EMX

The
Embedded Mutable eXecutive
Framework

Draft

Version: 1.0

Index

1 Abstract....3
1.1 Acknowledgements....3
1.2 License, Copyright Notice, Disclaimer....3
2 What is “EMX” ?....4
2.1 What is EMX ! ?....4
3 Quick-Start....5
3.1 “Hello, Kaptain!”....5
3.2 “Hello, HELLO!”....7
4 Basic EMX Concept....9
4.1 It's (Almost) all About UI and Configuration....9
4.2 A First “Menu”....9
4.2.1 Code Discussion....10
4.2.2 Running the Menu....10
5 Menus and Callbacks....12
5.1 Adding a Sub-menu....12
5.2 A 'Copyright' Callback Function....12
5.3 Concept Summary....13
6 More Callback Types....14
6.1 Naming Conventions....15
6.2 Examples, Type by Type....15
6.2.1 U8PER: Display a uint8_t Value in EEPROM....16
6.2.2 U8PE, Edit a uint8_t Value in EEPROM....17
6.2.3 U8MMPE: Edit a min/max uint8_t Value in EEPROM....18
6.2.4 U8MMPEC: Edit a uint8_t Value in EEPROM and Call an 'OnChange' Function 19
6.2.5 U8MMSPE: Edit a 'scaled' uint8_t Value in EEPROM....21
7 Advanced Types....23
7.1 Textual Value Representation....23
7.1.1 ITLPE: Text-Based Value Display and Editing....23
7.2 Dynamic Value Processing....24
7.2.1 U8MMDPE: Dynamic Address Calculation Upon Runtime, Array Processing....24
7.2.2 ITLDPE: Text Based Display/Editing of Array-Values....27
8 Application Control....29
8.1 Menu Return Codes: Governing the EMX Framework....29
9 Debugging Aid....32
10 More to Come....35
10.1 EMX: Small, Medium, Large....35

1 Abstract

This manual describes EMX, the “Embedded Mutable eXecutive” framework.

1.1 Acknowledgements

Parts of this work are based on the work of others, without which this library wouldn't exist.

So I would like to express my very great appreciation to (in the order of their appearance in my project):

- Rolf Bakke, aka “Kaptain Kuk”, the designer of the KK2 board and the first author of the KK2-firmware
- David “HappyHappySundays” Thompson from Australia, as his software was the first one I have seen which was entirely written in 'C' which is my personal 'native' Programming language and thus helped me to get started with my library.
- All the developers from rcgroups.net, which -without knowing it- helped me a lot with their assembler-code to better understand the functioning of the kk2-board.
- The makers of “http://www.mugui.de” and the developers of the “mugui” library. It was a great starting point to understand what a 'font' is and how it can be organized.

and lots of other people which were more or less involved even without knowing this, as I benefited from their work in one or another way.

1.2 License, Copyright Notice, Disclaimer

2 What is “EMX” ?

EMX is an open-source library written in C for the so called 'KK2' board family. The original 'KK2' board was originally designed and programmed by Rolf Bakke. it is used to stabilize RC-controlled aircraft vehicles.

The original firmware of the KK2 flight-controller and their derivatives are mostly programmed in assembler. Now, with the EMX framework, convenient firmware-development for the KK2 architecture becomes tangible.

EMX focuses on the user-interface and the eeprom-based configuration of the KK2 board and provides fonts, menus, callback-functions and EEPROM handling with convenient yet lightweight functions. EMX supports the KK2 "ST7565" LCD with automatic initialization and graphical output as well as miscellaneous fonts and text output. Basic graphical functions are included.

As EMX is designed as a library with a small footprint and a very fine granularity, only the really needed, indispensable binary code is linked to the final code thus yielding lightweight results by means of code-size.

Since EMX is focused on user-interaction, the priority was not on execution speed. Instead, code-size had the highest design-priority. However, EMX is by no means slow in that it would influence firmware-execution beyond the user-interface: at any time, the developer has full control over the hardware resources as far as not limited by the KK2 hardware itself.

2.1 What is EMX ! ?

Read: What is EMX NOT ?

1. EMX is not a firmware.
2. EMX is not an application.

Instead, EMX provides the tools to write the frame for a firmware, not only for RC-controlled aircraft vehicles, but for any purpose one might the KK2 board find useful for.

3 Quick-Start

Chewing theory is boring so lets immediately start with an example.

Alas, I have to harass you with a few but probably necessary hints. I'll try to keep it short:

1. All my examples (code, compilation, linkage, download) were developed, tested and loaded into the KK2-board on a pure Linux platform, a plain text-editor, avr-gcc, avrdude, no IDE, nothing special.
2. I can not answer any questions related to IDEs like Atmel Studio, Eclipse and/or platform-questions, e.g. Windows. Please consult the forums around the world, there are tons of sources.
3. Any examples shown here are based on common practices and can similarly be applied to any other development environment. The parameters and procedures are similar to the ones used in IDEs, except that IDEs usually hide such details.
4. In the reference manual (*yet to be written*) you may find some tips which can help to solve platform or IDE related stumble-stones.
5. My compiler version is avr-gcc 4.7.2. There should exist a similar version for Atmel-Studio 6.1
6. Basic development knowledge as well as C-programming language skills is assumed.

The only requirements to use EMX are

- the inclusion of the EMX header files
- the linking of the EMX-library

For both of the above requirements exist compiler-parameters. In IDEs one usually has to define something like 'additional include-dirs' and 'libraries to link with'. On the command-line, the invocation is as follows [1]:

```
emax@hal:~$ gcc […] -Ipath/to/emx-includes -Lpath/to/libemx -lemx
```

where […] is a placeholder for the other parameters of the compiler invocation.

Note: As Linux is told to be 'complicated' and IDEs are 'known' to be 'much easier to use', it shouldn't be a problem to reproduce all steps described here on such a 'much easier to use' IDE and/or platform.

SCNR.

3.1 "Hello, Kaptain!"

Let's write a first string on the LCD screen: "Hello, Kaptain!", which is of course a tribute to the designer of this board, Rolf Bakke, aka "Kaptain Kuk" :-)

- **NOTE:** code listings in this document concentrate on the essentials. The line numbers reflect to the best of my knowledge the original line-numbers of the corresponding example in the `examples/...` directory.

  Three dots […] indicate omission of lines *within the listing*. For example, the first three dozen lines contain the header-comment of each source-file which is not relevant for this documentation and thus not listed here. The original examples of course contain the complete code.

Example 1:

```
...
 38   #include <emx/Fonts.h>
```

1 *In general, the command line prompt is **black**, input is **blue**, output is **red**. Important parts are highlighted*

```
39   #include <emx/Lcd.h>
40
41   #include <util/delay.h>
42
43   int main(void)
44   {
45       // wait for voltages and levels to settle
46       _delay_ms(100);
47       lcdInit();
48       lcdBufPuts(AT_RAM, "Hello, Kaptain!", &Verdana11, 0, 10, COL_BLACK);
49       lcdUpdate(1);
50
51       while(1)
52           ;
53   } // main
```

If you compile and flash the code, the string “Hello, Kaptain!” should appear on the display.

By the way, that's how my Makefile compiles the program:

```
emax@hal:~$ make
avr-gcc -c -mmcu=atmega324a -I.  -DF_CPU=20000000UL -DDEBUG_CODE
-I../../../emx/include -Os -Wall -Wstrict-prototypes -std=c99  -funsigned-
char -funsigned-bitfields -fpack-struct -fshort-enums -save-temps main.c -o
main.o
avr-gcc -mmcu=atmega324a -I.  -DF_CPU=20000000UL -DDEBUG_CODE
-I../../../emx/include -Os -Wall -Wstrict-prototypes -std=c99  -funsigned-
char -funsigned-bitfields -fpack-struct -fshort-enums -save-temps  main.o
--output main.elf  -Wl,-Map=main.map,--cref    -L../../../emx/lib -lemx
```

The highlighted parameters show how the include files and EMX-library is included. With these parameters and the current version (as of February 2014) I get

```
emax@hal:~$ avr-size main.elf
   text    data     bss     dec     hex filename
   2298      16    1024    3338     d0a main.elf
```

So we have about 2.3 KB of code including the LCD initialization, LCD-buffer handling, character output and font. The 16 Bytes in the '`data`' segment are the string “`Hello, Kaptain!`”, 1024 bytes in `bss` are the `LcdBuffer` which is unavoidable and results from 128*64*1 bits = 8192 bits = 1024 bytes.

Lets discuss each line:

- Lines 38 and 39 include the EMX-files for the font declaration “`Verdana11`” and the Lcd functions which is in our case only the `lcdBufPuts(..)` call.

  NOTE: I've made a habit of `#include`-ing files in the order of “most special” first and “most basic” last. It has turned out to be a safe way to avoid or identify clashed with compiler-assumptions in cases, were a compiler has to make assumptions of any kind: What I have already defined, doesn't have to be guessed by the compiler. You will find this policy throughout the library and the examples as well.

- Line 41 is the `avr-gcc` include file for the delay-macros.
- In line 46 we give the system some time to settle.
- In line 47, the Lcd hardware, and the Lcd-buffer is initialized.
- In line 48 we write the string “`Hello, Kaptain!`” to the lcd-buffer. This does not yet write anything to the display, but stores the data in the internal buffer.

- In line 49 this buffer is finally flushed to the display.
- The rest is self-explanatory.

For the time being, there is not much more to say. So lets see another example.

3.2 “Hello, HELLO!”

The code below can be compiled in the same way as the previous example:

Example 2:

```
...
38   #include <emx/Fonts.h>
39   #include <emx/Lcd.h>
40
41   #include <util/delay.h>
42
43   int main(void)
44   {
45       // wait for voltages and levels to settle
46       _delay_ms(100);
47       lcdInit();
48       lcdBufPuts(AT_RAM, "Hello, Kaptain!", &Verdana11, 0, 10, COL_BLACK);
49       lcdBufPuts(AT_RAM, "Hello, Kaptain!", &Verdana19, 0, 30, COL_BLACK);
50       lcdUpdate(1);
51
52       while(1)
53          ;
54   } // main
```

If you run the code, you will see two strings with identical texts on the display, but with different sizes. Line 49 reveals the trick: We use `Verdana19` as the font for the second string, which is bigger than `Verdana11`.

But things are never for free, look at the size:

```
emax@hal:~$ avr-size main.elf
   text        data         bss         dec         hex     filename
   3764          16        1024        4804        12c4     main.elf
```

There is an increase of almost 1500 bytes, which can of course not be explained with a second call to `LcdBufPuts`. The reason is the font: `Verdana19` is now linked to the program which was not the case in the first example. This font consumes additional space thus making the program larger.

However, this shows one of the concepts which have been implemented consequently throughout EMX: *granularity*. All entities in EMX are almost as atomic as possible to enable the linker to efficiently pick only the modules from the EMX-library which are really necessary. For that reason, you cannot not precisely answer the question “how big is EMX?”, because the answer is: “It depends”.

It depends on what you use from that library. Let me explain some basics of that concept:

- EMX is designed, compiled and written in form of a “*one entity – one unit-of-compile*” pattern.
- To put it simple, this means:

  - one function – one source
  - one global entity – one source
- Each of these *units-of-compile* (i.e. “*each single source*”) is compiled alone to one single object. For example, the `LcdBufPuts(..)` method is coded in an own source-file “`LcdBufPuts.c`” which in turn is compiled to “`LcdBufPuts.o`”.
- Each of these objects is then archived in the EMX-library `libemx.a`.
- That way, when you link your program with `libemx.a`, the linker can pick exactly what is needed and

thus creates a program which is as small as possible.

- So the size of the program is depending on what you use from EMX. Or, to put it differnetly: An EMX-method which you don't use will not be linked to the program.

This is of course a platitude. But from what I have seen in embedded applications, this doesn't seem to be common knowledge. The fact that a library should be as atomic as possible is very often ignored, and sometimes it is even the build-system which prevents developers from building small code: In larger projects, developers can very often simply not keep track of whats really needed and what not, they plainly link all `something.o` files from a development directory to a `main.elf` and finally a `main.hex` file, thus creating a binary which contains *all these objects and entities*, no matter *whether they are needed or not*. This is of course a safe method, but it is not efficient.

EMX avoids these effects by design.

4 Basic EMX Concept

What we have seen so far is just a twee demonstration that EMX really works. But EMX has more to offer.

4.1 It's (Almost) all About UI and Configuration

EMX is focused on configuration tasks and user-interfacing. From what I have seen, most firmwares for the KK2 board spend a lot of effort on configurations and on the user-interface. EMX simplifies these tasks very much.

The core concept implements an aggregate of a menu-system and callback pointers. This means, that a developer can easily write a 'menu' and add callback-functions to each single menu-entry which allow for

- calls of other menus, i.e. sub-menus, which are self-organizing by means of scrolling, button-handling, state and the like, only limited by stack-space
- calls of functions upon selection of a menu-entry
- automatic reading, displaying, editing and updating of EPROM-data
- miscellaneous editors for varying data types such as numbers and text-lists
- automatic min/max value processing upon data-entry
- '`onChange`' callbacks when editing
- optical and acoustic feedback upon EEPROM-update and button activities
- automatic menu-navigation, automatic navigation-icons
- full menu-control with immediate return from any nested menu-level and complete removal of the menu-stack from the program-stack
- dynamic editing of arrays of values and arrays of structs. For example, if a configurations maintains an array of similar values in the EEPROM, only one menu-entry is necessary to edit these values.

EMX offers quite a powerful and flexible application-spectrum which I will introduce next.

4.2 A First “Menu”

Lets build a quick menu which can be browsed and scrolled. The menu doesn't yet offer 'actions' by means off callbacks, but it will demonstrate

- how a menu is implemented
- how the navigation works
- and how a menu is called

Since a menu can display four items simultaneously on the display if we use the 'small' font, we will implement more than just four selections, as otherwise no scrolling could be demonstrated.

Here is the code:

Example 3:

```
...
38   #include <emx/Fonts.h>
39   #include <emx/Lcd.h>
40   #include <emx/Menu.h>
41   #include <emx/Buttons.h>
42
43   #include <avr/pgmspace.h>
44   #include <util/delay.h>
45
46   const char TX_Copyright[]         PROGMEM = "Copyright";
47   const char TX_SubMenu[]           PROGMEM = "Show submenu";
48   const char TX_EepromVersionRo[]   PROGMEM = "Show EEPROM version";
```

```
49  const char TX_EepromVersion[]    PROGMEM = "Edit EEPROM version";
50  const char TX_SetContrast[]      PROGMEM = "Set contrast";
51  const char TX_ScaledVoltage[]    PROGMEM = "Voltage alarm";
52
53  const Menu_t TopMenu PROGMEM = {
54      .m_entryCount = 6,
55      .m_entries = {
56          { TX_Copyright,       VOID(0) },
57          { TX_SubMenu,         VOID(0) },
58          { TX_EepromVersionRo, VOID(0) },
59          { TX_EepromVersion,   VOID(0) },
60          { TX_SetContrast,     VOID(0) },
61          { TX_ScaledVoltage,   VOID(0) },
62      }, // m_entries
63  }; // TopMenu
64
65  int main(void)
66  {
67      _delay_ms(100);  // wait for voltages and levels to settle
68      lcdInit();
69      BUTTON_INIT();
70      while(1)
71          displayMenu(&TopMenu, BT_ALL);
72  } // main
```

After compilation, lets first have a look at the size:

```
emax@hal:~$ avr-size main.elf
   text        data         bss         dec         hex     filename
   3300           0        1025        4325        10e5     main.elf
```

The size has decreased. As explained before, EMX only uses what's really needed. As we don't use the Verdana18 font any more, it is not any more linked in. However, the size didn't entirely shrink by the font-size which we have seen above, because additional code for the menu was added.

With the first menu, a number of related modules was linked in: The small font for the menu, the menu-texts and the menu itself. Button initialization and -handling and feedback-functions was added too.

4.2.1 Code Discussion

For the menu and the BT_ALL value in line 71, the corresponding `#include` files were added in lines 40 and 41. The `#include` file in line 43 is needed for the text-lines defined in lines 46-51 which reside in flash-memory. The can be used throughout the program, but In our example, they are only used for the menu which is defined in lines 53-63. A so called `Menu_t` object contains the pointers to the text-lines and an entry-count. Very simple.

NOTE: The entry-count value must not be bigger than the number of menu-entries. If it is smaller, everything will work as expected, except that only the number of entries which are defined in entry-count will be processed. The other ones are just ignored in that case.

In line 69, the `BUTTON_INIT()` macro was added. This macro is important if you use the buttons since it sets the button-pins to be used as inputs.

In line 71 the menu is called and thus displayed. The menu waits and processes button-input until the 'exit' button is pressed, then the loop is executed again and the menu is redisplayed. The `BT_ALL` parameter in the `displayMenu` call sets the mask of buttons which are allowed to be used in the menu.

4.2.2 Running the Menu

Now run the program and watch what happens on the display:

- The menu-texts are displayed in the order in which they appear in the `TopMenu` . The order in which these texts were *defined* in the program does not influence their appearance in the menu.
- A set of navigation-icons is displayed at the bottom: left arrow to return from the menu, down-arrow to move the cursor down.
- If the cursor is in the top line, no further icons are visible. If you move one item down, the arrow-up icon

appears to indicate that you can now navigate in up-direction as well.

- If you move the cursor to the bottom, the menu scrolls line by line forward until you reach the last menu-item.
- As soon as you reach the last item, the 'arrow-down' icon disappears to indicate that navigation down is not any longer possible.

An icon for the rightmost button is currently not visible as there is no item which you could 'activate': all item-pointers are '`void`'.

If you press the 'left-arrow' button, you will leave the current menu and go back to the 'main' function. However, this is currently not visible as the menu is called again in an endless loop and thus immediately redisplayed. The only visible effect is that if you exit the menu whilst the cursor is not on the first item, the redisplayed menu will show up upon redisplay with the cursor placed on the top item.

5 Menus and Callbacks

In the example above, we created a menu which can not do anything except that it shows up on the display and manages navigation and navigation-icons. But with callbacks, we can add intelligence to the menu.

5.1 Adding a Sub-menu

If you want to implement a hierarchy of Menus, you can just add another menu and hook this up into any already existing menu.

Example 4:

```
...
 53   const char TX_EmxLogo[]          PROGMEM = "Show EMX logo";
 54   const char TX_License[]          PROGMEM = "Show license";
 55
 56   const Menu_t InfoMenu PROGMEM = {
 57       .m_entryCount = 2,
 58       .m_entries = {
 59           { TX_EmxLogo,       VOID(0) },
 60           { TX_License,       VOID(0) },
 61       }, // m_entries
 62   }; // InfoMenu
 63
 64   const Menu_t TopMenu PROGMEM = {
 65       .m_entryCount = 6,
 66       .m_entries = {
 67           { TX_Copyright,       VOID(0) },
 68           { TX_SubMenu,         MENU(InfoMenu) },
 69           { TX_EepromVersionRo, VOID(0) },
 70           { TX_EepromVersion,   VOID(0) },
 71           { TX_SetContrast,     VOID(0) },
 72           { TX_ScaledVoltage,   VOID(0) },
 73       }, // m_entries
 74   }; // TopMenu
 75
 76   int main(void)
 77   {
 78       _delay_ms(100);  // wait for voltages and levels to settle
 79       lcdInit();
 80       BUTTON_INIT();
 81       while(1)
 82           displayMenu(&TopMenu, BT_ALL);
 83   } // main
```

Besides the new texts in lines 53 and 54 you can see the new `InfoMenu` starting in line 56. Please note the value in line 57 which must not be bigger than the number of menu entries.

In line 68, the `TX_SubMenu` menu-entry has now a parameter `MENU(InfoMenu)`. Compile and run the program and see what happens:

Now, when you move the cursor to the "`Show submenu`" menu-entry, another icon appears at the bottom of the display: The *activation icon* indicates that this menu entry can be activated with the corresponding button, and if press this button, the new menu is displayed on the screen. There isn't yet any activity in that menu implemented, but for the time being this is irrelevant. You can navigate the menu, and if you leave this menu, the top-menu is redisplayed exactly as it was before.

5.2 A 'Copyright' Callback Function

Now, lets add a 'Copyright' function and see how this works:

Example 5:

```
...
 53   const char TX_EmxLogo[]          PROGMEM = "Show EMX logo";
 54   const char TX_License[]          PROGMEM = "Show license";
 55
 56   uint8_t Copyright(MenuState_t* aMs)
 57   {
 58       LCD_BUF_CLEAR();
 59       lcdBufPuts(AT_PGM, PSTR("menu system,\n"
```

```
60                                "framework and font\n"
61                                "2013-2014 by\n"
62                                "Edgar (emax) Hermanns"),
63                  &Verdana11, 0, 0, COL_BLACK);
64      lcdUpdate(1);
65      pollButtons(0xFF, 1);
66      return 1;
67  }
...
76
77  const Menu_t TopMenu PROGMEM = {
78      .m_entryCount = 6,
79      .m_entries = {
80          { TX_Copyright,       FUNC(Copyright) },
81          { TX_SubMenu,         MENU(InfoMenu) },
82          { TX_EepromVersionRo, VOID(0) },
83          { TX_EepromVersion,   VOID(0) },
84          { TX_SetContrast,     VOID(0) },
85          { TX_ScaledVoltage,   VOID(0) },
86      }, // m_entries
87  }; // TopMenu
88
89  int main(void)
90  {
91      _delay_ms(100);  // wait for voltages and levels to settle
92      lcdInit();
93      BUTTON_INIT();
94      while(1)
95          displayMenu(&TopMenu, BT_ALL);
96  } // main
```

In line 80, you see a change: the menu entry `TX_Copyright` has now a `FUNC(Copyright)` parameter which replaces the `VOID(0)` parameter from the previous example. Lines 56-67 contain the corresponding `Copyright(..)` function. You can code in that function whatever you want.

If you compile and run the program, you will see that the navigation icons now change as well if the cursor is on the 'Copyright' menu-item: Again, the 'right-arrow' indicates that you can 'activate' this menu-position with the corresponding button. If you press this button, the copyright note appears on the display.

If you leave this screen with the 'return' button, the menu is restored in exactly the state which it had before you called the 'Copyright' function. You do not have to care for the display-content, the menu-position, the cursor-position, or the navigation icons.

Again, the menu can do much more for you as you will later in this document. But let's first summarize what we've worked through so far.

5.3 Concept Summary

The examples above have shown the basic principles of the menu-system. Of course, there are more capabilities which will be demonstrated in the following examples. As you can guess, additional features are added with entry-types other than `FUNC` or `MENU`.

And thus the **basic principle** of the entire menu system should be clear: **Callbacks of different types which cause the menu system to do different things**.

The different callback-types are explained in the following.

6 More Callback Types

In general, menu-entries point to menus or functions. besides this, they can point to data-types which contain the definitions which are necessary to display or to edit a value. An overview of the entry types which are currently available is shown below along with a short description of the corresponding data-types (Parameter) and some notes about what what a particular type does.

This table is only intended to give a first overview, a more elaborate example for each type follows later in this document:

Entry Type	Parameter	Effect	Notes
`VOID`	ignored	none	Just displays a menu-text, no activation
`MENU`	menu name	calls submenu	automatic restoration of calling menu
`FUNC`	function name	calls user-function	menu is completely restored upon return from function.
`U8PE`	`U8ValP_t` name	displays and edits a `uint8_t` value	displays/calls an editor for a `uint8_t` value which is stored in EEPROM. The value can be changed within the full `uint8_t` range of 0-255. The parameter is the name of the `U8ValP_t` struct which holds the address of that value.
`S8PE`	`S8ValP_t` name	see notes	like `U8PE`, but for a signed integer
`U8PER`	`U8ValP_t` name	displays a `uint8_t` value	like U8PE, but read only: value is displayed and can not be edited.
`S8PER`	`S8ValP_t` name	see notes	like `U8PER`, but for a signed integer
`U8MMPE`	`U8MMValP_t` name	displays and edit a `uint8_t` value within settable min/max limits	like U8PE, but a settable min/max range for the editor can be defined.
`S8MMPE`	`S8MMValP_t` name	see notes	like `U8MMPE`, but for a signed integer
`U8MMSPE`	`U8MMSValP_t` name	displays and edits a 'scaled' uint8_t value within settable min/max limits	Editing like `U8MMPE`, but with a decimal point. E.g. with a defined scale of `2`, an integer value of `156` is shown with two decimal place as `1.56`. See corresponding example.
`S8MMSPE`	`S8MMSValP_t` name	see notes	like `U8MMPE` but for a signed integer
`U8MMDPE`	`U8MMSValDP_t` name	like U8MMPE but dynamic, see notes	a uint8_t value editor within min/max limits. The value can be dynamic, i.e., the value address is not fixed and can be obtained upon runtime. See corresponding example.
`S8MMDPE`	`S8MMSValDP_t` name	see notes	like `U8MMDPE` but for a signed integer
`U8MMPEC`	`U8MMSValPCb_t` name	like U8MMPE but with a user defined callback function, see notes.	Upon editing, a user-written callback function can be called upon each change of the value in question (i.e. `onChange` callback).
`U16MMSPE`	`U16MMSValP_t`	see notes	like `U8MMSPE` but 16 bit

	name		
ITLPE	ITextList_t name	a text-based value editor, see notes	displays/calls an editor for an 'Indexed-Text-List-Pointer' were the uint8_t 'index' is stored in EEPROM. The index is displayed and can be edited in form of a textual representation. For example, the values 1,2,3... can be displayed and edited as Monday, Tuesday, Wednesday etc.
ITLDPE	ITextListDP_t name	a text-based, dynamic value editor, see notes	Same as ITLPE, but the value can be dynamic, i.e., the value address is not fixed and can vary upon runtime. See corresponding example.

As you can easily see, the list is not yet complete. Some 'signed' versions are missing as well as some 16-bit types. Additional 'dynamic' versions have to be added and also some callback-versions might be necessary in the future.

Hence: more types to come.

6.1 Naming Conventions

Except for the FUNC and the MENU types, the type-names above attempt to follow a naming pattern which shall be explained below:

- Types starting with a 'U' relate to unsigned data-types.
- Types starting with an 'S' relate to signed data-types.
- The number in the type name tells the size of the data-entity
- 'MM' indicates that a min/max range for editing can be set.
- The following 'S' says, that a 'scale' for editing can be defined.
- The 'P' tells that a data-pointer, i.e. an address to a variable is contained in the corresponding struct.
- If the 'P' is preceded by a 'D', the 'P' address can be obtained dynamically upon runtime
- The 'E' indicates that the address of the variable in question is an EEPROM-address
- The 'C' says, that an 'onChange' callback-function is defined in the corresponding struct.
- An appended 'R' indicates that value to be read-only, no changes are possible.

The ITLPE / ITLDPE types differs from this naming pattern in that they does not point to a classic numeric value, but to an “Indexed-Text-List Pointer” which points to an index in EEPROM. For example, if you want to set a an abstract configuration-value which shall be translated to user-readable text upon editing, this can be done with an ITLPE type as shown below:

EEPROM value	Displayed text upon editing
0	Futaba
1	Spectrum
2	Graupner
3	Jeti
4	Multiplex

6.2 Examples, Type by Type

The following paragraphs show an example for each particular callback-type which hasn't yet been explained before. For each example, I will basically reuse the code from the previous example thus getting to a final

collection of all examples within one single source.

6.2.1 U8PER: Display a `uint8_t` Value in EEPROM

If you want to display and to edit a `uint8_t` value in a menu, use the `U8PR` or `U8PER` callback type. Let's assume that you want to display and edit the 'EEPROM' version of you kk2-board in a menu. The EEPROM-version is usually implemented as a 'magic number' or a 'version-id' in the first byte of the processors EEPROM. This byte can be displayed and edited like that:

Example 6:

```
...

 53   const char TX_EmxLogo[]          PROGMEM = "Show EMX logo";
 54   const char TX_License[]          PROGMEM = "Show license";
 55
 56   typedef struct EEProm
 57   {
 58       uint8_t EEStart; // magic byte
 59       uint8_t m_contrast;
 60   } EEProm_t;
 61
 62   // this is only a DECLARATION, no definition:\
          a fake to enable the compiler to calculate EEprom addresses.
 63   extern EEProm_t EEProm;
 64
 65   #define EEP(name) ((uint8_t*) ((uint8_t*) &EEProm.name - &EEProm.EEStart))
 66
 67   const U8ValP_t EEVersion PROGMEM = {
 68       .m_dataP = (uint8_t*) EEP(EEStart),
 69   };
 70


...

 92   const Menu_t TopMenu PROGMEM = {
 93       .m_entryCount = 6,
 94       .m_entries = {
 95           { TX_Copyright,        FUNC(Copyright) },
 96           { TX_SubMenu,          MENU(InfoMenu) },
 97           { TX_EepromVersionRo, U8PER(EEVersion) },
 98           { TX_EepromVersion,    VOID(0) },
 99           { TX_SetContrast,      VOID(0) },
100           { TX_ScaledVoltage,    VOID(0) },
101       }, // m_entries
102   }; // TopMenu
103
104   int main(void)
105   {
106       _delay_ms(100);  // wait for voltages and levels to settle
107       lcdInit();
108       BUTTON_INIT();
109       while(1)
110           displayMenu(&TopMenu, BT_ALL);
111   } // main
```

Lets start with the `TopMenu` changes in line 97. You see a `U8PER` menu.entry which the parameter `EEVersion`. `EEVersion` is the `U8ValP_t` struct defined in lines 67-69. This struct points to an EEPROM address which is calculated by the `EEP` macro. This macro is explained below. The macro is just a convenient helper to calculate the correct EEPROM address. A direct value would suffice as well which is in this case just 0 (zero), since `EEStart` is the first variable in the EEPROM and thus has the the address 0.

To put it simple: *the* `U8ValP_t` *structure* `EEVersion` *points to the correct EEPROM address.*

`U8ValP_t` is one of the types provided by EMX for utilization with callback-entries in menus. `U8ValP_t` reads "**u**int**8**_t **Val**ue **p**ointer **t**ype" and obviously points to a `uint8_t` value.

Please note the difference between data-*struct* nomenclature (`U8ValP_t`) and *callback-type* nomenclature: The `struct` type `U8ValP_t` defines *how* data is stored, whereas the callback-type has the additional

information, *where* the data resides: `U8PE`: The '**E**' tells that the struct points to EEPROM.

So `EEVersion` is the real entity where the menu entry points to. Withe the callback-type `U8PE`, the menu system knows that `Eeprom` is a struct with a `uint8_t` value which is stored in EEPROM. The *address* of that variable is stored in the member `EEVersion.m_dataP`.

The EEP macro:

Instead of `#define`-ing each EEPROM address myself, I want the compiler to calculate these addresses for me. So I simply define a typedef'd struct `EEProm` and put all EEPROM variables in that struct. With the EEP macro I let the compiler calculate the relative address of each particular variable. As my EEPROM-addresses start at zero within the EEPROM, the resulting value equals the final address within the EEPROM:

```
#define EEP(name) ((uint8_t*) ((uint8_t*) &EEProm.name - &EEProm.EEStart))
```

called with e.g. m_contrast results in

```
EEP(m_contrast)

→ (( uint8_t*) ( (uint8_t*) &EEProm.m_contrast - &EEProm.EEStart)
→ (( uint8_t*) ( (uint8_t*) 0x01 - 0x00)
→ (( uint8_t*) 0x01
```

To enable the compiler to calculate the macro, an instantiation of EEProm must be pretended to exist which we achieve with the `extern` declaration in line 63. With that declaration the compiler can 'postpone' the real address substitution of EEProm to the linker. So only the 'relative' addresses of the macros are calculated which matches exactly what we want to have: The variable *position* in EEPROM.

As no real *definition* of `EEProm` is ever made, no real such entity exists and no space is wasted for it. And since no utilization of `EEProm` exists throughout the code, the linker never complains and just ignores the unresolved `extern` “announcement” in line 63.

The bottom line is: with the `EEP` macro you can obtain the EEPROM-address of any member of the `EEProm` struct in which this member resides.

With the `EEP`-macro (see box), the `EEVersion.m_dataP` member variable (which will itself in fact be a *constant pointer* in flash memory which points to the variable in question ;-)) is assigned the address of `EEProm.EEStart` which contains the version number which we want to display and/or edit.

As you will have noticed, the corresponding menu-text has already been in the sources since the first example.

In `TopMenu`, the content of the `EEStart` member from `EEProm` now shows up. The value is displayed, but cannot be changed: we used a read-only entry (`U8PE`**`R`** instead of `U8PE`).

The value displayed depends on what is currently stored at address 0x00 of your EEPROM and can vary from 0-255. With the next example, we make this value changeable and add an appropriate editor.

6.2.2 U8PE, Edit a uint8_t Value in EEPROM

The change from the previous example is minimal:

Example 7:

```
...
 92   const Menu_t TopMenu PROGMEM = {
 93       .m_entryCount = 6,
 94       .m_entries = {
 95           { TX_Copyright,       FUNC(Copyright) },
 96           { TX_SubMenu,         MENU(InfoMenu) },
 97           { TX_EepromVersionRo, U8PER(EEVersion) },
 98           { TX_EepromVersion,   U8PE(EEVersion) },
 99           { TX_SetContrast,     VOID(0) },
100           { TX_ScaledVoltage,   VOID(0) },
101       }, // m_entries
```

```
102   }; // TopMenu
...
```

In line 98 you see a new entry which is almost the same then before, except that a `U8PE` type is used instead of a `U8PER` type. The missing 'R' removes the read-only attribute from the entry.

Happily, no extra address-entity is necessary: The `EEVersion` struct can be reused. This is so, because the variable address is of course still the same and thus nothing really changes, except that the menu system will now allow to change the value pointed to by `EEVersion`.

If you press the rightmost-button (I.e. ENTER button) on the `U8PE` type, an editor for the value pops up. The icons are mostly self explaining. However, the changes are only stored[2] if you use the 'ENTER' icon, i.e. the rightmost button, to leave the editor. If you use the 'BACK' button, the change is not applied.

After you changed the value and use the ENTER button to apply the change, the menu is redisplayed again.

Of course, both values displayed in the menu are updated to the new value as they refer to the same variable, you do not have to care for this. This is of course true for the entire menu-system: You can display and/or edit any value anywhere in the menu system, simultaneously in as many menus and menu-entries as you want.

6.2.3 `U8MMPE`: Edit a min/max `uint8_t` Value in EEPROM

If you want to display and to edit a uint8_t value in a menu within a minimum/maximum range, use the `U8MMPE` callback type.

The underlying struct type is a `U8MMValP_t` and points, like the `U8ValP_t`,to a `uint8_t` variable which lies in the EEPROM. Additionally, this type allows to set a **m**inimum and **m**aximum value to limit the range in which this value can be edited.

The corresponding example allows to set the contrast value for the attached display within a predefined range, i.e. a minimum and a maximum value which is stored in EEPROM:

Example 8:

```
...
 46   #include <stdint.h>
...
 73   const U8MMValP_t Contrast PROGMEM = {
 74       .m_min = 28,
 75       .m_max = 50,
 76       .m_dataP = (uint8_t*) EEP(m_contrast),
 77   };
...
100   const Menu_t TopMenu PROGMEM = {
101       .m_entryCount = 6,
102       .m_entries = {
103           { TX_Copyright,       FUNC(Copyright) },
104           { TX_SubMenu,         MENU(InfoMenu) },
105           { TX_EepromVersionRo, U8PER(EEVersion) },
106           { TX_EepromVersion,   U8PE(EEVersion) },
107           { TX_SetContrast,     U8MMPE(Contrast) },
108           { TX_ScaledVoltage,   VOID(0) },
109       }, // m_entries
110   }; // TopMenu
111
```

Line 46 now includes `stdint.h` which is not really necessary but nevertheless is better code: Until now, any `uint8_t` utilization (and the like) just worked by chance since one of the other include files already includes `stdint.h`. As we use such types here in the `main.c` source, it is in general good coding style to include the appropriate files explicitly which has now been caught up.

2 *The EMX-library of course uses the Atmel-supplied '`update_...`' functions to write to EEPROM. These functions only write to the EEPROM if the value in questions differs from the current value, thus saving the hardware from unnecessary wear.*

There is not much more to say: Line 107 has changed to an entry of type `U8MMPE`, a **u**`int8_t` **m**in/**m**ax **p**ointer to an **E**EPROM value named `Contrast`..The latter one is defined in lines 73-77. The members should be self-explaining: `m_min` and `m_max` define the range within which one can change the contrast value for the display, `m_dataP` points to the location where the `m_contrast` value of the `EEProm` struct resides.

If you now compile and run the program, you will see that editing can be done within the range defined in `m_min` and `m_max` of the `Contrast` entity.

HOWEVER: two things seem to go wrong:

1. The value which is display upon the very first call of the corresponding menu entry might show something unexpected.
2. The contrast doesn't change at all.

- For the first topic there is a simple explanation: The EEPROM configuration hasn't been set properly ever before. For that reason, if you call the editor for the contrast setting, you will be prompted the value which is currently stored in the EEPROM, not matter whether or not this value is plausible. Once that you press the up or down button, the displayed value will change limit value in question, i.e. the highest allowed value if you press the up-button, or the lowest value if you press the down button. Once that you *store* this value, it will be set correctly and stored, so that any subsequent calls will cause plausible results.

  In real life, an application would of course write correct values into the EEPROM upon first application start and before the menu is displayed.

  For that reason, note: Whenever you introduce a new EEPROM variable, set this variable to a valid value upon first utilization[3]. The above shown `EEVersion` gives some hints on how a new EEPROM layout can be detected and how one can respond to such a state.

- The second 'problem' is in fact no problem: You can change what is stored in EEPROM as defined by the `U8MMPE` callback type. But the change doesn't come into effect by means of a changed contrast: EMX can of course not guess, what a 'contrast-change' is.

  However, there is a solution: The `U8MMPEC` callback type, which is demonstrated in the next example.

6.2.4 `U8MMPEC`: Edit a `uint8_t` Value in EEPROM and Call an '`OnChange`' Function

If you want to edit a `uint8_t` EEPROM value in a menu and want to execute a function upon each value change, use the `U8MMPEC` callback type. `U8MMPEC` is an acronym for "**u**`int8_t`, **m**in/**m**ax **val**ue **p**ointer with **c**allback" function.

The underlying struct type is a `U8MMValPCb_t` and points, like the `U8MMValP_t`,to a `uint8_t` variable which lies in the EEPROM and contains min/max editing values. Additionally, this type allows to set the address of a callback function to be called whenever the editor for this value is called.

To understand how the callback function is called, we have to understand the parameters which are passed to that function. The callback-function must be of type `CallbackFunctor_t` which is defined as

```
typedef void  (*CallbackFunctor_t)(CallbackType_t aCBType, void* aAdr);
```

A call to a corresponding function (in our case the `setContrast` function) looks like this:

```
setContrast(aCbType, aAdr)
```

where `aCbType` contains the flags which define the callback-time and `aAdr` points to the variable which is about to be changed. Whilst `aAdr` is no rocket-science to understand, `aCbType` probably needs some further explanation:

`aCbType` is of type `CallbackType_t` which in turn contains two components of information: an enumerated

3 *Do not underestimate this note: a wrong value which was not initialized correctly can cost ones life or finger or whatsoever: If a motor spins up unexpectedly due to a non-initialized EEPROM value, this can have serious consequences. And probably nobody will ever again want to use your software.*

value which defines *when* the callback was called. The variable-editor provides three different locations/moments when it calls the callback-function:

- `CT_ENTRY`, which happens when the editor is called. This occurs only once upon each single editor-entry before any value change.
- `CT_CHANGE` which happens after each single change, i.e. whenever the value changed due to an UP/DOWN button press.
- `CT_RETURN` which happens only once upon each single close of the editor.

The second information which is contained in a `aCbType` is the key which was pressed when the callback was triggered. This especially helpful when the editor is closed, as the callback will likely want to know whether the user pressed 'ENTER' or 'BACK' to close the editor: With 'ENTER', the action will likely 'accept' the change, whilst 'BACK' will discard the change.

The `setContrast(..)` function in the example documents how all this works together:

Example 9:

```
...
 44   #include <avr/eeprom.h>
...
 86   void setContrast(CallbackType_t aCbType, void* s32)
 87   {
 88       /**
 89        * s32 points to an int32_t value handled by the editor
 90        * - as opposed to the value-address itself.
 91        **/
 92
 93       // for the contrast, we only need 8 bits from the editor value
 94       uint8_t val = ((S32MMSVal_t*) s32)->m_val;
 95
 96       // pressed-key is LHS-nibble of aCbType
 97       uint8_t key = aCbType & 0xF0;
 98
 99       // callbacktype is in RHS(right-hand-side)-nibble of aCbType
100       switch (aCbType & 0x0F) // only use RHS nibble
101       {
102           case CT_ENTRY:              // entry to to the editor: nothing to do
103               break;
104
105           case CT_CHANGE:             // the value was changed in the editor
106               if (key == BT_ENTER)    // value committed: update EEPROM
107                   eeprom_update_byte((uint8_t*)EEP(m_contrast), val);
108               else                    // there was a change with BT_UP or BT_DOWN
109                   lcdSetContrast(val); // 'live-update' the contrast
110               break;
111           case CT_RETURN:             // editor is about to be closed
112               lcdSetContrast(val);
113               break;
114       } // switch
115   } // setContrast
116
117   const U8MMValPCb_t Contrast PROGMEM = {
118       .m_min = 28,
119       .m_max = 50,
120       .m_dataP = (uint8_t*) EEP(m_contrast),
121       .m_callback = &setContrast,
122   }; // Contrast
...
133   const Menu_t TopMenu PROGMEM = {
134       .m_entryCount = 6,
135       .m_entries = {
136           { TX_Copyright,       FUNC(Copyright) },
137           { TX_SubMenu,         MENU(InfoMenu) },
138           { TX_EepromVersionRo, U8PER(EEVersion) },
139           { TX_EepromVersion,   U8PE(EEVersion) },
140           { TX_SetContrast,     U8MMPEC(Contrast) },
141           { TX_ScaledVoltage,   VOID(0) },
142       }, // m_entries
```

```
143   }; // TopMenu
144
145   int main(void)
146   {
147       _delay_ms(100);  // wait for voltages and levels to settle
148       lcdInit();
149       BUTTON_INIT();
150       while(1)
151           displayMenu(&TopMenu, BT_ALL);
152   } // main
```

The changes are straight-forward:

In line 140, the callback type was changed to `U8MMPEC`. The entry now points to the redefined `Contrast` entity. The changes are visible in line 112: whilst the `m_min`, `m_max` and the `m_dataP` entries haven't changed, they now belong to a `U8MMValPCb_t` type which provides a new `CallbackType_t` member named `m_callback`, which points to our `setContrast(..)` function. This function is coded in lines 86-115. Since this function writes to the EEPROM, we need an additional include file in line 44: `avr/eeprom.h`.

Have a look at the `setContrast(..)` function: Whenever the editor for this value is called, the `setContrast` function is triggered

- upon editor-start: CT_ENTRY
- upon each value change: CT_CHANGE
- and when the editor is closed: CT_RETURN

The key which was pressed in the editor prior to the particular callback-execution is coded in the left-hand-side nibble of `aCbType`, whilst the callback-'time' (`CT_ENTRY` etc.) is coded in the right-hand-side nibble of `aCbType`.

Compile and run the program and see how the display-contrast changes synchronously to editing. Watch the value and the contrast when you 'RETURN' from the editor and when you end the editor with ENTER.

6.2.5 `U8MMSPE`: Edit a 'scaled' `uint8_t` Value in EEPROM

Like `U8MMPE`, this type lets you edit a `uint8_t` value in EEPROM. There is basically no difference in storage and utilization, except that the *displayed* value on the LCD-screen is 'scaled' with a decimal point. For that reason, the underlying type `U8MMSValP_t` contains an additional `m_scale` member which defines the 'scale'. For example, with `m_scale` set to '`n`', the value is displayed with '`n`' decimal digits:

> Value: `123`, scale: `2`, display: `1.23`

NOTE 1: The 'scale' feature only applies to the *display* and to the *editing* of the value. The value itself remains a `uint8_t` integer which stores e.g. a 'scaled' `12.5` value on the display as the integer value `123`.

NOTE 2: The min/max settings of this type relate to the unscaled integer value. i.e, if the `m_max` value in the struct is set to `123`, the maximum value on the display will be `1.23` if `m_scale` is set to `2`.

In our example, we add a voltage-alarm value to the configuration. It shall be settable withing the range of 5.1 to 12.5 volts.

Example 10:

```
...
 58   typedef struct EEProm
 59   {
 60       uint8_t EEStart; // magic byte
 61       uint8_t m_contrast;
 62       uint8_t m_voltageAlarm;
 63   } EEProm_t;
...
112   const U8MMSValP_t Voltage PROGMEM = {
113       .m_min = 51,
114       .m_max = 125,
115       .m_scale = 1,
```

```
116         .m_dataP = (uint8_t*) EEP(m_voltageAlarm),
117     };
...
141     const Menu_t TopMenu PROGMEM = {
142         .m_entryCount = 6,
143         .m_entries = {
144             { TX_Copyright,        FUNC(Copyright) },
145             { TX_SubMenu,          MENU(InfoMenu) },
146             { TX_EepromVersionRo, U8PER(EEVersion) },
147             { TX_EepromVersion,    U8PE(EEVersion) },
148             { TX_SetContrast,      U8MMPEC(Contrast) },
149             { TX_ScaledVoltage,    U8MMSPE(Voltage) },
150         }, // m_entries
151     }; // TopMenu
152
153     int main(void)
154     {
155         _delay_ms(100);  // wait for voltages and levels to settle
156         lcdInit();
157         BUTTON_INIT();
158         while(1)
159             displayMenu(&TopMenu, BT_ALL);
160     } // main
```

Again, we've added another entry type to the menu in line 149: The `U8MMSPE` entry type points to a struct of type `U8MMSValP_t` names `Voltage which` coded in lines 112-117 and points to the corresponding EEPROM value added in line 62.

The `U8MMSP_t` type is very similar to the `U8MMValP_t` type which was used in the first `Contrast`-example. However, this type contains an additional `m_scale` member which defines the number of decimal positions to be displayed on the screen and when the value is edited.

NOTE: these “decimal positions” do not change the value type by means of a floating point value. They only provide a more user-friendly interface to values which are still integers within the application, but are supposed to *represent* a value with a fractional part from a users point of view.

This means: If you want to provide a dialog to edit a value within e.g. a range of 5.1 and 12.5, but these values are integers within your application, you can define a 'scale' of one together with min/max values of 51 and 125 respectively. This is exactly what you should see when you compile and run the application.

6.3 16 Bit, 32 Bit and Signed Variants

Everything written before applies similarly for 16 bit, 32 bit and signed variants of the above types. However, there are minor artifacts which keep their `uint8_t` types even in 16 or 32 bit type-variants, of which indexes for text-lists and scaling factors are the first which come spontaneously to my mind.

In case of any blur just have a look into `<emx/Types.h>`.

7 Advanced Types

So far, EMX allows you to rapidly set up an application which contains one ore more menus, sub-menus, various fonts, value-types, editing capabilities, and their storage in the EEPROM of your hardware.

However, sometimes an abstraction is wanted for configuration values which are not self-explaining by their nature: 12.1 Volts is a clear representation for what it is meant to describe. But what's meant with 'Protocol type '0', '1' oder '2' ?

Such values are abstractions for algorithmic processing and much more difficult to remember unless they can be 'named' in some way. This is where text-list-types come in.

7.1 Textual Value Representation

Lets assume a configuration which allows to select one protocol out of the ones which the application can handle, as illustrated in the table below:

EEPROM value	Displayed text upon editing
0	Futaba
1	Spectrum
2	Graupner
3	Jeti
4	Multiplex

This can already be done with a `U8MMPE` type, with its min/max values set to '0' and '4' respectively. But instead of having to document the meaning of each protocol-number and to over and over again answer this question in Internet-forums, you can comfortably give 'names' to these protocols with EMX: The user will not even see the numeric equivalent, but just chose between `Futaba Spectrum`, and so on.

7.1.1 `ITLPE`: Text-Based Value Display and Editing

The `ITLPE` entry type allows such a textual representation. The acronym stands for ***I**ndexed-**T**ext-**L**ist **P**ointer to **E**EPROM*-value. Here's how it works:

Example 11:

```
...
 53   const char TX_ScaledVoltage[]    PROGMEM = "Voltage alarm";
 54   const char TX_Protocol[]         PROGMEM = "Protocol";
 55
 56   const char TX_Futaba[]           PROGMEM = "Futaba";
 57   const char TX_Spectrum[]         PROGMEM = "Spectrum";
 58   const char TX_Graupner[]         PROGMEM = "Graupner";
 59   const char TX_Jeti[]             PROGMEM = "Jeti";
 60   const char TX_Multiplex[]        PROGMEM = "Multiplex";
 61
 62   const char TX_EmxLogo[]          PROGMEM = "Show EMX logo";
 63   const char TX_License[]          PROGMEM = "Show license";
 64
 65   typedef struct EEProm
 66   {
 67       uint8_t EEStart; // magic byte
 68       uint8_t m_contrast;
 69       uint8_t m_voltageAlarm;
 70       uint8_t m_protocol;
 71   } EEProm_t;
...
127   const ITextListP_t Protocol PROGMEM = {
128       .m_entryCount = 5,
129       .m_idxP       = (uint8_t*) EEP(m_protocol),
130       .m_texts      = {
131           TX_Futaba,
132           TX_Spectrum,
133           TX_Graupner,
```

```
134             TX_Jeti,
135             TX_Multiplex
136         },
137     };
...
160     const Menu_t TopMenu PROGMEM = {
161         .m_entryCount = 6,
162         .m_entries = {
163             { TX_Copyright,        FUNC(Copyright) },
164             { TX_SubMenu,          MENU(InfoMenu) },
165             { TX_EepromVersionRo, U8PER(EEVersion) },
166             { TX_EepromVersion,    U8PE(EEVersion) },
167             { TX_SetContrast,      U8MMPEC(Contrast) },
168             { TX_ScaledVoltage,    U8MMSPE(Voltage) },
169             { TX_Protocol,         ITLPE(Protocol) },
170         }, // m_entries
171     }; // TopMenu
172
173     int main(void)
174     {
175         _delay_ms(100);  // wait for voltages and levels to settle
176         lcdInit();
177         BUTTON_INIT();
178         while(1)
179             displayMenu(&TopMenu, BT_ALL);
180     } // main
```

Let's walk through the code: as usual, we've added a new menu-entry in line 169, an `ITLPE` type as described above. It points to the corresponding `ITextListP_t` struct coded in lines 127-137. This struct contains

- the count of choices for this value: `5`
- a pointer to the index, i.e. the value to be configured: `m_protocol`.
- the texts to represent each choice. The corresponding values which are stored by the text-list-editor are `0 - (n-1)` where `n` is the number of choices defined in `m_entryCount`.

If you let aside the `PROGMEM`-texts and the EEPROM-variable, there remain just three values to define, that's all. Compile, run and enjoy!

BTW: Your menu doesn't show the 'Protocol' item? OK, control your `TopMenu.m_entryCount` value ;-)

7.2 Dynamic Value Processing

Sometimes, an array of values of congeneric type and semantics can be boring to code. But with the above entry types you can can only deal with fixed EEPROM addresses and not with variable ones which is uncomfortable in case that you have repeating values of an array which is located in the EEPROM: An own menu-entry for each occurrence of a single array element is bulky and awkward. It's furthermore a waste of flash-memory and on top of that it can be error-prone.

EMX offers a smart solution to handle such situations: The *dynamic* value types.

7.2.1 `U8MMDPE`: Dynamic Address Calculation Upon Runtime, Array Processing

A dynamic value type provides a way to obtain the address of a value upon runtime. The following example implements a use-case for an array of PID-value configurations, each of which constituting a separate 'profile'.

Just imagine an application which allows to define a set of profiles, each of which containing a bunch of different values and value-types, but all of them the same by means of the collection of values they contain.

Example 12:

```
...
65  const char TX_ProfileNo[]       PROGMEM = "Profile No.";
66  const char TX_PValue[]          PROGMEM = "P-Value";
67  const char TX_IValue[]          PROGMEM = "I-Value";
68  const char TX_DValue[]          PROGMEM = "D-Value";
69
70  typedef struct
71  {
```

```
 72      uint8_t          m_curveType;
 73      uint8_t          m_pValue;
 74      uint8_t          m_iValue;
 75      uint8_t          m_dValue;
 76  } Profile_t;
 77
 78  typedef struct EEProm
 79  {
 80      uint8_t EEStart; // magic byte
 81      uint8_t m_contrast;
 82      uint8_t m_voltageAlarm;
 83      uint8_t m_protocol;
 84      uint8_t m_curProfile;
 85      Profile_t m_profiles[3];
 86  } EEProm_t;
...
 91  #define EEP(name) ((uint8_t*) ((uint8_t*) &EEProm.name - &EEProm.EEStart))
 92  // return the offset of a variable in Profile_t array
 93  #define EEP_PRV(name) (EEP(m_profiles[0].name) - EEP(m_profiles[0]))
...
130  void* getPrfDynDataP(void* aAdressOffset)
131  {
132      // the aAdressOffset value contains the offset from
133      // EEProm.m_profiles[0] to the Profile-variable in question.
134      // To obtain this offset, the macro EEP_PRV(varname) is used.
135      // EEP_PRV -> EepromPointer to Profile-Variable
136
137      // get the current profile
138      uint8_t curProf = eeprom_read_byte(EEP(m_curProfile));
139      // return the corresponding variable address
140      return aAdressOffset + ( (int16_t) EEP(m_profiles[curProf-1])); // profiles 0,1,2 are\
                                                                          coded 1,2,3
141  }
...
157  // dynamic pointers
158  const U8MMValDP_t PValue PROGMEM = {
159      .m_min = 1,
160      .m_max = 100,
161      // this value is passed to getDynDataP (see GetDynDataP function)
162      .m_dynDataInfo = (uint8_t*) EEP_PRV(m_pValue),
163      .m_callback = &getPrfDynDataP
164  };
165  const U8MMValDP_t IValue PROGMEM = {
166      .m_min = 0,
167      .m_max = 127,
168      // this value is passed to getDynDataP (see GetDynDataP function)
169      .m_dynDataInfo = (uint8_t*) EEP_PRV(m_iValue),
170      .m_callback = &getPrfDynDataP
171  };
172  const U8MMValDP_t DValue PROGMEM = {
173      .m_min = 20,
174      .m_max = 80,
175      // this value is passed to getDynDataP (see GetDynDataP function)
176      .m_dynDataInfo = (uint8_t*) EEP_PRV(m_dValue),
177      .m_callback = &getPrfDynDataP
178  };
179
180  const U8MMValP_t CurProfile PROGMEM = {
181      .m_min = 1,
182      .m_max = 3,
183      .m_dataP = (uint8_t*) EEP(m_curProfile),
184  };
185
...
219  const Menu_t TopMenu PROGMEM = {
220      .m_entryCount = 11,
221      .m_entries = {
222          { TX_Copyright,       FUNC(Copyright) },
223          { TX_SubMenu,         MENU(InfoMenu) },
224          { TX_EepromVersionRo, U8PER(EEVersion) },
225          { TX_EepromVersion,   U8PE(EEVersion) },
226          { TX_SetContrast,     U8MMPEC(Contrast) },
227          { TX_ScaledVoltage,   U8MMSPE(Voltage) },
```

```
228           { TX_Protocol,        ITLPE(Protocol) },
229           { TX_ProfileNo,       U8MMPE(CurProfile) },
230           { TX_PValue,          U8MMDPE(PValue)   },
231           { TX_IValue,          U8MMDPE(IValue)   },
232           { TX_DValue,          U8MMDPE(DValue)   },
233       }, // m_entries
234   }; // TopMenu
235
236   int main(void)
237   {
238       _delay_ms(100);  // wait for voltages and levels to settle
239       lcdInit();
240       BUTTON_INIT();
241       while(1)
242           displayMenu(&TopMenu, BT_ALL);
243   } // main
```

Though the changes look laborious, it's more a matter of repetitions than of complexity. But lets work through it step by step:

- The menu now provides a '`Profile No.`' choice, where you can select the profile in question. Since we only allow for three profiles, this entry limits the input from 1 to 3. As you will see later, this does not affect the real profile number, which likely starts with '0' instead of '`1`': Users do not think in developers categories, so I've decided to display profile number '`0`' as number '`1`', profile number '`1`' as number '`2`' and so on.

  The entity in question is `CurProfile`, coded in lines 180-184. The corresponding menu text is defined in line 65.

- There are three more menu positions, each of which allowing to edit one of the P/I/D values. Though there are three instances for each of these items, they appear only once. The important part is the type `U8MMDPE`. The P-Value item for example points to the `PValue` entity which is a `U8MMValDP_t` type, which is very similar to the `U8MMValP_t` type , but with an additional 'D'ynamic component. So the structure contains the members:

  - `m_min` : the minimum value which is allowed for editing.
  - `m_max` : the maximum value which is allowed for editing.
  - `m_dynDataInfo` : the offset of the variable address within the current profile
  - `m_callback` : the function which delivers the actual address

  `m_min` and `m_max` do not need further explanation, I guess.

- The `m_dynDataInfo` value contains the offset from the beginning of the *current* Profile to the variable in question. This offset never changes since it is of course always the same in each particular occurrence of the profile-array.

  Finally, the `m_callback` member points to the function which is responsible for the calculation of the address which the variable editor needs to know.

  For each of the three values, `P`, `I` and `D`, there is an own `U8MMValDP_t` block, all off which are coded in lines 157-178. Each block defines own min/max values, and each block defines a different offset as each variable has a different address within its containing profile.

  The `m_dynDataInfo` value is calculated by the `EEP_PRV` macro in line 93 which should not be too difficult to understand after having used the `EEP` macro earlier in the other examples. Upon displaying/editing the variable, EMX passes this value to the function named in the `m_callback` member.

- Lets finally look at the `getPrfDynDataP(..)` function in lines 130-141.

  This function is of course not an EMX function, but an application specific implementation. The function first reads the current profile number in line 138 from the EEPROM storage. With that number, the address of the variable in question is calculated in line 140 and returned to the caller - which is the EMX display/editing module.

In line 140, you can see how the profile numbers 0, 1 and 2 are mapped to the user-interface as numbers 1, 2 and 3.

Compile the program and see how it works: Whenever you change the Profile number on the display, each subsequent value which belongs to that profile is changed automatically on the display as well. You do not have to re-read the new profiles values, EMX does it all for you.

NOTE: it's up to you to preset the EEPROM accordingly. Invalid values can cause unpredictable results, so before posting bugs, check your EEPROM-status, e. g. with

```
emax@hal:~$ avrdude -p m324pa -c jtag2isp -P usb -U eeprom:r:eedump.bin:r
```

and use a hex-editor to examine the content of eedump.bin. BTW: this is *my* command to read the EEPROM, your's may have to be different.

7.2.2 `ITLDPE`: Text Based Display/Editing of Array-Values

Like the `U8MMDPE` type described before, the `ITLDPE` text based menu entry type allows to edit values dynamically, i.e.: array values whose addresses vary upon runtime. Display and editing is done text-based, as described before for `ITLPE` entries. The following example extends the P/I/D example with a 'Curve-Type' value.

NOTE: Since we again use a 'virgin' EEPROM-value, you should update EEPROM with something plausible before you try this example.

Example 13:

```
...
 70   const char TX_Curve[]           PROGMEM = "Curve";
 71   const char TX_Flat[]            PROGMEM = "Flat";
 72   const char TX_Normal[]          PROGMEM = "Normal";
 73   const char TX_Steep[]           PROGMEM = "Steep";
 74
 75   typedef struct
 76   {
 77       uint8_t          m_curveType;
 78       uint8_t          m_pValue;
 79       uint8_t          m_iValue;
 80       uint8_t          m_dValue;
 81   } Profile_t;
...
191   const ITextListDP_t Curve PROGMEM = {
192       .m_entryCount  = 3,
193       .m_dynDataInfo = (uint8_t*) EEP_PRV(m_curveType),
194       .m_callback    = &getPrfDynDataP,
195       .m_texts       = {
196           TX_Flat,
197           TX_Normal,
198           TX_Steep,
199       },
200   };
201
202   const ITextListP_t Protocol PROGMEM = {
203       .m_entryCount =5,
204       .m_idxP       = (uint8_t*) EEP(m_protocol),
205       .m_texts      = {
206           TX_Futaba,
207           TX_Spectrum,
208           TX_Graupner,
209           TX_Jeti,
210           TX_Multiplex
211       },
212   };
213
214   uint8_t Copyright(MenuState_t* aMs)
215   {
216       LCD_BUF_CLEAR();
217       lcdBufPuts(AT_PGM, PSTR("menu system,\n"
218                               "framework and font\n"
219                               "2013-2014 by\n"
220                               "Edgar (emax) Hermanns"),
```

```
221                 &Verdana11, 0, 0, COL_BLACK);
222       lcdUpdate(1);
223       pollButtons(0xFF, 1);
224       return 1;
225   }
226
227   const Menu_t InfoMenu PROGMEM = {
228       .m_entryCount = 2,
229       .m_entries = {
230           { TX_EmxLogo,       VOID(0) },
231           { TX_License,       VOID(0) },
232       }, // m_entries
233   }; // InfoMenu
234
235   const Menu_t TopMenu PROGMEM = {
236       .m_entryCount = 11,
237       .m_entries = {
238           { TX_Copyright,       FUNC(Copyright) },
239           { TX_SubMenu,         MENU(InfoMenu) },
240           { TX_EepromVersionRo, U8PER(EEVersion) },
241           { TX_EepromVersion,   U8PE(EEVersion) },
242           { TX_SetContrast,     U8MMPEC(Contrast) },
243           { TX_ScaledVoltage,   U8MMSPE(Voltage) },
244           { TX_Protocol,        ITLPE(Protocol) },
245           { TX_ProfileNo,       U8MMPE(CurProfile) },
246           { TX_Curve,           ITLDPE(Curve) },
247           { TX_PValue,          U8MMDPE(PValue)  },
248           { TX_IValue,          U8MMDPE(IValue)  },
249           { TX_DValue,          U8MMDPE(DValue)  },
250       }, // m_entries
251   }; // TopMenu
252
253   int main(void)
254   {
255       _delay_ms(100);  // wait for voltages and levels to settle
256       lcdInit();
257       BUTTON_INIT();
258       while(1)
259           displayMenu(&TopMenu, BT_ALL);
260   } // main
```

The additional menu entry in line 246 points again to a member of the `m_profiles` array. It is an `ITLDPE` type pointing to the `ITextListPD_t` structure coded in lines 191-200.

Let's have a look at the members of this struct:

- `m_entryCount` is nothing new. It is the same as in the `ITextListP_t` type and contains the number of choices provided by this entity.
- `m_dynDataInfo` is already known from the `U8MMDPE` example and contains the offset of the value in question, which is in this case the index of the text-list. This value is the parameter for the function defined in the
- `m_callback` member. As in the `U8MMDPE` example, we can use the `getPrfDynDataP(..)` function to obtain the value-address for the text-list index.
- `m_texts` finally holds the texts to be displayed upon displaying/editing this value.

I'd dare say, that everything is pretty straight forward. In the last two examples, we could use the same function to obtain the address of any variable within the `m_profiles[3]` array. This is so, because all such variable-addresses are relative to the same base-address, namely `EEProm.m_profiles[n]`, where '`n`' is the number of the active profile.

NOTE: That way, you can address any variable in an array, no matter which type it has. You can as well mix different types in an array and display/edit their values at will.

8 Application Control

In an embedded application, total control of the program-flow is indispensable. Whilst a callback driven user interface shouldn't be a problem, the application itself will likely want to run freely, outside of any framework.

This is of course no problem with EMX, and it is already obvious how this can be achieved: When the `displayMenu(..)` call in `main()` function returns, EMX is completely out of the game and the application-developer can run any services at will.

However, this this is not a satisfying solution if the developer wants to get back control immediately from within any active menu, without having to walk back to the main function with repeated 'BACK' button presses.

For that reason, EMX allows the application to completely jump out from any menu or menu-called function to the up-most level from where the first EMX call was made. This is managed by the return code which a user-function gives back to the EMX-framework.

8.1 Menu Return Codes: Governing the EMX Framework

The following example shows how to control the EMX-framework. The applications implements a clock which is only visible whilst the program is running outside EMX in the main-loop. To demonstrate the role of a function-return-code in EMX, the clock can only be started and stopped from within the submenu which was already implemented in an example before. There is menu-choice to start and to stop the clock. To get back directly from the sub-menu to the main-loop, the sub-menu now provides an “Exit EMX” entry.

You can of course leave a menu at any time from any function without having an extra menu-choice for this. This entry is only for demonstration purposes, the underlying function just returns with an MR_EXIT return code.

Example 14:

```
...
 45   #include <avr/interrupt.h>
...
 76   const char TX_RunClock[]         PROGMEM = "Run Clock";
 77   const char TX_StopClock[]        PROGMEM = "Stop Clock";
 78   const char TX_ExitEMX[]          PROGMEM = "Exit EMX";
 79   const char TX_Started[]          PROGMEM = "Clock started";
 80   const char TX_Stopped[]          PROGMEM = "Clock stopped";
 81   const char TX_Menu[]             PROGMEM = "Menu";
...
235   volatile uint8_t g_runClock = 0;
236
237   uint8_t RunClock(MenuState_t* aMs)
238   {
239       g_runClock = 1;
240       LCD_BUF_CLEAR();
241       lcdBufPuts(AT_PGM, TX_Started, &Verdana19, 10, 25, COL_BLACK);
242       lcdUpdate(1);
243       pollButtons(BT_ALL, BF_DELAY);
244       return 1;
245   } // RunClock
246
247   uint8_t StopClock(MenuState_t* aMs)
248   {
249       g_runClock = 0;
250       LCD_BUF_CLEAR();
251       lcdBufPuts(AT_PGM, TX_Stopped, &Verdana19, 10, 25, COL_BLACK);
252       lcdUpdate(1);
253       pollButtons(BT_ALL, BF_DELAY);
254       return 1;
255   } // StopClock
256
257   uint8_t ExitEMX(MenuState_t* aMs)
258   {
259       return MR_EXIT;
260   } // ExitEMX
261
262   const Menu_t InfoMenu PROGMEM = {
263       .m_entryCount = 5,
264       .m_entries = {
265           { TX_EmxLogo,       VOID(0) },
```

```
266           { TX_License,       VOID(0) },
267           { TX_RunClock,      FUNC(RunClock) },
268           { TX_StopClock,     FUNC(StopClock) },
269           { TX_ExitEMX,       FUNC(ExitEMX) },
270       }, // m_entries
271   }; // InfoMenu
...
292   volatile uint8_t g_hrs;      // 0-255
293   volatile uint8_t g_min;      // 0-59
294   volatile uint8_t g_sec;      // 0-59
295   volatile uint8_t g_sec100th; // 0-99, 100th
296   volatile uint8_t g_04ms;     // 0.4ms
297
298   char clock[] = "00:00:00.00";
299
300   void showClock(void)
301   {
302       // display the time
303       lcdBufFillRect(0, 0, LCDWIDTH, LCDHEIGHT/2, COL_WHITE);
304       lcdBufPuts(AT_RAM, clock, &Verdana19, 17, 20, COL_BLACK);
305   }
306
307   void initClockTimer(void)     // timer 0 settings
308   {
309       TCCR0A = (1<<WGM01); // CTC mode
310       TCCR0B |= (1 << CS01) | (1 << CS00); // prescaler 64
311       OCR0A = 124; // count (((0..124)=125 cycles) * prescaler) / F_CPU = 0.0004sec
312       TIMSK0 |= (1<<OCIE0A);     // allow compare interrupt
313   } // initClockTimer
314
315   ISR (TIMER0_COMPA_vect)
316   {
317       if (g_runClock) {
318           if (g_04ms < 24)            ++g_04ms;
319           else
320           {
321               g_04ms = 0;
322               if (g_sec100th < 99)    ++g_sec100th;
323               else {
324                   g_sec100th = 0;
325                   if (g_sec < 59)     ++g_sec;
326                   else {
327                       g_sec = 0;
328                       if (g_min < 59) ++g_min;
329                       else {
330                           g_min = 0;
331                           ++g_hrs;
332                           clock[0] = (g_hrs/10) + '0';
333                           clock[1] = (g_hrs%10) + '0';
334                       }
335                       clock[3] = (g_min/10) + '0';
336                       clock[4] = (g_min%10) + '0';
337                   }
338                   clock[6]  = (g_sec/10) + '0';
339                   clock[7]  = (g_sec%10) + '0';
340               }
341               uint8_t sec10th = g_sec100th/10;
342               clock[9]  = (sec10th) + '0';
343               clock[10] = (g_sec100th%10) + '0';
344           }
345       }
346   } // ISR (TIMER0_COMPA_vect)
347
348   int main(void)
349   {
350       _delay_ms(100);  // wait for voltages and levels to settle
351       initClockTimer();
352       sei();
353       lcdInit();
354       BUTTON_INIT();
355       while(1)
356       {
357           displayMenu(&TopMenu, BT_ALL);
```

```
358
359             while (pollButtons(BT_ALL, BF_NOBLOCK) != BT_BACK)
360             {
361                 LCD_BUF_CLEAR();
362                 showClock();
363                 lcdBufPuts(AT_PGM, TX_Menu, &Verdana11, 0, 50, COL_BLACK);
364                 lcdUpdate(1);
365             }
366         };
367     } // main
```

When you run the program, you first see the menu as you did before. If you enter the submenu and select one of the "Run Clock" or "Stop clock" entries, the appropriate action is taken and a confirmation is written on the display.

You can now leave the menu with the BACK-button as before, or select "Exit EMX" instead. The difference is obvious: With the BACK button, you navigate back through the previous menu, but with the "Exit EMX" choice you jump completely out of EMX directly to the location where EMX was entered, which is in this case the main-loop. The next statement to be executed then in line 359.

Selecting "Exit EMX" calls the ExitEMX() function which just gives back the MR_EXIT value to the menu system. MR_EXIT is a 0x80 value which can be binary or-ed with the regular return value of any function which was called by the menu system.

NOTE: Bits 7 and 6 of the return-codes have a special meaning in the EMX menu-system :

- **0x80** is the MR_EXIT bit demonstrated here.
- **0x40** is the so called MR_REPEAT flag and causes the called function to be called again immediately until the function returns with this bit cleared.

MR_EXIT has precedence over MR_REPEAT.

You are free to use the lower 6 bits of the return-code. All 8 bits of the return-code remain untouched are and given back to the up-most level from where EMX was called.

Lets have a final look at the source: In lines 359-365 the showClock() function is called repeatedly until pollButtons return the BT_BACK value. With the BF_NOBLOCK flag passed to pollButtons(..) in line 359, the loop is not blocked since with this flag, pollButtons returns immediately, regardless whether a button was pressed or not.

The rest of the source should be self-explaining, the additional and changed lines are marked blue as usual.

BTW: That little clock turned out to be quite precise on my hardware. After about 10 hours or so it was still spot on – as far as I could verify visually.

9 Debugging Aid

EMX offers a visual debugging aid. To make use of the debugging aid, you have to `#include` the file `<emx/Config.h>` .

The prototype of the debug function is

```
uint8_t DebugHex(AdrType_t aAdrType, const char* aLbl,
      const void* aPtr, uint8_t aLen);
```

and allows to debug RAM and FLASH addresses and values and the free memory. The parameters are

- `aAdrType`: the address type which can be `AT_RAM` for RAM-values or `AT_PGM` for flash values.
- `aLbl`: a text-label which is displayed on the debug screen
- `aPtr`: the address to be debugged
- `aLen`: the number of bytes to be displayed.

However, I recommend to use the debugging macros provided by EMX. They are written in that way, that debug-commands are not compiled in unless you define `DEBUG_CODE` upon compiler invocation. Using the macros does thus ignore any debugging-code by simple omitting this definition.

There are currently three macros implemented:

- `DBR(label, entity)`: debug a RAM-address. calls

  ```
  DebugHex(AT_RAM, label, &entity, sizeof(entity))
  ```

- `DBP(label, entity)`: debug a flash address. calls

  ```
  DebugHex(AT_RAM, label, &entity, sizeof(entity));
  ```

- `DBF(label)`: debug free RAM

  ```
  {uint16_t __freeRam=SP-(uint16_t) &__heap_start; DBR(a, __freeRam);}
  ```

When using the debugging aid, the following rules apply:

- Debug utilizes the lower half of the display. This area is cleared before debug-information is displayed.
- Debug does not restore the display, nor does it clear the display after debug information was displayed.
- Debug does not utilize, stop or change interrupts
- Debug displays all values in hex.
- All addresses are displayed byte by byte, up to the `aLen` length which was passed to the debug function.
- Since debug does process all data byte by byte, no endianess is regarded. atmega processors have little endian architectures, so the value 1024 is not stored as 0x0400 but instead with the 'little' byte first: 0x0004, and displayed by debug in that order.
- If free RAM is debugged, debug does not consider own RAM consumption. i.e.: the displayed free-RAM shows in fact too little free RAM because debug uses ram as well.

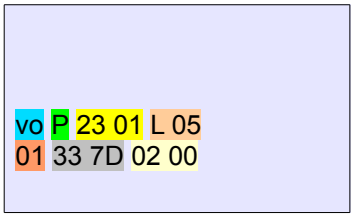
If you have a look at the compiler invocation for the first example, you will see the "**`-DDEBUG_CODE`**" parameter there which must be set if you want to debug with the macros described above. Without this parameter, all debug macros are ignored (i.e.: you do not have to remove debugging statements later if you compile for production).

Example 15:

```
...
 38   #include <emx/Config.h>
...
```

```
144   void* getPrfDynDataP(void* aAdressOffset)
145   {
146       // the aAdressOffset value contains the offset from
147       // &Eeconfig.m_fc[0] to the FC-variable in question.
148       // To obtain this offset, the macro EEP_FCV(varname) is used.
149       // EEP_PRV -> EepromPointer to Profile-Variable
150       DBR("aO", aAdressOffset);
151       DBF("ram");
152       // get the current profile
153       uint8_t curProf = eeprom_read_byte(EEP(m_curProfile));
154       // return the corresponding variable address
155       return aAdressOffset + ( (int16_t) EEP(m_profiles[curProf-1])); // ...
156   }
157
...
165   const U8MMSValP_t Voltage PROGMEM = {
166       .m_min = 51,
167       .m_max = 125,
168       .m_scale = 1,
169       .m_dataP = (uint8_t*) EEP(m_voltageAlarm),
170   };
...
350   int main(void)
351   {
352       _delay_ms(100);  // wait for voltages and levels to settle
353       initClockTimer();
354       sei();
355       lcdInit();
356       BUTTON_INIT();
357
358       DBP("vo", Voltage);
359       while(1)
360       {
361           displayMenu(&TopMenu, BT_ALL);
362           while (pollButtons(BT_ALL, BF_NOBLOCK) != BT_BACK)
363           {
364               LCD_BUF_CLEAR();
365               showClock();
366               lcdBufPuts(AT_PGM, TX_Menu, &Verdana11, 0, 50, COL_BLACK);
367               lcdUpdate(1);
368           }
369       };
370   } // main
```

When you compile an run the program with the `-DDEBUG_CODE` parameter set, you'll first hit the debug call in line 358. The debug output looks somewhat like this:



And here is what you see:

- `vo` : the label which you passed to the debug call
- `P` : tells you that you're looking at a `PROGMEM` entity (address type `AT_RAM`)
- `23 01`: the address of the entity in atmega byte-order. So the correct address is `0x0123 = 291`(dec). This is the address of the entity within the binary code of your program.
- `L 05`: the length of the debug output is 5 bytes
- `01` the `m_scale` value of the `Voltage` data struct.
- `33 7D` the `m_min` and `m_max` value of the Voltage data struct, `51` (`0x33`) and `125` (`0x7D`)
- `02 00` the `EEProm` address in `m_dataP` in atmega byte-order. So the correct address is `0x0002 = 2`(dec).

You might be surprised that though `m_min` and `m_max` appear first in the Voltage struct, the `m_scale` value is displayed first. This is so, because it doesn't of course play any role, in which order these appear in the code, but instead in which order they are defined in the struct. And `m_scale` *is* the first member in that struct.

Within the debug display, any button is accepted. In the current example, If you press a button, you move on to the menu. Move the cursor down until you reach the "`Profile No.`" item. If you move the cursor down one more position, the menu is overlapped with another debug-output.

You can now examine what the address-offset value which was passed to the `getPrfDynDataP(..)` function is. You can as well see how much RAM is still unused, but keep in mind that the debug-function itself consumes RAM as well thus making the available ram appear smaller than it in fact is.

The free RAM display (which you can identify by the '`ram`' label which you passed to the `DBF` macro) shows you an address similar to the first yellow marked value above: This is the address of a temporary variable which is used to calculate the available RAM. This address is therefore meaningless.

NOTE: When debugging, don't forget to `#include <emx/Config.h>` as shown above.

10 More to Come

EMX has some more features which I wasn't yet able to write examples for. I must even admit, that I do not remember all features which I have built in (but can of course recover them). So this document will likely grow a bit, but for the time being I should provide a good starting point to use EMX.

10.1 EMX: Small, Medium, Large

If there is demand, I can eventually provide miscellaneous flavors of EMX: Currently, EMX can process 8, 16 and 32 bit values, and this is not always required. I think, this matter needs some explanation:

- Though EMX is designed to be as atomic as possible, there is a small compromise in the Display and Editing modules: If you write a function which can only process one single data-type, this function can be held very small. But if you need another function for another data-tape, e.g. 16 bit instead of 8, it costs more code to have two such small functions then to have one single function which can do it all.
- Unfortunately, the single can-do-it-all function wastes code in case that a developer doesn't need the extra functionality for processing multiple data-sizes.
- This dilemma is hard do master, but it could be a solution to build versions which only support exactly the subset of data-type which is really needed for a particular application.

But that's the future. Eventually.

Enjoy EMX!