by STEVE DREVIK

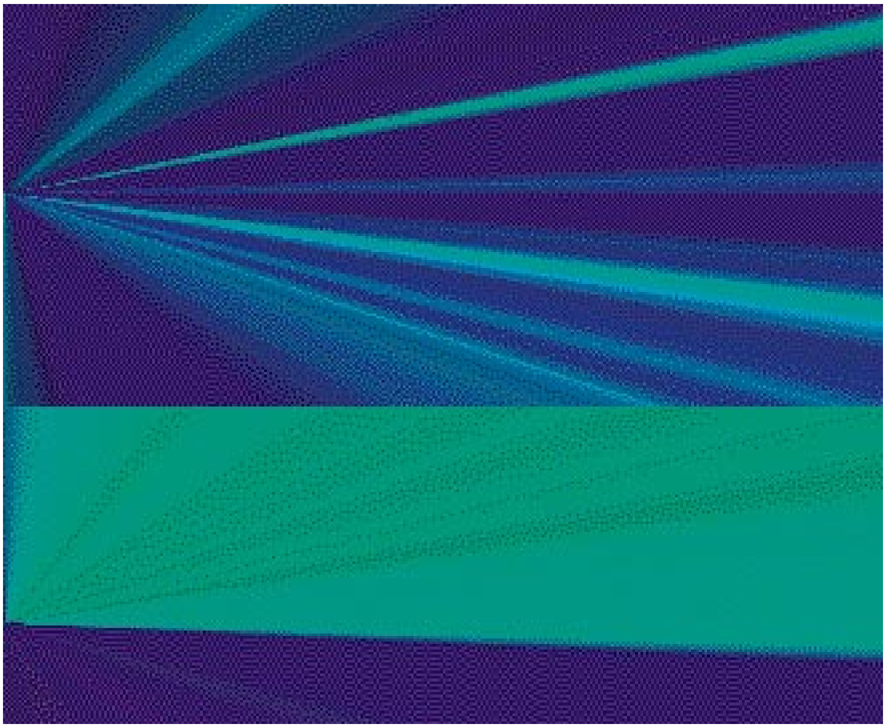
< BACK

Text Interfaces for Embedded Systems

A text-based menu system is a “middle ground” interface, more complex but smaller and simpler than graphical Windows-style interfaces. This article demonstrates how to develop a text-based menu system, using an automatic teller machine as an example.

User interfaces for embedded systems come in a wide variety of styles and complexity. The interface can be as simple as a set of pushbuttons with LEDs for feedback or as complex as a graphical Windows-style interface. The level of sophistication is generally limited by the amount of available memory and code space, and a good graphical interface with online help is typically larger than one Mbyte of code space, which is out of the range of most embedded systems.

This article traces the development of a text-based menu system that provides a balance between functionality, expandibility, and code space. We’ll look at the basics of such a system. I’ll suggest additional features that may be layered onto the basic system.



Millionaire Graphics

A text-based menu system presents full-screen menus to the user containing several selections. These selections may lead to other menus or perform other functions. This is the same type of interface provided by dial-up bulletin board systems (BBSs), most automatic teller machines (ATMs), and so on. Text-based menus can be presented on a built-in LCD or over a serial port to a device capable of VT100 or ANSI terminal emulation.

Text-based menus provide several advantages:

- They give a high level of control and access by the user (given a sufficient number of menus and prompts).
- They are user-friendly—prompts can be relatively descriptive in 20 to 40 characters, and menus are easier to navigate than command-line interfaces (like DOS).
- They gain wide acceptance from a variety of interface devices (PCs, handheld terminals, built-in displays).

Perhaps the biggest advantage for the programmer is that a well-designed menu system can be easily changed and adapted for increasing requirements and functionality.

DESIGN

A menu is defined as the contents of any particular screen presented to the user. A menu will present two or more selections for the user to choose from:

```
/* 25 lines per page is reasonable for
most terminal types, but programmers can
allocate space for headers, footer text */

#define MAX_SELECTIONS  24

<insert typedef for SELECTION here >

typedef struct _menu
{
  int           id;
  int           num_selections;
  SELECTION     selection[MAX_SELECTIONS];
}MENU;
```

We add an ID code to each menu for reference later. These ID codes can be attached to descriptive labels in the code using an enumeration:

```
enum      _menu_ids
{
  MAIN_MENU,
  SECONDARY_MENU,
  ....
  };
```

The selection is the basic building block of the menu. When selected, it can lead to another menu or invoke an action function such as saving a file or spawning a control task. It consists minimally of a prompt, or label string, and selection action:

```
typedef struct      _selection
{
char                prompt[MAX_PROMPT_LEN];
int                 (*function)(int);
int                 fn_arg;
}SELECTION;
```

Here, we assume that the selection action will have an integer argument (for additional flexibility) and will be designed to return an integer value (to return possible errors).

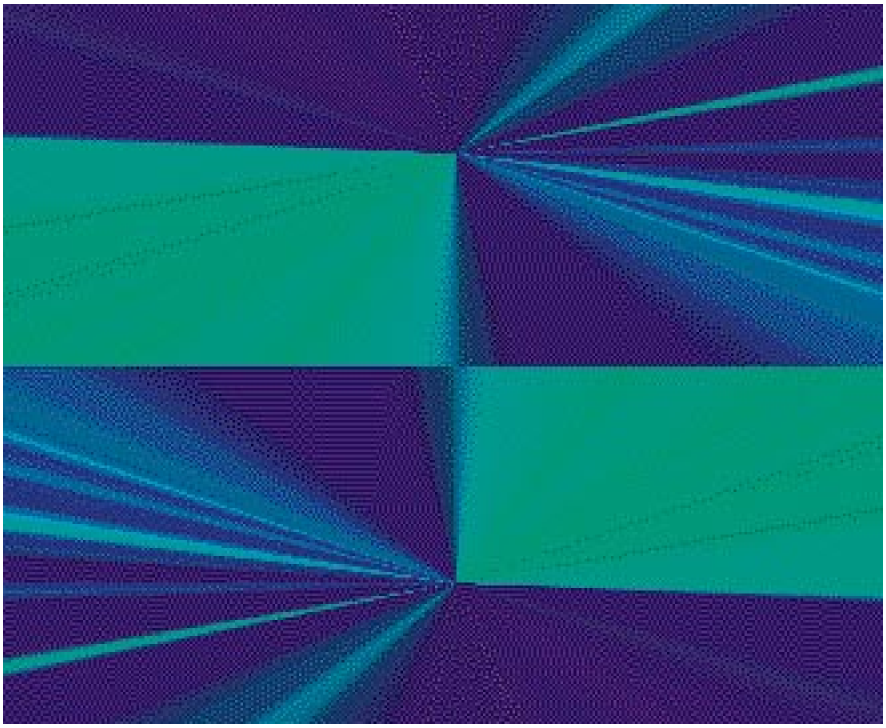
The menu structure for an ATM machine (presented after inserting the card and entering the PIN) might be defined as shown in Listing 1.

As new account types are added (money markets, IRAs, credit cards), all the programmer needs to do is to add appropriate items to the menu, and write appropriate handlers for `do_deposit()`, `do_withdrawl()`, and `show_balance()`. If the ATM needs to be expanded to initiate a videophone call to the bank using the built-in camera and microphone, a new item can be added to the main menu and a function handler written to support the new feature.

The menu is managed by a single function:

```
void mdi()        /* Menu Driven Interface */
{
char              c;
MENU              *curr_menu =
                  find_menu(MAIN_MENU);
int               curr_selection = 0;

/* Peform I/O initialization, draw an
introduction page, and so on */

init_mdi()

/* Draw the main menu */

draw_menu(curr_menu, curr_selection);

while(TRUE)
```

The selection is the basic building block of the menu. It can lead to another menu or invoke an action function such as saving a file.



Text Interfaces for Embedded Systems

LISTING 1
Simple ATM menus.

```
enum    _menu_ids
{
  MAIN_MENU,
  DEPOSIT_MENU,
  WITHDRAWL_MENU,
  BALANCE_MENU
};

MENU              menu[] =
{
  {    MAIN_MENU,4,
  {"Perform A Deposit",              goto_menu(),               DEPOSIT},
  {"Perform a Withdrawl",            goto_menu(),               WITHDRAWL},
  {"Obtain a Balance Statement",     goto_menu(),               BALANCE),
  {"Quit",                           logout(),                  0}
  },

  {    DEPOSIT_MENU,      3,
  {"Deposit to Checking",            do_deposit(),              CHECKING},
  {"Deposit to Savings",             do_deposit(),              SAVINGS),
  {"Go Back",                        goto_menu(),               MAIN_MENU}
  },

  {    WITHDRAWL_MENU,               3,
  {"Withdrawl from Checking",        do_withdrawl(),            CHECKING),
  {"Withdrawl from Savings",         do_withdrawl(),            SAVINGS),
  {"Go Back",                        goto_menu(),               MAIN_MENU}
  },

  {                                  BALANCE_MENU,              3,
  {"Print Checking Balance",         show_balance(),            CHECKING},
  {"Print Savings Balance",          show_balance(),            SAVINGS},
  {"Go Back",                        goto_menu(),               MAIN_MENU}
  },

  /* Other menus here */
};
```

```
 {
```

The following instruction is where `mdi()` will spend the majority of its time waiting. If this program is running in a real-time environment, this function should have hooks to return control of the CPU to the multitasking kernel.

```
c = readch();
  // returns special codes defined by us
  // for arrow keys, page_up, and so on.

 switch(c)
{
 case UP_ARROW:
   if(curr_selection !=0)
     {
     curr_selection--;
     draw_menu(curr_menu, curr_selection);
     }
   break;

 case DOWN_ARROW:
   if(curr_selection < (curr_menu->
     num_selections-1))
     {
     curr_selection++;
     draw_menu(curr_menu, curr_selection);
     }
   break;

 case ENTER:
   (*curr_menu->selection
     [curr_selection].function)
     (curr_menu->selection
       [curr_selection].fn_arg);
   break;

 }

}
```

The `draw_menu()` function not only puts the selections on the screen, it also shows which selection is currently active using inverse text mode or a side arrow.

The screen flicker caused by redrawing the menu when a selection is changed using the arrow keys can be annoying. A function that deselects the current selection and highlights the new selection (a single function, called with FORWARD or BACKWARD arguments) is more visually appealing.

EDITORS

The same menu structures can be expanded to make simple editors. Rather than having an action function, an editor menu selection will have a key handler to take console input and store it in memory and, optionally, a prompt function to take current settings in memory and construct a string to display to the user:

```
typedef struct _selection
{
char       prompt[MAX_PROMPT_LEN];
int        (*function)(int);
           /* Action function */
int        fn_arg;
char *     (*pro_function)(int);
           /* Prompt function */
int        pro_fn_arg;
int        (*key_function)(int);
           /* Key handler */
int       key_fn_arg;
}SELECTION;
```

Assume that our ATM will now allow the client to renew a driver's license by entering appropriate information (name, licence number, address). This code is shown in Listing 2.

The key handler and prompt function arguments will generally be the same and can probably be merged to save space in the menu. The functions `display()` and `keyedit()` themselves will be basically large `switch()` statements, switching on the prompt/key function argument.

```
typedef struct      _selection
{
char      prompt[MAX_PROMPT_LEN];
int       (*function)(int);
          /* Action function */
int       fn_arg;
char *    (*pro_function)(int);
          /* Prompt function */
int       (*key_function)(int);
          /* Key handler */
int       pro_key_fn_arg;
}SELECTION;

/* Global variables */

char      global_curr_name[80];

int       global_curr_lnum;

char      global_curr_address[80];
```

```
char     workbuf[80];
         // scratchpad I/O buffer

char     *         prompt
(
  int    state
)
{
switch(state)
  {
  case NAME:
         strcpy(workbuf,
global_curr_name);
         break;

  case LNUM:
         sprintf(workbuf, "%d",
         global_curr_lnum);
         break;

  case ADDRESS:
         strcpy(workbuf,
         global_curr_address);
         break;
  }

return(workbuf);
}
```

The `draw_menu()` function now loops through all the selections, displays the prompt string (as before), and checks for the existence of a prompt function. If it does exist, the system will call the prompt function, which will return a string in the port's working buffer (`workbuf`). This string is then displayed

LISTING 2
License renewal menu.

```
MENU menu[] =
{
  /* Previous menus here */

 {RENEW_LICENSE_MENU,       3,
  {"Enter Name:",           NULL,       0,      prompt, NAME,    keyedit,NAME},
  {"Enter License Number:", NULL,       0,      prompt, LNUM,    keyedit, LNUM},
  {"Enter Address:",        NULL,       0,      prompt, ADDRESS,keyedit, ADDRESS},
  {"Quit, Back to Main Menu",           goto_menu(),MAIN_MENU, NULL,0,      NULL, 0}
  },

  /* Other menus here */
};
```

in the correct area of the screen:

```
void      draw_menu
(
MENU      *           menu
)
{
  int     selection_num;
  SELECTION          *       selptr;

  /* Clear the screen */
  cls();

  /* Loop through the actual number of
    selections in THIS menu */

  for(selection_num=0; selection_num
    menu->num_selections; selection_num++)
    {
    /* Create a pointer for convenience */

    selptr = &menu- >selection
      [selection_num];
    /* If this is the "active" menu selec-
tion (determine by referencing the global
variable `active_selection´) then draw it
in inverse text. */

    if (selection_num == active_selection)
      inverse(ON);

    /* Print the prompt string */
    setcur(selection_num + MENU_INDENT_Y,
      MENU_INDENT_X);

    printf(selptr-->prompt);

    /* If this selection has a prompt
       function, use it! */

    if(selptr->pro_function)
      {
      /* Print the prompt string */
      setcur(selection_num + MENU_INDENT_Y,
        PROMPT_INDENT_X);
      printf((*selptr->pro_function)
        (selptr->pro_key_fn_arg));
      }

  /* Turn inverse back off if need be */

  if (selection_num == active_selection)
    inverse(OFF);
    }
  }
```

The mdi() function is expanded, as shown in Listing 3, to call the keyhandler input function only after valid input has been completed.

Now that a basic menu system is in place, how does the programmer handle additional requirements? This very generic approach to menus is well suited for add-on features.

PASSWORD PROTECTION

An integer variable can be added to each SELECTION structure to designate a privilege access level. The user's current login level can be checked whenever the user tries to edit or activate that selection. If the user's access is too low, a message can be generated. It is also possible to modify the draw_menu() function to check the user's current access against the access of each selection before it is drawn, and not draw that selection if the user's access is too low. Thus, a low-level user will not even see the higher-access selections.

FEATURE CONTROL SWITCHES

In some applications, the programmer develops one main line of executable code for all clients but may want to charge extra for certain software options. It is undesirable to create and test a copy of the software for each possible combination (disabling or enabling options using compilation switches). The ideal route would be to create and test one copy of the software for all clients and to enable or disable access to items not purchased.

If the hardware has some nonvolatile memory, such as a small EEPROM, a bitfield can be stored indicating which options the client has purchased. When the client wants to upgrade to add a feature, a new EEPROM can be shipped. The bitfield in the EEPROM is then compared against a features bitfield add to each SELECTION, and if the appropriate bits are set, the selection is displayed:

Text Interfaces for Embedded Systems

```
typedef   struct    _selection
{
  ....    // original structure
int  features_bitfield;
}SELECTION;

enum  _feature_bits
{
  FEATURE_ATM_VIDEOPHONE=0,
  FEATURE_ATM_RENEW_LICENSE,
  FEATURE_ATM_ORDER_MOVIE
};

#define  FEATURE_ALL 0
#define  FEATURE_ATM_VIDEOPHONE_BIT
          (0x01 << FEATURE_ATM_VIDEOPHONE)
#define  FEATURE_ATM_RENEW_LICENSE
          (0x01 <<
          FEATURE_ATM_RENEW_LICENSE)
#define  FEATURE_ATM_ORDER_MOVIE
          (0x01 << FEATURE_ATM_ORDER_MOVIE)

MENU  menu[] =
{
  {  MAIN_MENU,  6,
  {"Perform A Deposit", goto_menu(),
     DEPOSIT,
    FEATURE_ALL},
  {"Perform a Withdrawl",  goto_menu(),
    WITHDRAWL,
    FEATURE_ALL},
  {"Obtain a Balance Statement",
    goto_menu(),  BALANCE,
    FEATURE_ALL),
  {"Make A Video Phone Call",
goto_menu(),  BALANCE,
    FEATURE_ATM_VIDEOPHONE_BIT),
  {"Watch A Movie Right Here",
goto_menu(),  BALANCE,
    FEATURE_ATM_VIDEOPHONE_BIT),
  {"Quit",  logout(),  0}
  },

/* Other Menus Here */

};
```

Finally, embed the following condition in the `draw_menu()` function as a requirement to draw each selection:

```
  if(!curr_menu->selection
[selection_loop_ctr].feature_bitfield ||
    curr_menu->selection
[selection_loop_ctr].feature_bitfield &
eeprom.feature_bitfield)
    {
    /* Draw the selection */

    }
```

If the programmer has adapted the approach of using a `hilite()` function to move the highlight selection, this function will also need to look at the features switches to determine which

LISTING 3
Expanded version of mdi.c.

```
void mdi()  /* Menu driven interface */
{
char    c;
MENU    *curr_menu =
          find_menu(MAIN_MENU);
int  curr_selection = 0;
int  status;

draw_menu(curr_menu, curr_selection);

while(TRUE)
  {
  c = readch();   // returns special
codes defined by us for arrow keys,
page_up, etc

  switch(c)
  {
  case UP_ARROW:
    if(curr_selection !=0)
    hilite(FORWARD);

    break;

  case DOWN_ARROW:
    if(curr_selection <
      (curr_menu->num_selections-1))
    hilite(REVERSE);
    break;

  case ENTER:
    (*curr_menu->selection
      [curr_selection].function)
    (curr_menu->selection
      [curr_selection].fn_arg);
    break;

  default:
    if(curr_menu->selection
[curr_selection].key_handler != NULL)
    {
    if(get_input())
/* false if exited by ESC or CTRL-C,
returns true otherwise */
      {
      status =   (*curr_menu->
selection[curr_selection].key_handler)
      (curr_menu->selection
[curr_selection].pro_key_fn_arg);
      }
    else
      {
      redraw_prompt();
      }
    }
    /* Else, ignore key input entirely
or feedback error to user */
  }

int  keyedit
 (
  int  state
)
{
int  okay = TRUE;

switch(state)
  {
  case NAME:
    if(validate_name(workbuf))
    strcpy(global_curr_name,workbuf);
    else
    okay = FALSE;
    break;

  case LNUM:
    if(validate_lnum(workbuf))
    global_curr_lnum = atoi(workbuf);
    else
    okay = FALSE;
    break;

  case ADDRESS:
    if(validate_address(workbuf))
    strcpy(global_curr_address,workbuf);
    else
    okay = FALSE;
    break;
  }
return(okay);
}
```

prompt is the displayed next and previous prompt.

RETURN VALUES

In the original design, all key handler and action functions returned a status. The preceding examples only show a range of two possible values, `OK` and `NOT_OK`. It would be better to provide the user with more specific feedback. The programmer can predefine a number of error codes and even an array of strings listing the appropriate error messages. The key handler functions can then return those specific error codes, and the `mdi()` function can convert those codes into appropriate error strings via a lookup table.

The preceding examples had menu selections for return to the previous menu or directly to the home menu. In actuality, this a waste of a selection and is counterintuitive to users of DOS applications, who often expect the ESC key or F10 key to return them to the previous selection. The `switch()` statement in `mdi()` can certainly look for such a key, but the program needs to know which menu to return to. The solution is to add a parent menu ID code to each MENU structure. The programmer can then designate the backward link for each menu item.

It is often handy to have a global variable to designate a parent menu ID as well. Programmers may find instances where one submenu can be reached by three or four menus. When the user types ESC, the program needs to come back to the correct menu. In this case, the true parent is not known at compile time. So, when entering that submenu, set an alternate parent menu ID variable in RAM. The ESC handler in `mdi()` will check to see if that variable is set first. If it is set, it goes to that menu ID and clears the variable. If it is not set, it looks into the MENU structure for the active menu and uses the hardcoded parent menu ID.

The previous examples of editors assumed that the edit field width was small enough to put on the end of the line after the prompt string. If prompt strings are 30 characters, and we allow some space before reaching the edit area, we are limited to 40 to 50 characters for an 80-column terminal. This may not be sufficient for some applications.

One solution is to specify a field width in the SELECTION structure for each selection containing prompt and key handler functions (zero for noneditor fields). The `draw_menu` function could then be designed to handle longer fields by putting the additional data on extra lines. The programmer must be cautious because now a menu with 10 selections may take 24 lines of a page if many selections are longer than a single line. Another programmer might add a new selection that uses three lines, and now the menu no longer fits on the screen even though it only contains 11 selections.

If edit fields longer than 40 characters are the norm rather than the exception, the menu design should probably be rethought with this in mind.

DYNAMIC MENUS

In most applications, the menu prompt strings and actions will be defined by the programmer at compile time. However, there are occasions where the user may want a menu whose prompts are dynamic. Using our ATM license-renewal example, we might prompt the user for names of immediate relatives in one menu and go to another editor menu to query the user as to what the exact relation is for each name given:

Barney Williams:	Father
Eleanor Williams:	Mother
Michael Tate:	Husband
Kelly Williams:	Sister
Robert Williams:	Brother

In this case, the prompt strings are dynamic. Since the `mdi()` function does everything through a pointer to the active menu, we can dynamically allocate a MENU structure, fill in the appropriate fields (number of selections, prompts, access rights, handler functions), and point the active menu pointer to that block of memory.

In this situation, special care must be taken to properly maintain the parent menu ID and handle the backup (ESC) key. The programmer might create a standard that a parent menu ID code of zero indicates a dynamic menu, signaling that the `active_menu_ptr` is a pointer to memory that must be deallocated before going to the parent menu (but don't deallocate until you have read the parent menu ID).

THE MENU STRUCTURE

One disadvantage of the listed implementation is that a MENU structure will take the same amount of memory, no matter if the menu has two prompts or all 25. To conserve space, the programmer could create several MENU `typedefs` that vary only in terms of the maximum number of allowed prompts (`TINY_MENU, SMALL_MENU, MEDIUM_MENU, LARGE_MENU,` and so on). The programmer would then put each menu into the appropriate variable definition, depending on the number of prompts available. Check Listing 4 for the corresponding code.

For a system that executes out of PROM, it is typically desirable to put the `MENU`[] variables into PROM, inhibit their copy, and later reference in RAM. This can be accomplished by making a single C file (`menus.c`) that does nothing but define or include the header file that defines the menu variables. Compile that file with directives that make the resulting object file a unique segment. The code locator can then be told not to automatically copy that data segment to RAM.

This implementation of a text-based menu system provides an interface that is scalable in terms of the number of menus as well as the complexity of features. The breakdown of menus as an

LISTING 4
Expanded menu management.

```
typedef struct _medium_menu
{
  int  id;
  int  num_selections;
  SELECTION  selection[12];
}MEDIUM_MENU;

typedef struct _small_menu
{
  int  id;
  int  num_selections;
  SELECTION  selection[6];
}SMALL_MENU;

typedef struct _tiny_menu
{
  int  id;
  int  num_selections;
  SELECTION  selection[3];
}TINY_MENU;

/* This somewhat complicates the
goto_menu function, but not much */

void  goto_menu  /* Set & display
active menu */
(
  int  new_menu_id  /* new menu ID */
)
{
  MENU  *  menu;
  int  i,j;
  extern  MENU * curr_menu;

  menu = findmenu(new_menu_id);

  if(menu == NULL)
    {
    /* System error */
    }
  else
    curr_menu = menu;  /* curr_menu is
now a global or external variable */

  draw_menu();

  return;

}

////////////////////////////////////////
MENU  *  findmenu
(
  int  new_menu_id  /* new menu ID */
)
{
  MENU  *  ptr;
  int  i,j;

  /** First, try to find menu in long
menu list **/

  for(i = 0; i < MENU_END; i++)
    {
    if(new_menu_id == menu[i].menu_id)
    {
    ptr = (MENU *)&menu[i];
    break;
    }
    }

  /** If not found, try to find menu in
short menu list **/

  if(new_menu_id >= MENU_END)
    {

for(i=MENU_END+1,j=0;i<MEDIUM_MENU_END;
i++, j++)
    {
    if(new_menu_id ==
medium_menu[j].menu_id)
    {
    ptr = (MENU *)&medium_menu[j];
    break;
    }
    }

    if(new_menu_id >= MEDIUM_MENU_END)
    {
    for(i = MEDIUM_MENU_END+1, j = 0; i
< SHORT_MENU_END; i++, j++)
    {
    if(new_menu_id ==
short_menu[j].menu_id)
    {
    ptr = (MENU *)&short_menu[j];
    break;
    }
    }

    if(new_menu_id >= SHORT_MENU_END)
    {
    for(i=SHORT_MENU_END+1,j=0;i <
TINY_MENU_END; i++, j++)
    {
    if(new_menu_id ==
tiny_menu[j].menu_id)
    {
    ptr = (MENU *)&tiny_menu[j];
    break;
    }
    }
    /** Error if no match found in two
lists! **/

    if(new_menu_id >= TINY_MENU_END)
    {
    return(NULL);
    }
    }


    }
    }

  return(ptr);

}
```

array of selections makes the menu structures easy to modify, and lends itself well to code reuse. Much of the code in the key handler and prompt functions will be identical, referring to other system variables. Adding a new handler is a quick cut-and-paste operation. In fact, the development manager will probably begin to see more bugs caused by a programmer cutting and pasting code and forgetting to modify the underlying code for the new need. The reuse advantages typically outweigh this annoyance, however, especially if everyone involved is made aware of the pitfall. **ESP**

Steve Drevik is a senior project engineer at Environmental Systems Corp. in Knoxville, TN. He has worked in embedded systems development since 1988, developing data acquisition and handling platforms for national and international markets. He holds a MS degree from the University of Tennessee in electrical engineering. He can be reached electronically at thedrev@aol.com.