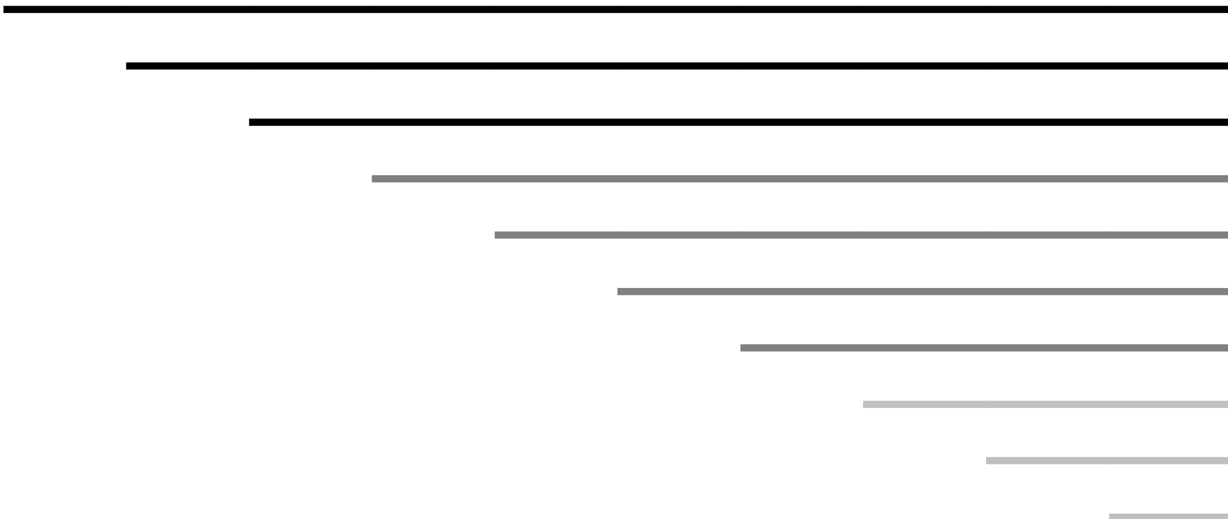
Embedding it better...

µTasker Document

Generic Display Menu



Copyright © 2021 M.J.Butcher Consulting

Table of Contents

1.Introduction..3
1.1 Firmware Files...3
1.2 Starting the Display Menu..3
1.3 Handling Key Presses...4
2.Menu Structure...5
1.4 Navigating in and out of Sub-Menus..8
3.Viewing System Parameters...9
1.5 Text Languages...12
4.Graphical Representation of Parameter Values..14
5..16
6.Conclusion...17
Appendix A..19
a)…...19

1. Introduction

The µTasker uGLCDLIB offers a platform to users of various display technologies to efficiently create products integrating character or graphic based displays. Such products often have the requirement to use this interface for controlling the setup of the equipment and viewing its parameters.

This document discusses a generic menu control utility that is part of the µTasker project which allows a menu driven interface for such purposes that can essentially be used in a compatible manner on all type displays, whether character or graphic based. This document shows the operation on a mono-chrome graphic display a reference. It uses mechanical buttons as inputs to control the menu (that is inputs that can be interpreted by the processor to be either pressed or released) but the same operation can be achieved by touch screen displays generating equivalent press/release events on touch areas. To ensure compatibility the menu control is based exclusively on single input press or release events and not on combinations of multiple buttons, although individual products may adapt the general details if there are advantages in doing so when the input capabilities exist.

When the µTasker project is configured to use a display the display menu support is enabled with the project defined (in `config.h`)

```
#define GLCD_MENU_SUPPORT
```

1.1 Firmware Files

The display menu operation is implemented in the application file `application_lcd.h`, which is the overall application's LCD functionality included in `application.c`.

The user defined menus and their interaction and control of specific project variables is configured in the file `display_menu.h`, which is included in `application_lcd.h`.

If images are used, which is an option in the display menu, these are contained in the file `widgets.h`, which is included in `application_lcd.h`. As explained later in this document the images are created by the µTasker utility `uTaskerFileCreate.exe` from standard bit maps. This procedure can be automatic to generate this file each time the project is built or else can be performed manually by executing the appropriate bat file (usually `widgets.bat` in the application directory). *Should there be an error when building the project due to the file `widgets.h` not being found, or due to images that `widgets.h` is expected to supply it probably means that the conversion utility step has been missed and should first be performed.*

1.2 Starting the Display Menu

By default the display menu is started when the LCD is ready (event `E_LCD_INITIALISED`) but it can be started at any time using the command (see later in the document for the definition of the user's menu (in this case `my_menu`)

```
fnDrawMenu(my_menu);                                // draw the top level menu
```

Once enabled, the display menu interface handles button events to autonomously control the overall operation of moving through menus, viewing and modifying system variables.

1.3 Handling Key Presses

This document doesn't focus on how key presses are controlled; they could be generated by scanning keyboards or from touch screen interfaces. These key changes result however in events that are passed to the display menu via the function

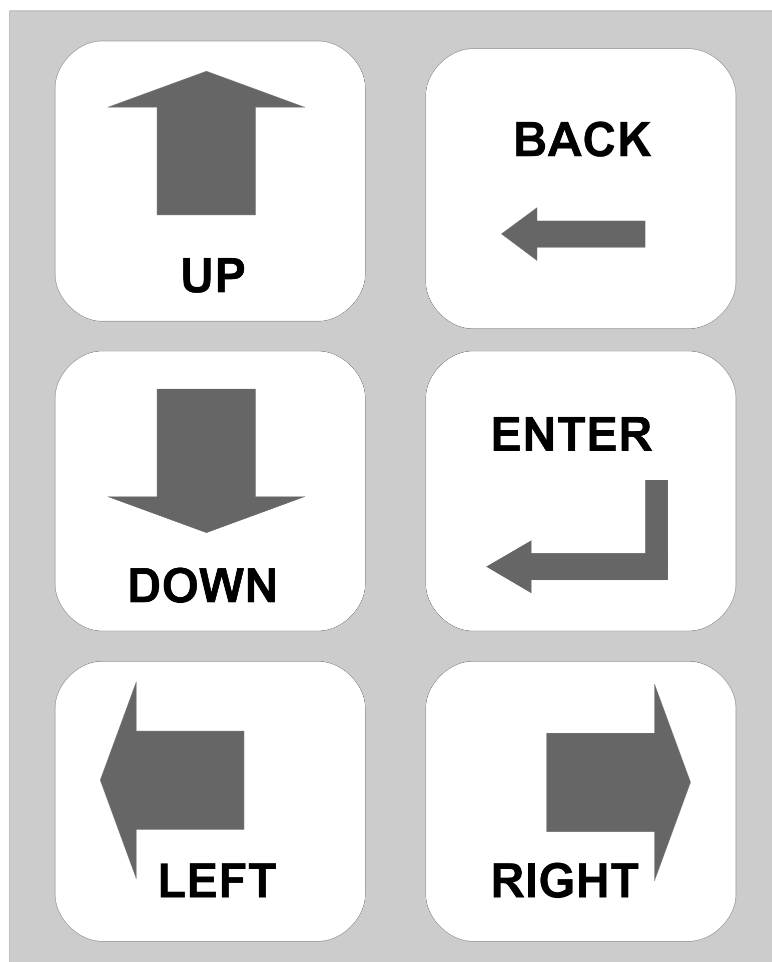
```
static void fnHandleKey(unsigned char ucKeyAction);
```

whereby a number of key action events are handled (event numbers can vary since they are defines)

```
#define MENU_BUTTON_UP_PRESSED        63
#define MENU_BUTTON_DOWN_PRESSED      64
#define MENU_BUTTON_ENTER_PRESSED     65
#define MENU_BUTTON_RETURN_PRESSED    66
#define MENU_BUTTON_LEFT_PRESSED      67
#define MENU_BUTTON_RIGHT_PRESSED     68
```

6 button events are required for full control of the menu, although less many be necessary if some menu control features are not required. Furthermore these events may be realised by combined actions of a more limited number of buttons causing them (such as holding one down for a certain duration or using one key as a function key to change the events generated by others).

A general keypad design with 6 keys generating individual events for the overall control is shown below which will be used as reference for illustrating the menu operation in this document.



2. Menu Structure

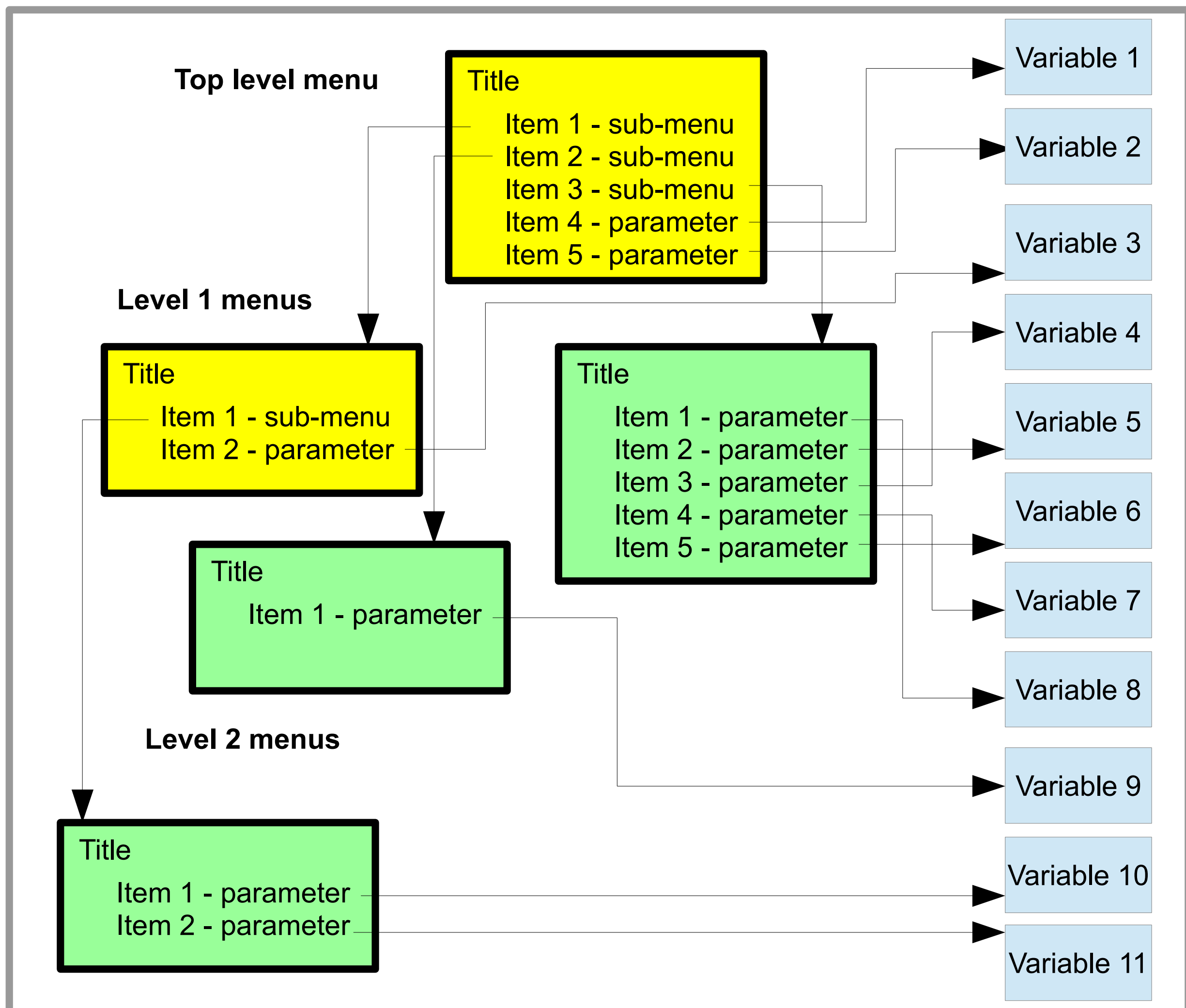
Menus are defined as follows:

– A menu has a single title, for example a menu may groups certain related control/display possibilities. This title is always visible.

– A menu has a certain number of items, whereby an item can be a sub-menu or a parameter: sub-menus are the titles of the next level down in the menu structure that again breaks down members of a menu group in to more specific sub-menus or items. A parameter allows viewing and modifying a particular value in the system. Not all items may be visible at the same time but can be located by 'scrolling' through the menu.

– A single menu is effectively unlimited in the number of items that it has but practically will tend to make use of submenu items to aid in efficiently navigating to parameter that the user is interested in.

The following diagram shows a typical menu structure that the the user can easily define, extend and modify during the course of a product development:



It is seen that there is always a top level menu and optionally further sub-level menus. A parameter item can placed in menus at any level and a lower level menus continue until there is a menu without any further sub-menu items.

The sub-menu depth that is required is defined by

```
#define LCD_MENU_LEVELS      3              // top level, sub-level and sub-sub-level
```

and can be set to whatever value is needed to achieve the sub-menu level for the project (*a value greater than the depth actually used is safe but a value smaller will restrict the depth obtainable*).

In order to define the top level menu structure as shown in the image the following code can be used in `display_menu.h`:

```
// Top menu
//
static const MENU_POSITION my_menu[] = {
    {"Main menu", 0,        0},                          // first entry is title

    {"Menu 1",    &subMenu1,0},                          // sub-menu item
    {"Menu 2",    &subMenu2,0},                          // sub-menu item
    {"Menu 3",    &subMenu3,0},                          // sub-menu item
    {"Language",  0,        &varLanguageDef},            // parameter item
    {"Contrast",  0,        &varContrastDef},            // parameter item
    {0}                                                  // end of items
};
```

The sub-level menus are defined in a similar manner, whereby these are best defined *before* the top level menu in the header file so that no prototypes are required for then.

```
// Level 1 menus
//
static const MENU_POSITION subMenu1[] = {
    {"Sub-menu 1", 0,        0},                         // first entry is title

    {"Menu 1-1",  &subMenu1_1,0},                        // sub-menu item
    {"Brightness",0,         &varBrightnessDef},         // parameter item
    {0}                                                  // end of items
};

static const MENU_POSITION subMenu2[] = {
    {"Sub-menu 2", 0,        0},                         // first entry is title

    {"Var1",       0,        &var1Def},                  // parameter item
    {"Var2",       0,        &var2Def},                  // parameter item
    {"Var3",       0,        &var3Def},                  // parameter item
    {"Var4",       0,        &var4Def},                  // parameter item
    {"Var5",       0,        &var5Def},                  // parameter item
    {0}                                                  // end of items
};

static const MENU_POSITION subMenu3[] = {
    {"Sub-menu 3", 0,        0},                         // first entry is title

    {"Var8",       0,        &var8Def},                  // parameter item
    {0}                                                  // end of items
};

// Level 2 menu2
//
static const MENU_POSITION subMenu1_1[] = {
    {"Sub-menu 1-1", 0,      0},                         // first entry is title

    {"Var6",       0,        &var6Def},                  // parameter item
    {"Var7",       0,        &var7Def},                  // parameter item
    {0}                                                  // end of items
};
```

This shows how simple it is to defined the overall menu structure. The first entry is always a string pointer that defies the menu's title or the item's title. The second entry is a pointer to a sub-menu (of the same `struct` array type), or the third entry (when the sub-menu pointer is zero) is a pointer to parameter details. These parameter details are discussed in subsequent sections.

Finally note that these menus have been defined as `static const` types so that that are fixed. The menus could however be defined at run time in RAM so that their content can be constructed according to other criteria. This may be useful if certain items should only be available when allowed to be, based on other system run-time settings.

For this part of the operation only two keys are needed (**UP** and **DOWN**) and the menu is displayed as follows:

The first line displays the menu title, which is centred in the display. Following it is the first item, which happens to be a sub-menu. Since this is the initially selected item it is highlighted.

Pressing the **UP** key has no effect since the first item is selected and the menu items cannot be scrolled upwards due to this.

Pressing the **DOWN** key moves down one item in the menu:

whereby the second sub-menu becomes highlighted since it is now the selected item.

Pressing the **DOWN** key moves down one further item in the menu:

which is again a sub-menu. Since the menu title field is showing an arrow pointing down we are aware that there are further items if we continue pressing the **DOWN** key.

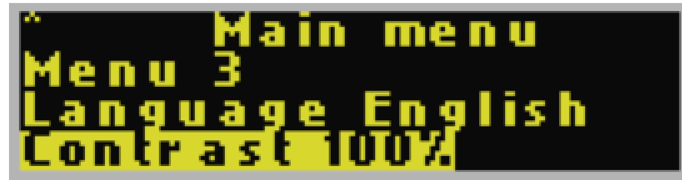
After a subsequent press of the **DOWN** key the following is seen:

whereby this next highlighted item is a parameter and not a sub-menu. Its present value is

displayed so that it is not necessary to enter a different display view in order to simply see what value it presently has.

The items in the menu list have been scrolled one position upwards so that the new item becomes visible and now the menu title field shows two arrows – one upward indicating that there are not displayed items above the first item and one downward indicating that there are still further items beyond the end of the visible menu items.

Pressing the **DOWN** key again scrolls down one item in the menu:



which happens to be a second parameter value. It is clear that this is now the final item in the menu due to the fact that the menu title no longer indicates further items after the end of the item list.

In this particular menu case further presses on the **DOWN** key don't cause any change due to the fact that the highlighted item is already the last one in the menu.

The menu items can be navigated upwards with the **UP** key, showing that a menu of any length can be scrolled though with the use of two keys. The menu title is always displayed and the presently selected item - sub-menu or parameter - is always highlighted. Indications in the menu title field indicate when not all items are visible and in which direction the additional items can be scrolled to.

1.4 Navigating in and out of Sub-Menus

Theoretically it could be possible that a single main menu is adequate for a product but in most cases it will be more practical to divide menus into multiple levels with sub-menus. To navigate into and back out of sub-menus two additional key events are required **ENTER** and **BACK**. Enter is used to move from the presently selected sub-menu item into that menu and **BACK** is used to move from a lower level sub-menu back to the previous menu.

3. Viewing System Parameters

As was seen in the previous chapter menu items van be either sub-menus or parameters, whereby parameter values are already displayed, avoiding the need to have to navigate into the parameter entry itself.

Menu items are simple as they are only names (strings) with either a pointer to the next level sub-menu or a pointer to menu parameters. Parameters however can be rather more involved due to the fact that in order to display and optionally modify them a number of pieces of information may be involved. In this section a number of parameter possibilities are shown, starting with the simplest case and adding more control as needed by the particular parameter involved.

Remembering the sub-menu 1.1 with two parameters items, and no further sub-menus:

```
// Level 2 menu2
//
static const MENU_POSITION subMenu1_1[] = {
    {"Sub-menu 1-1", 0,      0},                               // first entry is title

    {"Var6",       0,        &var6Def},                        // parameter item
    {"Var7",       0,        &var7Def},                        // parameter item
    {0}                                                        // end of items
};
```

we can take a look at how its parameter "Var7" is defined, which represents a simple byte that should be displayed but is to be read-only (not modifiable):

```
static unsigned char var7 = 72;

static const MENU_VARIABLE var7Def = {
    &var7,                                                     // the variable to control
    (MENU_VARIABLE_READ_ONLY | MENU_VARIABLE_TYPE_UNSIGNED_CHAR),     // type of variable
    0,                                                         // range
    0                                                          // images or text
};
```

The variable `var7` is referenced by a pointer to its location and its characteristic defined with the type being set to `MENU_VARIABLE_TYPE_UNSIGNED_CHAR` with the flag `MENU_VARIABLE_READ_ONLY`. The menu controller know that the user may not be given the means to modify this value and that its range is that of an unsigned char (0..255) and, in this case, it is just displayed as a decimal value.

The parameter and its present value is seen both in its menu item as well as when its menu item is executed, using the ENTER KEY when it is selected:

whereby the fact that it is *read-only* is indicated at the right of the parameter's title. When in this read-only parameter display page the only key that is responded to is the **BACK** key, which returns to the previous menu view.

Assuming that the value were to display a particular unit this can be added by setting the appropriate unit flag in the variable's type, for example

```
static const signed short var6 = -2327;

static const MENU_VARIABLE var6Def = {
    (void *)&var6,                                                          // the variable to control
    (MENU_VARIABLE_TYPE_SIGNED_SHORT | MENU_VARIABLE_READ_ONLY | MENU_VARIABLE_UNIT_MMETER),
                                                                            // type of variable
    0,                                                                      // no range
    0                                                                       // no images or text
};
```

so the `var6`'s display page shows:

where the variable's range is now that of a signed short, thus `-32'768` to `32'767` and is displayed as negative value in this particular case.

A more interesting case is the language parameter, which was seen in the main menu. This variable is retrieved from the parameter system, displayed not as a number but represented instead by further text and is limited to the number of languages that are supported.

```
static MENU_VARIABLE varLanguageDef = {
    0,                                              // the variable to control (inserted at run time)
    (MENU_VARIABLE_TYPE_UNSIGNED_CHAR | MENU_VARIABLE_RANGE_ROLL_OVER),  // type of variable
    ucRangeVarLanguage,                                                 // range limits
    &languageText                                                       // text representation
};
```

The first things to note are that its `MENU_VARIABLE` struct is not a *const* struct and that its variable pointer is set to 0. This is because its particular memory location is defined on heap at run-time and so is not known at compile time.

Before the display menu is started, the variable will be added to the struct using something like:

```
#define LCD_MENU_LANGUAGES   3

if (temp_pars->temp_parameters.ucLanguage >= LCD_MENU_LANGUAGES) {
                               // ensure that uninitialised parameters are set to a valid range value
    temp_pars->temp_parameters.ucLanguage = 0;
}
varLanguageDef.ptrVariable = &temp_pars->temp_parameters.ucLanguage;
```

The additional pointers in the MENU_VARIABLE struct are to an optional variable range limit:

```
static const unsigned char ucRangeVarLanguage[2] = { 0, (LCD_MENU_LANGUAGES - 1) };
                                                              // restrict the variable to this range
```

and the translation between the parameters individual values to value texts is defined by

```
static const MENU_REPRESENTATION languageText = {
    (VARIABLE_REPRESENTATION_TYPE_TEXT),     // the representation uses text in place of values/ranges
    clanguages,                              // the list of text to insert
    ucRangeTextVarLanguage                   // ranges controlling the text
};


static const unsigned char ucRangeTextVarLanguage[LCD_MENU_LANGUAGES] = { 0, 1, 2 };
                                                                         // value to text conversion

static const CHAR *clanguages[] = {

    "English",                                                           // valid for 0
    "German",                                                            // valid for 1
    "French",                                                            // valid for 2
    0                                                                    // end of list
};
```

The result is that the variable range is limited from 0..2 (since three languages are supported) and so the different parameter settings can be scrolled by using **UP** and **DOWN** key presses. If MENU_VARIABLE_RANGE_ROLL_OVER is used the range rools over from lower to highest value or from highest to lowest, depending on the scrolling direction. Without this flag the scrolling stops when either end of the range has been reached and further **UP** or **DOWN** key presses are ignored.

Scrolling upwards through the language display page this does:

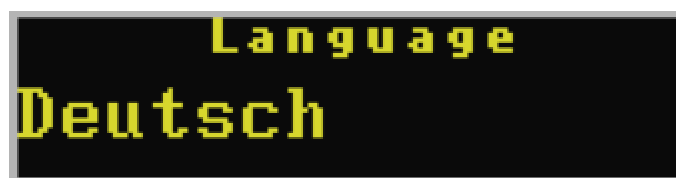
where an indication appears in the title line when the display value represents a change against the present parameter value. A new parameter value can be committed to the variable by pressing the **ENTER** key, and the change indication then disappears.

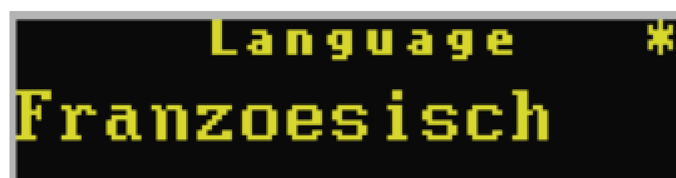
Exiting the parameter page wthe the **BACK** key resets any changes that were made in the parameter page without committing it to its variable.

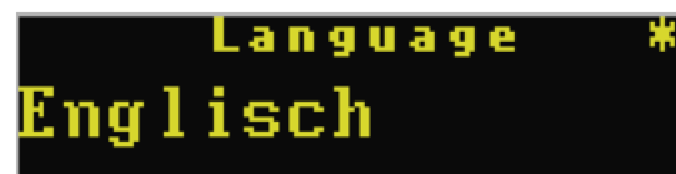
1.5 Text Languages

The previous example of a parameter controlling the language variable is a good introduction to the display menu's ability to automatically control text in multiple languages.

If the language menu is scrolled again after setting the German language mode it now looks like this:







meaning that not only has the language variable been changed but also strings used

to represent parameter values of used as menu items have been translated too. Moving up to the main menu shows this:

Controlling language strings is very simple since they just need to be entered into a single array:

```
static const CHAR *menuText[][LCD_MENU_LANGUAGES] = {
    {"German",  "Deutsch",     "Alemande"},
    {"English", "Englisch",    "Anglais"},
    {"French",  "Franzoesisch","Francais"},
    {"Language","Sprache",     "Langue"},
    {"COLD",    "kalt",        "froid"},
    {"WARM",    0,             "TIEDE"},
    {"HOT",     "heiss",       "chaud"},
    {0}                                                                 // end of list
};
```

The first column of the array is the base language.

Each time a string is used in the menu and the present language is not equal to the base language the table is scanned to see whether there is an entry for the string that is to be displayed.

- If no match is found the string is used as it is. Thus, when no translation of an item is needed, no entry is required in the `menuText[]` array.
- If a match is found in the base language column but the present language entry is 0 the base language is still used – no translation s necessary as in the case of the string „WARM“ which is written identically in German to English.
- If a match is found in the base language column and there is also a string entry for the present language the translated string is inserted instead

Langage extensions can thus be easily added at any time to a menu by adjusting the language parameter range accordingly and inserting a further column in the `menuText[]` array.

4. Graphical Representation of Parameter Values

In some circumstances it can be interesting to not represent parameter values simply as numbers, or even as text equivalents, but as graphical symbols instead. A classic case would be a volume or brightness setting for which the following illustrates such a utilisation:

First of all we add a parameter variable for a range of 0..3, which is displayed as "Volume" in a menu.

```
static unsigned char ucVolume = 0;                              // the parameter variable
static const unsigned char ucRangeVoume[2] = { 0, 7 };          // restrict the variable to this range

static const MENU_VARIABLE varVolumeDef = {
    &ucVolume,                                                  // the variable to control
    MENU_VARIABLE_TYPE_UNSIGNED_CHAR,                           // type of variable
    &ucRangeVoume,                                              // range
    0                                                           // neither images nor text
};
```

It is defined presently as a value between 0 and 7 (eight volume levels), which allows it to be displayed in the same was as other numerical variables:

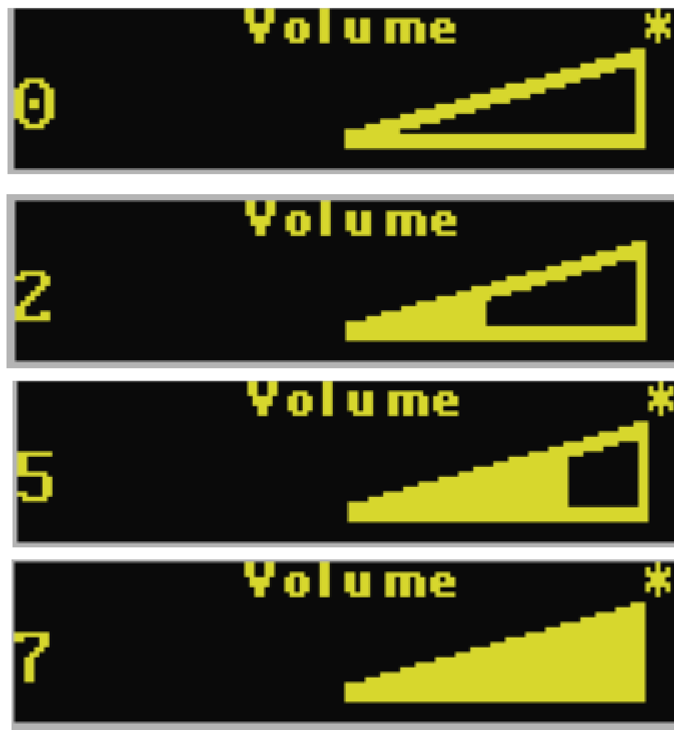
It could also be displayed textually as "LOW", .."HIGH" or similar but in this case we decide to display it graphically so it is entered as follows:

```
static const MENU_VARIABLE varVolumeDef = {
    &ucVolume,                                                  // the variable to control
    MENU_VARIABLE_TYPE_UNSIGNED_CHAR,                           // type of variable
    &ucRangeVoume,                                              // range
    &volumeGraphic                                              // image representation
};

static const MENU_REPRESENTATION volumeGraphic = {
    (VARIABLE_REPRESENTATION_TYPE_BMP | VARIABLE_REPRESENTATION_SHOW_VALUE),
                                          // the representation uses images to represent values/ranges
    volume_images,                                    // the list of graphics to insert
    ucRangeGraphicsVarVolume                          // ranges controlling the graphics
};

static const unsigned char ucRangeGraphicsVarVolume[4] = { 0, 2, 4, 6 };
                                                      // value to graphic conversion
static const GBITMAP *volume_images[] = {
    (GBITMAP *)level0,                                          // valid for 0..< 2
    (GBITMAP *)level1,                                          // valid for 2..< 4
    (GBITMAP *)level2,                                          // valid for 4..< 6
    (GBITMAP *)level3,                                          // valid for >= 6
    0                                                           // end of list
};
```

and looks like this instead:



In this case 4 images have been defined and so 0 and 1 show the first level, 2 and 3 the second, etc. in order to show how to reduce the quantity of images used. Due to this, the flag `VARIABLE_REPRESENTATION_SHOW_VALUE` has be used so that the actual value is also shown together with the image. If this flag is not used only the image is displayed and generally there would be one unique image for each parameter variable value to ensure that the user has the positive feedback that each increment and decrement in the range is being correctly responded to.

When the parameter value is displayed as a menu item only its value is shown due to the fact that there is usually not enough space for the image, but the image in the parameter variable display helps to show the relationship of the present value to its overall range. Images also help to make the general feel of the menus more lively and interesting.

The bit maps used by the display menu can be created and embedded into the code by the `µTaskerFileCreate` utility as explained in the uGLCD User's Guide https://www.utasker.com/docs/uTasker/uTaskerLCD.PDF

5. ...

6. Conclusion

Modifications:

V0.04 8.02.2021 Preliminary version in progress

Appendix A

a) ….