MLX - Menu Language Xml

Petr Novotník

December 19, 2005

Abstract

This document describes the *mlx* byte-code compiler v0.5, its input language *melx* “Menu Language Xml”, and also its generated output CMF “Compact Menu Format” v0.4.

Contents

1 License

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

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."[1]

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 10. Looking closely at both languages, we will notice that there is little difference between them in structure.

3 Bird's-eye view

As first, let's look at what *mlx* is for and how we can use it.

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. Then we can use programs like `xmlproc_val`[2] for validation to be sure the structure of the input document to *mlx* is valid.

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`.

[1] Hubert Högl; [`http://www.fh-augsburg.de/~hhoegl/da/da-22/mel.html`]

[2] This program comes with the `python-xml` package found at [`http://pyxml.sourceforge.net/`].

3. Now the generated C file can be compiled and linked to an interpreter and/or a menu displaying program.Together with the *mlx* compiler an interpreter library has been written which can navigate through the menu and invoke callback functions. More information about *mexc* can be found in its documentation.

To understand why *mlx* can be so useful in conjunction with an interpreter, imagine we have an embedded system with an LCD connected to it and want to present a menu to the user. Not using *mlx*, we would probably write a not trivial assembler or C program which would do the displaying and navigation through the menu. When we create another system with a completely different menu structure but the same look and feel ... why should we write another assembler or C program, why not use one library with a defined set of functionality and only implement the system dependent functions? Exactly this is the idea behind the concept of *mlx* and a byte code interpreter. As indicated in the above listed points, all we have to define is the menu structure and the rest is more or less already done.

There is one thing to note. The design of the byte code, the compiler, and the provided *mexc* interpreter is driven by the idea that the byte code itself is placed in a read-only accessible memory region. Another read-write accessible memory region must be available where the current state of the menu is stored. However, both regions are separated from each other.

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`[3] package, are needed.

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 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`,

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

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.

```
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').

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.

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.

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.

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.

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.

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 5.4.1.

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 5.4.1.

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

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 5.4.1.

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

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 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.

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 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.

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 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.

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 5.4.1.

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

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 5.4.1.

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

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 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 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.

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.

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 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.

```
~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 1: Command line parameters of *mlx*

--help This option prints a listing as shown in figure 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 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[4].

--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[5].

--output This option requires an argument. By default, the *mlx* compiler will output two files

[4] *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.

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

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, 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.

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), however, CMF differs to make the code more compact and efficient.

The following sections reflect many things already discussed in section 5 and will be interesting to programmers who want to implement a CMF parsing program.

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.

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 2: Overall structure of CMF

The definition in figure 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`.

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 1: Checking the byte–order mark

```
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 3: Definition of `line-opts` in CMF

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`.

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 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. *mexc*, for example, doesn't draw menu lines which the enable value is zero. Another way would be to gray out these lines.

`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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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 2.

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

Listing 2: Definition of cmf_time_t

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

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.

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 3.

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

Listing 3: Definition of cmf_date_t

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

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.

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.

```
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: 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.

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.

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'.

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.

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.

7.6 CMF parser

With the *mexc* interpreter library, C code has been implemented to navigate through the CMF and provide the programmer with structure definitions for each line component. This code is released under a free license and can be used for new projects. The parser is written in the files `cmf.c` and `cmf.h` of the *mexc* sources.

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 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 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.

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 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 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 7.5.1.

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 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 6 shows what we would append to `melx.dtd`.

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

Listing 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 7.

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

Listing 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.

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 7.5. In our example, we will add a '*checkbox*' to the dictionary with the next free ID as shown in listing 8.

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

Listing 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 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 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.

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 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 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.

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.

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
                  %blink-attr; >

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

Listing 11: The Melx Data Type Definition