             |      |      | 0   . |
|................|  row 4         |      v      |       |
+----------------+                +-------------+-------+
```

Figure 1: A 16x4 LCD and keyboard design

## 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.

```
+----------------+
|TTTTTTTTTTTTNLLK| <- Header line
|C..............S| <- Menu line 1
|C..............S| <- Menu line 2
|C..............S| <- Menu line 3
+----------------+
```

Figure 2: Display layout

Figure 2 shows how a four line display [2] 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 ('`!`').

[2] The original sketch and documentation to it can be found at [`http://www.hhoegl.com/mel/doc/mex_2.html`].

- 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 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[3] shown in figure 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.

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

Figure 3: A display example

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.

[3]The original example which was introduced by Mr. Högl originates from the documentation of MEX, it was slightly modified here. It can be found at [`http://www.hhoegl.com/mel/doc/mex_2.html`].

## 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.

## 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 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 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.

## 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.

### 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.

### 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.

## 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.

## 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.

## 6.5 Date fields

The date field is very similar to the time component. It consists of three parts: year, month, and day.

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

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.

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.

## 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.

## 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.

## 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.

```
                           +------+-----+------+
0 -> .,?!’"0-()@/:_        | ‘1’  | ‘2’ |  ‘3’ |
1 ->  1                    |  _1  | ABC |  DEF |
2 -> ABC2abc               +------+-----+------+
3 -> DEF3def               | ‘4’  | ‘5’ |  ‘6’ |
4 -> GHI4ghi               | GHI  | JKL |  MNO |
5 -> JKL5jkl               +------+-----+------+
6 -> MNO6mno               | ‘7’  | ‘8’ |  ‘9’ |
7 -> PQRS7pqrs             | PQRS | TUV | WXYZ |
8 -> TUV8tuv               +------+-----+------+
9 -> WXYZ9wxyz             | ‘0’  |     |  ‘.’ |
. -> .+[]{}<>=*$           | .,?  |     |  .+[ |
                           +------+     +------+
```

Figure 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 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.

### 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 5.

## 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.

### 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.

### 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. 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.

```
              40
   +----------------------+
   | screen               |
   |                      |
   |                      |
20 |                20    |
   |           +----------+
   |         5 |   mexc   |
   +-----------+----------+
```

Figure 6: Display arrangement for an example application

*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 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.

### 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. 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.

### 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.

### 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.

## 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 7.3. Using the GNU C compiler, the size of *mexc* for a 32-bit Intel platform should vary between 10KB[4] and 20KB[5] with `assert`s disabled.

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.

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

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.

## 7.3 Compiling the interpreter

*mexc* is known to compile smoothly with GCC[6] and was successfully tested on Intel architectures as well as on a Motorola DragonBall VZ processor. Using the OpenWatcom C/C++ compiler tools[7], *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 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 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`

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

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 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 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 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 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.

`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 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 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.

### 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 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. It takes an XML document and creates the binary image, also referred to as byte code. The output of the compiler are two files, `a.c` and `a.h`. In the C file there is the byte code as an array. The header file contains a declaration of a pointer to the byte code and definitions which originate from the input document. For more information read the *mlx* documentation.

- 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.

### 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:

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

Listing 3: mtypes.h / 28–32

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

Listing 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`.

### 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`.

### 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.

### 7.4.4 mexc_set_callback_handler

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

As already mentioned in section 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.

### 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!

### 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.

### 7.4.7 An example

Figure 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 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 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. Read the *mlx* documentation to learn more about data types of the individual components. 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 6: Accessing menu variables in callbacks

# 8 Simulator

During the development of *mexc*, a simulator was needed to test and debug the code. `gsim` is a GTK+-2.0 [8] 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.

## 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 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[9] project. The GTK+ library 2.0 or higher is required. When using `Makefile` also the `pkg-config` program will be needed.

## 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 8: Command line arguments of the simulator

[8] To learn more about GTK+ visit its home page at [`http://www.gtk.org`].

[9] The home page of the MinGW project can be found at [`http://www.mingw.org`].

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.

## 9 Known issues

Currently, there are the following restrictions:

- *mexc* doesn't support a password for the top-level menu table. If there is one, the library will simply ignore it and dislay the menu table as if there was no password protection.
- *mexc* is written in a manner that assumes there is always at least one menu line to be displayed. Having dynamic menu lines can lead to a situation where all lines of a menu table are disabled. This must be avoided. The simplest way to do so is to put a non-dynamic line into the menu table.
- When rendering line components there arises the question what to do when a component doesn't fit completely on the screen. *mexc* is very strict in this respect and won't render such a component at all.