Generalised User Interface for Embedded Applications using an LCD screen and keypad.

This article is concerned with firmware design and implementation for microcontroller-based devices incorporating a "local" user interface (front panel) comprising an LCD screen and keypad (or a number of push-buttons). The technique can be extended to build a graphical user interface (GUI) comprising an LCD screen with touch-panel.

Typical applications will have many "screens" to be presented and, for each screen, a selection of user options for various actions, most of which will result in a switch to a different screen or a change in displayed information on the current screen. For all but the most trivial of applications, navigation from one screen to the next presents a challenge to the developer and can easily become a convoluted mess if not handled methodically.

Test box fitted with monochrome graphics LCD module (128 x 64 pixels) and 4 x 4 keypad

A generalised UI screen navigation system makes the firmware architecture more elegant, the code easier to read and hence easier to modify and extend. The scheme to be presented here integrates well with real-time embedded operating systems. However, it is also suitable for firmware architectures without a 3rd-party embedded OS. With the exception of applications at the very high end of the complexity scale, a 3rd-party OS is unnecessary and, in the author's experience, usually introduces inefficiencies and annoying restrictions.

An example main function of a firmware architecture without a 3rd-party embedded OS is given below. The example application comprises a number of "background tasks", i.e. functions which are executed periodically, scheduled by (but not called directly from) a real-time (or relative time) clock interrupt service routine. Non-periodic (*ad hoc*) background tasks may be executed when activated by a system event, e.g. peripheral signal, timer expiry, signal from another task, etc.

It is important to note that background tasks must be "non-blocking", i.e. must not contain software delays or wait loops. This requirement is best achieved by implementing background tasks as state machines. That topic is outside the scope of this article, but there is an abundance of literature on software state-machine design available on the Internet.

The example application includes a "local" user interface ("LUI" or simply "UI") and a command-line interface (CLI). The latter is implemented by a serial port (UART) to which may be connected a PC running a terminal emulator application (e.g. PuTTY-tel).

```
int  main(void)
{
    InitializeMCU();
    Init_Application();
    LUI_NavigationInit();       // Initialize LUI
    PrepareForNewCommand();     // Initialize CLI

    while ((1 + 1) == 2)        // main loop
    {
        BackgroundTaskExec();
        ConsoleCLI_Service();
        LUI_NavigationExec();
    }
}
```

Running "concurrently" with the background tasks and user interface routines, interrupt service routines (ISRs) provide a real-time clock (periodic interrupt) and some hardware device drivers, e.g. for the keypad or touch-panel. In this context, interrupting task functions may be termed "foreground tasks".

Called frequently from the main loop, the function "LUI_NavigationExec()" checks for a request to "switch" screens and, if true, calls a function to render the requested new screen. At periodic intervals thereafter, typically 50ms, the UI service routine, LUI_NavigationExec(), calls the same screen function again to update displayed information.

The "application program interface" (API) to the local UI is comprised of a small suite of functions, listed below:

```
void    LUI_NavigationInit();
int     GetNumberOfScreensDefined();
uint16  GetCurrentScreenID();
uint16  GetPreviousScreenID();
void    GoToNextScreen(uint16 nextScreenID);
bool    ScreenSwitchOccurred(void);
int     ScreenDescIndexFind(uint16 searchID);
void    DisplayMenuOption(uint16 x, uint16 y, char symbol, char *text);
```

Each different screen to be presented is given a unique ID number. The screen ID's are enumerated in a header file included in the UI code module. An incomplete example follows:

```
enum  Set_of_Screen_ID_numbers   // Any arbitrary order
{
    SCN_STARTUP = 0,
    SCN_HOME,
    SCN_SELFTEST_REPORT,
    SCN_SETUP_MENU_PAGE1,
    SCN_SETUP_MENU_PAGE2,
    SCN_SETUP_MENU_PAGE3,
    SCN_PRESET_SELECT,
    SCN_PRESET_EXT_INFO,
    SCN_SET_AUDIO_RANGE,
    SCN_SYSTEM_INFO,
    SCN_DATA_ENTRY,
    SCN_DATA_ENTRY_TEST,
    NUMBER_OF_SCREEN_IDS        // Last entry in table
};
```

Each of the enumerated screens is further defined by a data structure comprising the screen ID number, the address of (i.e. pointer to) a "screen update function" and a pointer to a text string which is to be displayed in a "title bar" rendered at the top of the screen. If the title bar string is NULL, no title bar will be displayed. This structure is termed "screen descriptor".

```
// An object of this type is needed for each different UI screen.
// An array of structures of this type is held in program memory.
// For screens which have no Title Bar, initialize TitleBarText = NULL
//
typedef struct LUI_screen_descriptor
{
    uint16  screen_ID;              // Screen ID number
    void    (*ScreenFunc)(bool);   // Screen update function (addr)
    char    *TitleBarText;          // Pointer to title string

} LUI_ScreenDescriptor_t;
```

In a more elaborate GUI with touch-screen for user input, the "screen descriptor" might be extended to include pointers to "touch region" (hot spot) definitions and perhaps "tool bar button" definitions. For the moment, let's keep it simple and assume a UI with a physical keypad or a set of push-buttons.

The "screen update function" (or simply "screen function") is not called directly from the application program. It is called (conditionally) by the LUI_NavigationExec() routine with a single boolean argument, the value of which is false (0) on the first call following a screen switch, indicating to the function that it must render the new screen content. Thereafter, the argument is "true" (1) indicating to the function that it must update any displayed information that has changed since the previous call.

Screen (update) functions also have the responsibility to monitor user input (i.e. for keypad/button hits) and other system "events", e.g. timer expiry flags, peripheral activity, etc, and to take appropriate action on these signals. Thus, the UI itself comprises a major part of the "generic operating system". (Where a 3rd-party OS is incorporated, UI screen functions may send signals to other tasks via the OS.)

Often, a "user input event" (e.g. key hit) or other signal will be expected to cause a screen switch. The screen update function achieves this by a call to the API function GoToNextScreen(), the argument of which is the ID number of the new screen to be presented.

The complete user interface is defined by an array of "screen descriptors" initialised and located in program memory (as "const" data), as in the example below:

```
// Screen descriptors (below) may be arranged in any arbitrary order in
// the array m_ScreenDesc[], i.e. the table doesn't need to be sorted into
// screen_ID order.
// Function LUI_ScreenIndexFind() is used to find the index of an element
// within the array m_ScreenDesc[], for a specified screen_ID.
//
static  const  LUI_ScreenDescriptor_t  m_ScreenDesc[] =
{
    {
        SCN_STARTUP,                 // screen ID
        ScreenFunc_Startup,          // screen update function
        NULL                         // title bar text (none)
    },
    {
        SCN_HOME,
        ScreenFunc_Home,
        NULL                         // title bar text (none)
    },
    {
        SCN_SELFTEST_REPORT,         // screen ID
        ScreenFunc_SelfTestReport,   // screen update function
        "SELF-TEST Failed"           // title bar text
    },
    {
        SCN_SETUP_MENU_PAGE1,
        ScreenFunc_SetupMenuPage1,
        "SETUP Config Param's"
    },
    {
        SCN_SETUP_MENU_PAGE2,
        ScreenFunc_SetupMenuPage2,
        "SETUP PRESET Param's"
    },
    {
        SCN_SETUP_MENU_PAGE3,
        ScreenFunc_SetupMenuPage3,
        "SETUP PRESET Param's"
    },

        ...  // etc
};
```

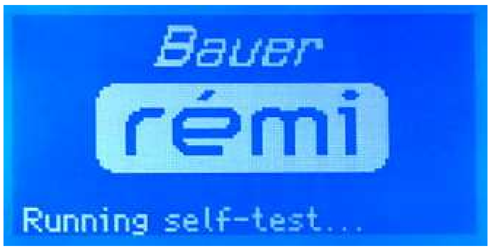
Screen functions are declared PRIVATE (static) because they are defined in the same code module (source file) as the UI service routine, LUI_NavigationExec(), and all other code comprising the UI. This makes the UI code module application-specific, of course, but it still helps to abstract the UI from other application functions. Only the screen descriptor table and screen update functions are application-specific. The rest of the UI code module is generalised and hence "portable" to other applications. Purists might prefer to separate out the application-specific code and put it into a dedicated source file.

Let's have a look at one or two example screen functions forming part of a real application. At power-on/reset, the firmware runs a number of "self-test" routines taking a second or two to finish. While the self-test is running, the UI displays a "start-up" screen with a graphics image and a text message: "Running Self-Test...". There is no title bar. Rather than check a signal indicating completion of the self-test task, the screen update function simply waits for a 3-second timer to expire, allowing ample time for the self-test to complete, but more importantly, ample time for the user to marvel at the developer's graphic image creation. Code for handling the timer-counter variable is built into the LUI_NavigationExec() routine.

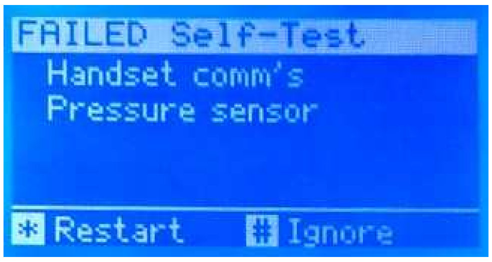
When the timer-counter reaches 3000ms, the screen function invokes a switch to the next screen. If the self-test passed, the next screen is a "Home" screen displaying a "main menu". Otherwise, the next screen is a "Self-Test Results" screen showing which test(s) failed.

```
/*
 * The function below displays the "startup" screen for 3 seconds, then
 * triggers a switch to the "Home" (Main Menu) screen. The screen timer,
 * m_ElapsedTime_ms, is managed by LUI_NavigationExec().
 * The timer is re-started (zeroed) whenever a screen switch occurs or
 * if a key-hit is actioned, and at system restart.
 */
PRIVATE  void  ScreenFunc_Startup(bool update)
{
    int   i;
    bool  isFailedSelfTest;

    if (!update)  // render new screen...
    {
        LCD_Mode(SET_PIXELS);
        LCD_PosXY(21, 2);
        LCD_PutImage(Bauer_logo_85x45, 85, 45);

        LCD_SetFont(BAUER_PROP_8_NORM);
        LCD_PosXY(3, 56);
        LCD_PutText("Running self-test...");
    }
    else  // do periodic update...
    {
        if (m_ElapsedTime_ms >= 3000)  // 3 sec timer expired
        {
            isFailedSelfTest = 0;

            // Check self-test results... if fail, go to results screen
            for (i = 0;  i < NUMBER_OF_SELFTEST_ITEMS;  i++)
            {
                if (g_SelfTestFault[i] != 0) isFailedSelfTest = 1;
            }

            if (isFailedSelfTest)  GoToNextScreen(SCN_SELFTEST_REPORT);
            else  GoToNextScreen(SCN_HOME);
        }
    }
}
```

Note: The coding standard adopted by the author specifies a 2-character prefix on data identifiers indicating their scope. The prefix "g_" means global, i.e. is declared “extern”, whereas "m_" means that an identifier is declared “static” outside of function definitions, i.e. its scope is restricted to the code module (*sic*) in which it is defined. Data identifiers without a prefix should have their scope limited to the function in which they are defined.


Bauer
rémi
Running self-test...

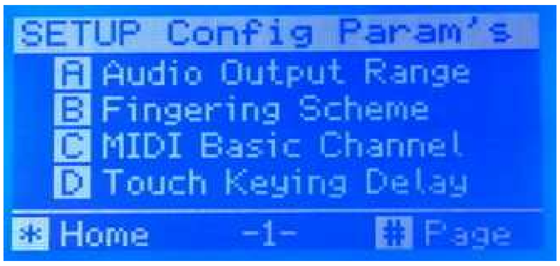
Example “startup” screen, as drawn by ScreenFunc_Startup()


```
FAILED Self-Test
  Handset comm's
  Pressure sensor

* Restart      # Ignore
```

Example “self-test report” screen, shown if a self-test fails during startup.

The next example screen function displays a list of user options and monitors the keypad for specific keys, one of which might be pressed to select a corresponding option. The keypad driver API provides a function KeyHit() which returns 'true' if any key has been hit since the last time the function was called. Another function, GetKey(), returns the keycode of the last key pressed. GetKey() does not wait for user input. It assumes KeyHit() has been called prior and that it has returned 'true'. In this manner, both the keypad driver and screen functions are "non-blocking", i.e. they don't hold up the works.

```
PRIVATE  void  ScreenFunc_SetupMenuPage1(bool update)
{
    if (!update)  // render new screen...
    {
        DisplayMenuOption(10, 12, 'A', "Audio Output Range");
        DisplayMenuOption(10, 22, 'B', "Fingering Scheme");
        DisplayMenuOption(10, 32, 'C', "MIDI Basic Channel");
        DisplayMenuOption(10, 42, 'D', "Touch Keying Delay");
        LCD_PosXY(0, 53);
        LCD_DrawLineHoriz(128);
        DisplayMenuOption(0, 56, '*', "Home");
        DisplayMenuOption(88, 56, '#', "Page");
        LCD_PosXY(56, 56);
        LCD_PutText("-1-");
    }
    else  // updating – monitor key presses
    {
        if (KeyHit())
        {
            if (GetKey() == '*') GoToNextScreen(SCN_HOME);
            else if (GetKey() == '#') GoToNextScreen(SCN_SETUP_MENU_PAGE2);
            else if (GetKey() == 'A') GoToNextScreen(SCN_SET_AUDIO_RANGE);
            else if (GetKey() == 'B') GoToNextScreen(SCN_SET_KEYING_SCHEME);
            else if (GetKey() == 'C') GoToNextScreen(SCN_SET_MIDI_CHANNEL);
            else if (GetKey() == 'D') GoToNextScreen(SCN_SET_TOUCH_DELAY);
        }
    }

    if (m_ElapsedTime_ms >= LUI_INACTIVE_TIMEOUT)  // 10 minute time-out
    {
        GoToNextScreen(SCN_HOME);
    }
}
```



Example "SETUP" screen (page 1 of 3), as drawn by ScreenFunc_SetupMenuPage1()

Next, the internal workings of the UI service routine, LUI_NavigationExec(), may be examined. Here is the source listing:

```
/*
 * LUI Navigation engine (service routine, or whatever you want to call it).
 * This is the "executive hub" of the Local User Interface.
 *
 * The function is called frequently from the main application loop, but...
 * must not be called from BackgroundTaskExec() !!
 */
void  LUI_NavigationExec(void)
{
    short  current, next;  // index values of current and next screens

    if (m_ScreenSwitchFlag)   // Screen switch requested
    {
        m_ScreenSwitchFlag = 0;
        m_ElapsedTime_ms = 0;
        m_lastUpdateTime = millisecTimer();
        current = ScreenDescIndexFind(m_CurrentScreen);
        next = ScreenDescIndexFind(m_NextScreen);

        if (next < NUMBER_OF_SCREENS_DEFINED)  // found next screen ID
        {
            m_PreviousScreen = m_CurrentScreen;     // Make the switch...
            m_CurrentScreen = m_NextScreen;         // next screen => current

            if (m_NextScreen != m_PreviousScreen)
            {
                LUI_EraseScreen();

                if (m_ScreenDesc[next].TitleBarText != NULL)
                    DisplayTitleBar(next);
            }

            (*m_ScreenDesc[next].ScreenFunc)(0);  // Render new screen
            m_screenSwitchDone = TRUE;
        }
    }
    else  // no screen switch -- check update timer
    {
        if (millisecTimer() - m_lastUpdateTime >= SCREEN_UPDATE_INTERVAL)
        {
            current = ScreenDescIndexFind(m_CurrentScreen);

            (*m_ScreenDesc[current].ScreenFunc)(1);  // Update current screen

            m_lastUpdateTime = millisecTimer();
            m_ElapsedTime_ms += SCREEN_UPDATE_INTERVAL;
        }
    }
}
```

The application program requests a screen switch by calling the API function, GoToNextScreen(), the argument of which is the ID number of the new screen to be presented. Most, if not all, screen switch requests occur inside screen update functions.

```
void  GoToNextScreen(uint16 nextScreenID)
{
    m_NextScreen = nextScreenID;
    m_ScreenSwitchFlag = 1;
}
```

A few static UI system variables need to be initialized at power-on/reset. This is done by function LUI_NavigationInit(), thus:

```
/*
 * LUI initialization function...
 */
void  LUI_NavigationInit()
{
    m_CurrentScreen = SCN_STARTUP;
    m_PreviousScreen = SCN_STARTUP;
    m_DataEntryNextScreen = SCN_HOME;
    m_ScreenSwitchFlag = 1;
    m_screenSwitchDone = 0;
}
```

Additional functions listed below complete the generalized UI screen navigation system:

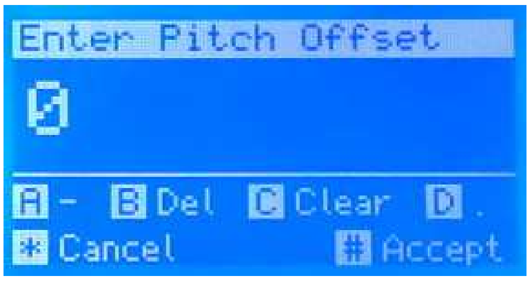
```
/*
 * Function returns the index of a specified screen in the array of Screen
 * Descriptors, (LUI_ScreenDescriptor_t)  m_ScreenDesc[].
 *
 * Entry arg(s):  search_ID = ID number of required screen descriptor
 *
 * Return value:  index of screen descriptor in array m_ScreenDesc[], or...
 *                NUMBER_OF_SCREENS_DEFINED, if search_ID not found.
 */
int  ScreenDescIndexFind(uint16 searchID)
{
    int   index;

    for (index = 0; index < NUMBER_OF_SCREENS_DEFINED; index++)
    {
        if (m_ScreenDesc[index].screen_ID == searchID)  break;
    }

    return index;
}

/*
 * Function renders the Title Bar (background plus text) of a specified screen.
 * The title bar text (if any) is defined in the respective screen descriptor
 * given by the argument scnIndex. The function is called by LUI_NavigationExec();
 * is not meant to be called directly by application-specific screen functions.
 *
 * The location and size of the Title Bar and the font used for its text string
 * are fixed inside the function.
 *
 * Entry arg:  scnIndex = index (not ID number!) of respective screen descriptor
 */
PRIVATE  void  DisplayTitleBar(uint16 scnIndex)
{
    char  *titleString = m_ScreenDesc[scnIndex].TitleBarText;

    LCD_Mode(SET_PIXELS);
    LCD_PosXY(0, 0);
    LCD_BlockFill(128, 10);  // Erase top line of text (10px high)

    if (titleString != NULL)
    {
        LCD_Mode(CLEAR_PIXELS);  // Title bar is reverse video
        LCD_SetFont(BAUER_MONO_8_NORM);
        LCD_PosXY(2, 1);
        LCD_PutText(titleString);
    }

    LCD_Mode(SET_PIXELS);
}
```

```
/*
 * This function displays a single-line menu option, i.e. keytop image plus text.
 * The keytop image is simply a square (size 9 x 9 pixels) with a character drawn
 * inside it in reverse video.
 * The given text string is printed just to the right of the keytop image.
 * The character font(s) used are fixed within the function.
 *
 * Entry args:   x = X-coord of keytop image (2 pixels left of key symbol)
 *               y = Y-coord of keytop symbol, same as text to be printed after
 *               symbol = ASCII code of keytop symbol (5 x 7 mono font)
 *               text = string to print to the right of the keytop image
 */
void  DisplayMenuOption(uint16 x, uint16 y, char symbol, char *text)
{
    uint16  xstring = x + 12;  // x-coord on exit

    LCD_Mode(SET_PIXELS);
    LCD_PosXY(x, y-1);
    LCD_DrawBar(9, 9);

    LCD_SetFont(BAUER_MONO_8_NORM);
    LCD_Mode(CLEAR_PIXELS);
    LCD_PosXY(x+2, y);
    if (symbol > 0x20) LCD_PutChar(symbol);

    LCD_SetFont(BAUER_PROP_8_NORM);
    LCD_Mode(SET_PIXELS);
    LCD_PosXY(xstring, y);
    if (text != NULL) LCD_PutText(text);
}
```

Most applications will also benefit from a generalised data-entry screen function which can be incorporated in the user interface. The screen function given below allows a real number to be entered in similar manner to a calculator. The Title Bar text is variable (context-dependent) and should be assigned by the currently active screen function, prior to the data entry function, to prompt the user to enter a number and to indicate its purpose.

Four buttons on the keypad are assigned to editing the numeric input string: [A] Enters a minus sign (must be first character), [B] Backspace (delete last character), [C] Clear entry (show zero) and [D] Decimal point. Up to 10 characters may be input.

The asterisk key [*] cancels data entry and exits. The hash key [#] accepts the number entered and proceeds to the next screen, the ID of which is expected to be assigned to the variable m_DataEntryNextScreen by the screen function which led to the data entry screen being shown. On exit with the “accept” key, the data entry function converts the input string to a floating point value and assigns it to a variable, m_DataEntryValue. On exit with the “cancel” key, the data entry function sets a variable m_DataEntryAccept to False (0).



Example “Data Entry” screen

Typically, the next screen on exit from Data Entry will be a function to test the validity of the number entered. An example of such a function is given following the data entry function.

```c
PRIVATE  void  ScreenFunc_DataEntry(bool update)
{
    static  short  charPlace = 0;
    static  char   inputString[32];
    static  bool   got_DP = 0;
    uint8   key;
    uint16  xpos;

    if (!update)  // render new screen...
    {
        LCD_Mode(SET_PIXELS);  // Draw title bar
        LCD_PosXY(0, 0);
        LCD_BlockFill(128, 10);
        LCD_Mode(CLEAR_PIXELS);
        LCD_SetFont(BAUER_MONO_8_NORM);  // Title text
        LCD_PosXY(2, 1);
        if (m_DataEntryTitle != NULL)  LCD_PutText(m_DataEntryTitle);
        else  LCD_PutText("Enter value...");

        LCD_Mode(SET_PIXELS);
        LCD_PosXY(0, 40);
        LCD_DrawLineHoriz(128);
        DisplayMenuOption(0, 44, 'A', "-");
        DisplayMenuOption(25, 44, 'B', "Del");
        DisplayMenuOption(60, 44, 'C', "Clear");
        DisplayMenuOption(105, 44, 'D', ".");
        DisplayMenuOption(0, 56, '*', "Cancel");
        DisplayMenuOption(82, 56, '#', "Accept");

        m_DataEntryValue = 0.0;
        m_DataValueAccept = 0;   // Nothing entered yet
        inputString[0] = '\0';
        charPlace = 0;
        got_DP = 0;

        LCD_SetFont(BAUER_MONO_16_NORM);
        LCD_Mode(SET_PIXELS);
        LCD_PosXY(4, 16);
        LCD_PutChar('0');
    }
    else  // do periodic update...
    {
        if (KeyHit())
        {
            xpos = 4 + charPlace * 12;
            LCD_SetFont(BAUER_MONO_16_NORM);
            LCD_Mode(SET_PIXELS);
            LCD_PosXY(xpos, 16);

            if (charPlace == 0)  // clear input field
            {
                LCD_Mode(CLEAR_PIXELS);
                LCD_PosXY(4, 16);
                LCD_BlockFill(12, 16);
                LCD_Mode(SET_PIXELS);
            }

            if ((key = GetKey()) == '*')  // Cancel data entry
            {
                m_DataEntryValue = 0.0;
                m_DataValueAccept = 0;
                GoToNextScreen(m_DataEntryNextScreen);
            }
            else if (key == '#')  // Accept value entered
            {
                if (charPlace == 0) m_DataEntryValue = 0.0;
                else  sscanf(inputString, "%f", &m_DataEntryValue);
                m_DataValueAccept = 1;
                GoToNextScreen(m_DataEntryNextScreen);
            }
            else if (key >= '0' && key <= '9' && charPlace < 10)
            {
                inputString[charPlace] = key;
                charPlace++;
                inputString[charPlace] = '\0';
                LCD_PutChar(key);
            }
            else if (key == 'A' && charPlace == 0)  // Minus sign
            {
                inputString[0] = '-';
                charPlace++;
                inputString[1] = '\0';
                LCD_PutChar('-');
            }
            else if (key == 'B' && charPlace > 0)  // Backspace (Del)
            {
                charPlace--;
                if (inputString[charPlace] == '.') got_DP = 0;
                inputString[charPlace] = '\0';
                xpos = 4 + charPlace * 12;
                LCD_Mode(CLEAR_PIXELS);
                LCD_PosXY(xpos, 16);
                LCD_BlockFill(12, 16);
            }
            else if (key == 'C')  // Clear input field
            {
                LCD_Mode(CLEAR_PIXELS);
                LCD_PosXY(0, 16);
                LCD_BlockFill(128, 16);
                m_DataEntryValue = 0.0;
                m_DataValueAccept = 0;
                inputString[0] = '\0';
                charPlace = 0;
                got_DP = 0;
            }
            else if (key == 'D' && charPlace < 10 && !got_DP)  // Decimal Point
            {
                inputString[charPlace] = '.';
                charPlace++;
                inputString[charPlace] = '\0';
                LCD_PutChar('.');
                got_DP = 1;
            }

            if (charPlace == 0)  // input field cleared -- show '0'
            {
                LCD_Mode(SET_PIXELS);
                LCD_PosXY(4, 16);
                LCD_PutChar('0');
            }
        }
    }
}
```

Typically, the next screen on exit from Data Entry will be a function to test the validity of the number entered. An example of such a function is given below...

```
PRIVATE  void  ScreenFunc_DataEntryTest(bool update)
{
    char    textBuf[32];

    if (!update)  // render new screen...
    {
        LCD_Mode(SET_PIXELS);
        LCD_SetFont(BAUER_PROP_8_NORM);
        LCD_PosXY(30, 22);
        LCD_PutText("Input value:- ");
        LCD_SetFont(BAUER_PROP_8_NORM);

        if (m_DataValueAccept)  sprintf(textBuf, "%8.3f", m_DataEntryValue);
        else  strcpy(textBuf, "N/A");

        LCD_SetFont(BAUER_MONO_8_NORM);
        LCD_PosXY(30, 32);
        LCD_PutText(textBuf);

        LCD_SetFont(BAUER_PROP_8_NORM);
        LCD_PosXY(0, 53);
        LCD_DrawLineHoriz(128);
        DisplayMenuOption(0, 56, '*', "Home");
        DisplayMenuOption(80, 56, '#', "Exit");
    }
    else  // do periodic update...
    {
        if (KeyHit())
        {
            if (GetKey() == '*') GoToNextScreen(SCN_HOME);  else
            if (GetKey() == '#') GoToNextScreen(SCN_DATA_ENTRY);
        }
    }
}
```

Example UI functions and screen-shots presented in this article were extracted from a digital musical instrument project. Full details of that project and C source code are available on the author’s website. See reference [3] below.

Example application “HOME” screen.

References:

[1] LCD graphics library (C code) for monochrome LCD module 128 x 64 pixels using ST7920 LCD controller:
http://www.mjbauer.biz/mjb_resources.htm#LCD%20graphics%20lib

[2] Keypad interface to microcontroller using 4 wires:
http://www.mjbauer.biz/Four-wire%20keypad%20interface.pdf

[3] Build the “REMI” – a ‘DIY’ Electronics Project by M.J. Bauer:
http://www.mjbauer.biz/Build_the_REMI_by_MJB.htm