Using the Amulet display with the Rabbit microcontroller

Jonathan Somers
Nov 18, 2002

One of the main criteria in choosing an embedded platform is the time to develop a user interface. When an application calls for a graphical interface, embedded developers generally have to choose between designing their own graphic library or licensing a third-party library (or an entire operating system). In either case, the developer then has to consider the impact of the user interface on CPU bandwidth and memory requirements. For smaller applications, the size and bandwidth of the graphic library can easily outweigh the rest of the application. These can be nontrivial considerations, especially in a real-time system.

The Amulet LCD controller is an efficient solution to these problems. By offloading the graphical interface to a dedicated coprocessor, the embedded systems designer can focus on real-world interface issues. The Amulet controller manages its own flash memory that contains the text, graphics, and navigational cues for every page of an application. The Amulet not only renders the interface on the LCD, but also collects user input via a touch overlay – so the user can navigate through major portions of the application without the host CPU's assistance. The U/I designer composes the application pages in a subset of HTML, so GUI implementation time is minimal.

In this paper, I discuss the steps needed to design an embedded system using an Amulet 5.7" LCD display as the GUI for the Rabbit R2000 microcontroller. The Rabbit is available both as a chip and as several pre-built modules. In many cases, you can quickly assemble a prototype by wiring a Rabbit module to an Amulet display – little more than a plumbing exercise. I'll discuss both hardware and software design issues, and you can download an easy-to-use communications library to get your Rabbit module and Amulet display talking in a snap.

Hardware Interface

Amulet Technologies offers a 5.7" Starter Kit (Amulet part # STK-GT570) with an RS-232 interface for rapid development from a PC for Proto typing. Contact Amulet Technologies, LLC if you wish to have your Rabbit solution designed into a custom board. You can draws power from your host board without the need for a separate wall supply. Your application simply snaps onto the back of this Amulet module using a pair of box-style 0.100" grid connectors. Figure 1 shows the mechanical layout of these connectors; reference markers denote the locations of pin 1, and are relative to the upper left corner of the board as viewed from behind.

J12 is a 16-pin single inline connector that provides supplemental functions like contrast and backlight control. You also supply both main power and backlight power through this connector. Pin 1 supplies regulated 5V to the Amulet chip and the LCD; maximum rated current drain on this pin is 0.5A. Pin 3 powers the backlight. The simplest way to do this is to use a p-channel FET that is driven by a parallel I/O pin on the Rabbit. Figure 2 illustrates this.

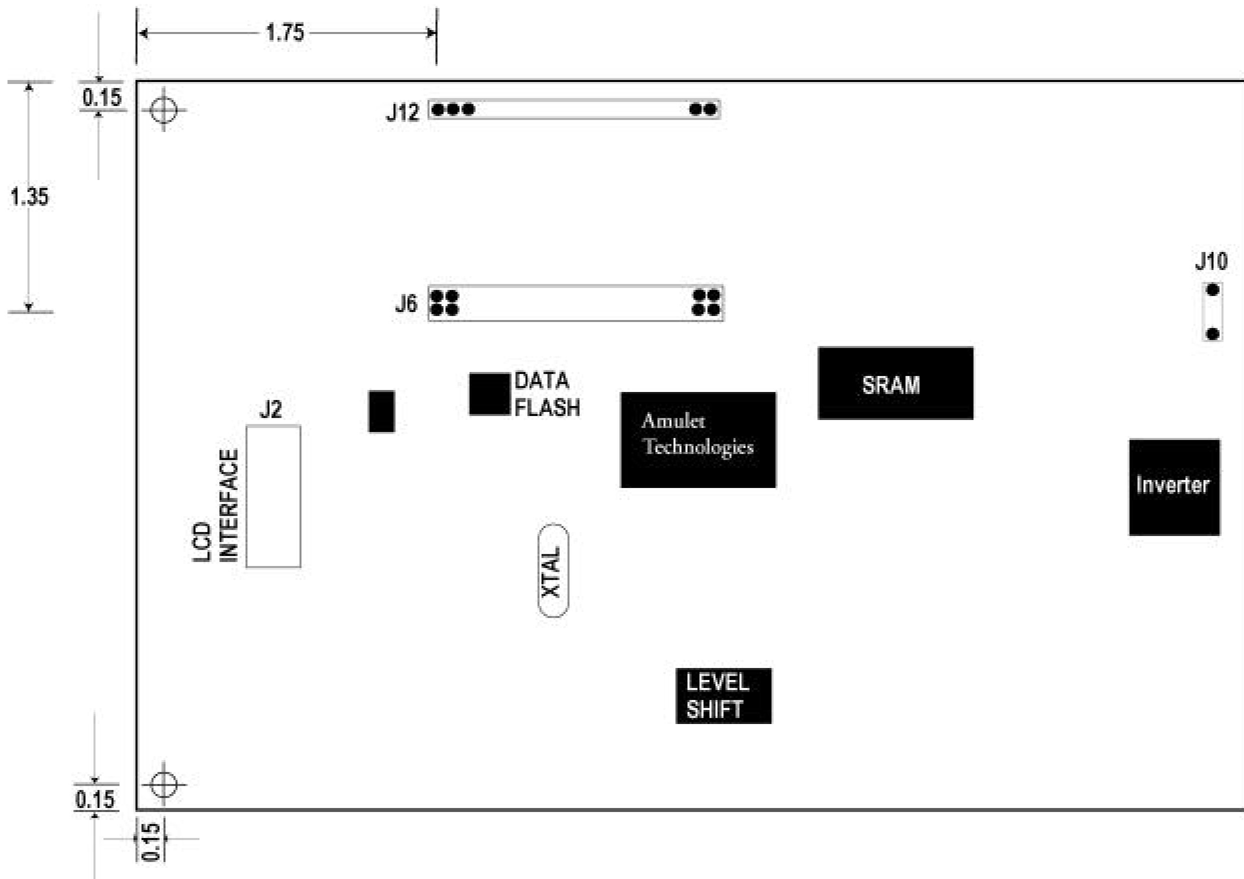

Figure 1. Location the connectors on the Amulet controller board.

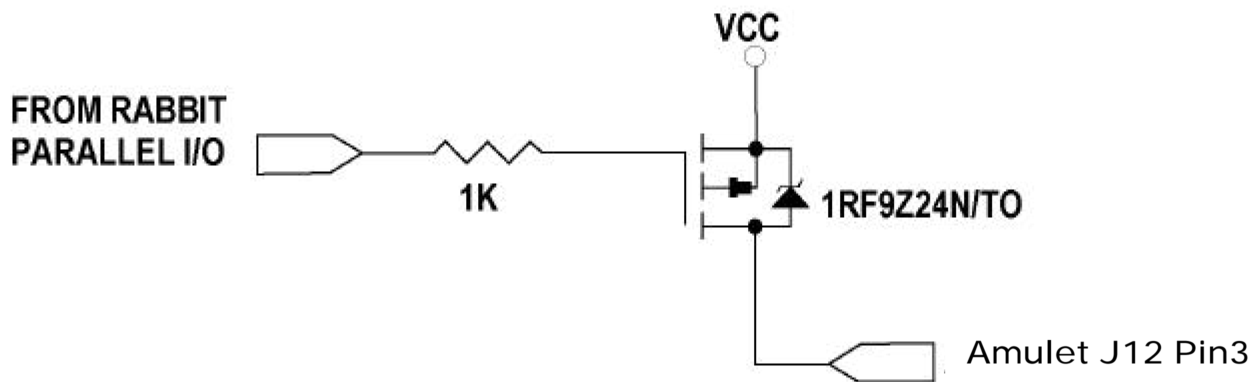

Figure 2. Controlling the LCD backlight from the Rabbit

The application designer can then shut off the backlight when the user touches a soft button on the display to put the unit to sleep, or when a period of inactivity occurs. In either case, the application should switch to a "tap anywhere" screen – basically a single button the size of the LCD that generates a Remote Procedure Call (RPC) to turn the backlight back on again. Alternately, you can provide an external momentary pushbutton for on/off control.

Pins 11, 12, and 13 on J12 are normally wired to a trimpot for contrast adjustment. Amulet prescribes a 20K trimpot, although I have used a 25K trimpot with no trouble. If you have sufficient free I/O pins available, you may want to consider designing in a digital trimpot and putting the display contrast under software control. If you plan to control contrast through an on-screen panel, be sure to limit the range of the digital pot so that the display is always legible under a range of temperature conditions, or else provide some means of temperature compensation. Otherwise your user won't be able to find the soft contrast buttons!

J6 is a 32-pin dual-row connector that contains the digital control and communications lines. Note that the Amulet chip uses 3.3V logic levels, and is not 5V tolerant. While the Rabbit 2000 processor can be operated at 3.3V, the Rabbit 2000-series modules are all designed to operate at 5V. If you drive the Amulet display from a Rabbit module, be sure to use a 5V/3.3V logic buffer. If you are designing the Rabbit CPU directly into your own board and you can use a single 3.3V supply, no level shifting is needed.

Pins 15, 17, 19, and 21 on J6 are used to set the configuration of the display. On the Amulet module, these pins are wired to a four-position DIP switch; closing a switch shorts the corresponding pin to ground, whereas opening the switch leaves it pulled up.

Pin 15 is used to set the powerup mode. This is useful if you plan to give the Rabbit processor the ability to reflash the Amulet display during an application upgrade. The pin should be at 0V for normal operation, and should be set to 3.3V for a reflash.

Pin 17 controls the flash programming rate; this pin is tied it to a 3.3V pullup within the Amulet chip to fix the programming rate at 115.2kbps. Pin 19 forces the Amulet to perform a memory test at powerup, but the results are only available on a test pad and can't be detected by the Rabbit; this pin is also tied to a 3.3V pullup inside the Amulet chip. Pin 21 will force the Amulet to perform a calibration at powerup, but you can incorporate a software-initiated calibration into your app anyway, so you really don't need to use the hardware feature to do this; tie this pin to ground for normal operation.

The other pins you will need are 25 and 26, which are the Amulet transmit and receive pins, respectively. Communications with the Amulet display is through an asynchronous serial interface at baud rates up to 115.2kbps. Because Rabbit serial ports A and B support both sync and async operation, you may want to choose port C or D for your Amulet connection since they support only asynchronous connections – this will leave your synchronous ports free for other purposes. The communications library in this article uses port C by default, but you can easily modify it to use any Rabbit port.

In early development, it is often useful to change the Amulet's serial connection from the Rabbit to your development PC. Figure 3 shows a simple eight-pin jumper block connection that allows you to connect the Amulet to your PC, the Rabbit to your PC, or the Rabbit to the Amulet by simply moving a pair of jumper shunts. Don't forget to properly buffer the Amulet transmit and receive lines if you are running the MAX232A and your Rabbit at 5V.

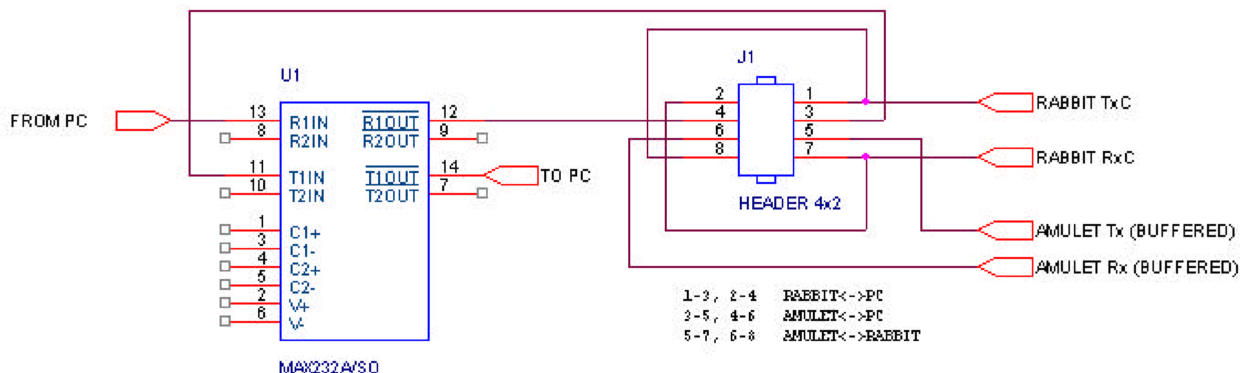

Figure 3 – Jumper shunts for PC/Rabbit/Amulet interconnections

Serial communications (See Appendix A for complete source code)

The Amulet uses a very simple serial protocol to communicate with an attached device. In this protocol, the Amulet behaves as a master device and the Rabbit acts as an application “server” that responds to display requests. In other words, the Rabbit does not tell the Amulet which screen to display – instead, the Amulet tells the Rabbit which screen it is on, and the Rabbit accepts, processes, and provides information as directed by the user. This can be a confusing model for developers who are not used to user-driven application architectures, but in practice it is not very difficult and works quite well.

Packets transmitted between the Rabbit and the Amulet do not use fixed block sizes or leading and trailing start-of-message markers. This rules out the use of Dynamic C’s block communications libraries. The serCread() and serCwrite() functions, or their cofunction equivalents, still do the job quite nicely. Amulet’s online documentation includes a simple example of a state machine written in C that collects characters and dispatches requests as each packet type is recognized and completed.

The Amulet can make five basic types of requests:

- It can read any one of 256 bytes from the Rabbit.
- It can write any one of 256 bytes to the Rabbit.
- It can read any one of 256 16-bit words from the Rabbit.
- It can read any one of 256 null-terminated strings from the Rabbit. The documentation claims that strings may be up to 255 characters in length, although Amulet firmware version 2.1.9 only supports strings up to 128 characters in length for now. However there is nothing in the protocol that prohibits strings even longer than 256 bytes at a future date.
- It can request any one of 256 remote procedure calls from the Rabbit. These RPCs are extremely simplistic: the Amulet supplies no arguments (although some clever tricks are possible by using the function number as an argument) and the Rabbit returns no value. “Event handlers” might be a better term since the Amulet is really just notifying the Rabbit that a particular UI event has occurred.

The address space of 256 bytes, words, strings, and RPCs is just a mutually agreed upon abstraction. The Amulet does not know or care where the variables and procedures are located in the Rabbit’s memory, or even if they are in memory at all. For example, when the Amulet asks for byte value 24, the Rabbit may choose to calculate a value on the fly (from the real-time clock, for example) instead of retrieving a value from a memory location. This flexibility can be very useful for certain types of application programming.

Unfortunately, your Rabbit application will be written in C and your Amulet application will be written in HTML. This means that you will have a double maintenance problem since you will need to keep your HTML constants in sync with your C source code. For example, you may create a table of named constants in your C source for several RPC calls:

```
#define RPC_OPEN_RELAY       0x20
#define RPC_CLOSE_RELAY      0x21
```

Likewise you may create corresponding RPC requests in your Amulet HTML code:

```
<PARAM NAME=href VALUE="Amulet:UART.invokeRPC(0x20)">
<PARAM NAME=href VALUE="Amulet:UART.invokeRPC(0x21)">
```

If you (or anyone maintaining your code in the future) ever decide to change the value of RPC_OPEN_RELAY for any reason, you will almost certainly need to comb through all of your HTML source and change the RPC 0x20 calls as well. This is further hampered by the fact that your Amulet uHTML code does not support symbolic constants.

You have two choices:

1. Write a custom preprocessor that lets you define a table of symbolic constants and apply them to both your C source code and your HTML code prior to compilation, or
2. Maintain a design document with tables of variable and RPC locations, and refer to it often.

The design document is a good practice in any case, so unless you really enjoy building preprocessors, you will probably opt for that route.

Variable pool interface (See Appendix B for complete source code)

To simplify the task of communicating with the Amulet, I developed a library that implements a variable pool interface. With this interface, you need not hassle with the parsing of communications requests and you can share information between the Rabbit and the Amulet through a relatively opaque API.

The easiest way to understand the variable pool interface is to follow the progression of an application from a simple prototype through to a complete, robust implementation. When getting started with the Amulet, the first thing I did was write just enough communications code to get one or two byte variables and display them as numeric fields on the LCD. This relatively hard-coded approach works well if you are only dealing with one or two values. However, it falls apart rather quickly once you outgrow that initial scale.

The Amulet's architecture of 256 bytes, words, strings, and RPCs suggests a very simple implementation scheme. The logical choice is to write:

```
unsigned char auchBytes[ 256 ];
unsigned int auiWords[ 256 ];
psz apszStrings[ 256 ];
pfRPC apfRPCs[ 256 ];
```

Any time the Amulet needs a byte, the async code simply reads it from the corresponding slot in the byte array; anytime the Rabbit application needs to update a variable that is shared with the Amulet, it refers to the same slot in the same byte array. For applications with only a few variables, of course, you could replace 256 with a smaller constant to avoid wasting unused variables (provided that you number them consecutively starting from zero).

Many developers will not need to go beyond this level of sophistication. In some cases, however, it is inconvenient to have the Rabbit keep updating values in these arrays. Suppose your application has a record structure:

```
typedef struct _itemInfo
{
	char name[ 128 ];
	char qty;
	int sku;
} itemInfo, *pitemInfo;
itemInfo currentItem;
```

Anytime you want to display information about `currentItem` on the display, you must copy `currentItem.qty` to a location (say location 23) in `auchBytes`, and also copy `currentItem.sku` to a location (say location 37) in `auiWords`. You would also have to set a pointer in `apszAmuletStrings` to point to `currentItem.name`, but then as long as the location of `currentItem` does not change, you would not need to update the pointer in `apszAmuletStrings`. Anytime `currentItem` is modified, `apszAmuletStrings` will automatically point to the new info.

This suggests the next level of refinement. Suppose we add an array of pointers to each of the 256 bytes and words:

```
unsigned char *apuchBytePtrs[ 256 ];
```

```
unsigned int *apuiWordPtrs[ 256 ];
```

At startup, we could initialize these arrays:

```
int i;
for ( i = 0; i < 256; i++ )
{
      apuchBytePtrs[ i ] = &auchBytes[ i ];
      apuiWordPtrs[ i ] = &auiWords[ i ];
}
```

By default, each pointer would then point to the corresponding byte or word. Now suppose your application needs to override just a handful of those pointers to point to fields in your record:

```
      apuchBytePtrs[ 23 ] = &currentItem.qty;
      apuiWordPtrs[ 37 ] = &currentItem.sku;
```

Now the Amulet will always get the correct display information from `currentItem` without copying data back and forth; in fact, your application need not take any further action at all. For complex structures, this can be quite a savings.

As mentioned previously, there are times when you don't even have a value stored in memory. Perhaps you want to display the status of a bit on a parallel I/O port, or read a value from an A/D converter. Alternately, you may not want to generate the value, but you may want be *notified* that the Amulet is asking for the information so that you can do some related processing. In these cases, your application needs to be notified that a value is being read or written. Again we can create an array of pointers, but this time we are creating pointers to *functions*:

```
      typedef BOOL (*pfGetByte)();
      typedef BOOL (*pfGetWord)();
      pfGetByte apfGetByte[ 256 ];
      pfGetWord apfGetWord[ 256 ];
```

Note that the Dynamic C compiler used for Rabbit development doesn't actually care about the prototype for a pointer to a function. Again we can initialize these arrays to point to default accessor functions:

```
BOOL VarGetByteDefault(
      unsigned char uchWhich,
      unsigned char *puchValue
){
      if (
            ( puchValue != NULL ) &&
            ( apuchBytePtrs[ uchWhich ] != NULL )
      ){
            *puchValue = *apuchBytePtrs[ uchWhich ];
            return TRUE;
      }
      return FALSE;     // no value is defined
}

root BOOL VarGetWordDefault(
      unsigned char uchWhich,
      unsigned int *puiValue
){
      if (( puiValue != NULL ) && ( apuiWordPtrs[ uchWhich ] != NULL ))
      {
            *puiValue = *apuiWordPtrs[ uchWhich ];
            return TRUE;
```

```
    }
    return FALSE;     // no value is defined
}
```

Why set up the functions to return `BOOL`? One example is the case of the Amulet graph widget. The Amulet graph widget will continue to loop forever, asking for values as long as the Rabbit is willing to supply them. The on screen graph simply wraps around and around, overwriting old values with new ones. However, suppose you just want to draw a graph one time, containing six values? The only way to do this is to reply to the first six byte requests from the graph widget, and then acknowledge all further requests from the widget without supplying data. The first six times your own get-byte function is called, it should return `TRUE` – this tells the communications code that a valid value was obtained, and it sends the appropriate message to the Amulet. Each subsequent time, your function should return `FALSE` – this tells the communications code that no valid value was available, and so it will just ACK the request without supplying a value.

What about the argument `uchWhich` in the get byte function? This simply provides your function with the index of the value that the Amulet display is asking for. Instead of writing a separate function for a block of related values, you can write one function and use a switch statement to handle any special processing. For example, say you have an Amulet uHTML screen that displays the Rabbit's real-time clock. You could write one get-byte function to handle three values:

```
xmem BOOL GetByteTime(
    unsigned char uchWhich,
    unsigned char *puchValue
){
    unsigned long ultime;
    struct tm time;

    ultime = read_rtc();
    mktm( &time, ultime );

    switch( uchWhich )
    {
        case GET_HOUR:
            if ( time.tm_hour == 0 )
                time.tm_hour = 12;
            else if ( time.tm_hour > 12 )
                time.tm_hour -= 12;
            *puchValue = time.tm_hour;
        break;

        case GET_MINUTE:
            *puchValue = time.tm_min;
        break;

        case GET_AMPM:                // 0 for AM, 1 for PM
            *puchValue = ( time.tm_hour > 11 );
        break;
    }
    return TRUE;
}
```

This is easier to read and maintain, and more compact, than three separate functions that each make calls to `read_rtc()` and `mktm()`.

You can use a similar technique with the RPC function calls. Suppose you want a soft keyboard for your application, to allow the user to enter their name. You could write one RPC handler function for each key on the soft keyboard, or you can write:

```
root void RpcKeystroke( unsigned char uchWhichRPC )
```

```
{
      if ( iUserNameLen < ( NAME_LENGTH - 1 ) )
      {
            newuser.achName[ iUserNameLen++ ] = uchWhichRPC;
            newuser.achName[ iUserNameLen ] = 0;
      }
}
```

If you assign the key 'A' to RPC 0x41, 'B' to RPC 0x42, and so on, then the RPC value is the ASCII value of the key that was typed. The function above simply adds the incoming character (in the guise of the RPC function number) to the name string. In practice, of course, you might also want to handle a backspace or perform other processing, either in this function or in a separate RPC call.

The source code for the serial communications handler is found in Appendix A. The source code for the variable pool interface, with complete Dynamic C, is found in Appendix B.

Concluding remarks

The Amulet LCD controller and the Rabbit R2000 microprocessor are a well-matched combination. Unlike PICs or other tiny micros, the Rabbit has a large memory space and plenty of I/O and communications to tackle larger, more complex tasks. These tasks frequently call for a more sophisticated interface than lightweight embedded applications. Amulet's uHTML approach allows a developer to create attractive user interfaces – even moderately complex interfaces – quickly and robustly. Together, they open a new range of applications on a shoestring engineering budget.

Jonathan Somers is an embedded systems designer. He can be reached at jon_somers@bellsouth.net.

Appendix A: Serial Communications Source Code — Rabbit/Amulet

```
/*** BeginHeader
      LcdInitialize,
      LcdBacklightOn,
      LcdBacklightOff,
      LcdProcessReceivedCharacters
*/

void LcdInitialize();
void LcdBacklightOn();
void LcdBacklightOff();
void LcdProcessReceivedCharacters();

unsigned char guchCurrentScreen;          // which screen is the application
currently on

/*** EndHeader */

////////////////////////////////////////////////////////////////////////////////
//
// lcd.c - Amulet LCD display controller APIs
//
//    Written by Jonathan Somers, jon_somers@bellsouth.net
// Copy Free and Released to the Public Domain
//
// Once you have modified and compiled this code to meet your needs, simply
// call LcdInitialize() once at startup and then call LcdProcessReceivedChars()
// periodically to service the serial port.  To comply with the Amulet timeout
// of 200ms, be sure to call at least that frequently for reliable performance.
//
//    An important note about this code - By convention, I always use RPC
// calls 128-255 to signal which page of the application is displayed.  In
// this code, you will see references to the backlight timeout.  Each time
// a new page is displayed, I reset the backlight timer.  I also store the
// current page # in guchCurrentScreen for reference.
//
// You are absolutely not required to use this RPC scheme - it is just
// a convention I have found useful.
//
// Also note that the timer code to actually count down the backlight timer
// is not included in this code, so you can either implement it or comment out/
// remove the irrelevant portions based on your needs.
//
// This code also includes a little bit-twiddling during initialization to set
// the display mode properly.  Depending on your application, you may have
// assigned different I/O pins on the Rabbit for these purposes - or you may
// have omitted them altogether.  Inspect LcdInitialize() carefully to ensure
// it meets your needs.
//
// $Header: $
//
// $History: $
//
////////////////////////////////////////////////////////////////////////////////

//    Messages from the Amulet tend to be very short, but they can accumulate
// so I've fixed the inbuffer size at 63.  The maximum length of an outgoing
// string is 127 characters at present, but the spec offers an unlimited
// potential so you may choose to increase this value from 127 in the future.
// The code will printf() a warning if it encounters such a condition.
```

```
#define CINBUFSIZE 63
#define COUTBUFSIZE 127

// The next global integer stores the current state of the communications state
// machine.  The constants that follow it represent the possible states.

int giCommState;

#define LCD_AWAITING_SOM                              0x00
#define LCD_AWAITING_GETBYTE_ADR_1        0x10
#define LCD_AWAITING_GETBYTE_ADR_2        0x11
#define LCD_AWAITING_PUTBYTE_ADR_1        0x20
#define LCD_AWAITING_PUTBYTE_ADR_2        0x21
#define LCD_AWAITING_PUTBYTE_VAL_1        0x22
#define LCD_AWAITING_PUTBYTE_VAL_2        0x23
#define LCD_AWAITING_GETWORD_ADR_1        0x30
#define LCD_AWAITING_GETWORD_ADR_2        0x31
#define LCD_AWAITING_GETSTRING_ADR_1      0x40
#define LCD_AWAITING_GETSTRING_ADR_2      0x41
#define LCD_AWAITING_RPC_ADR_1                  0x50
#define LCD_AWAITING_RPC_ADR_2                  0x51

//    Send the next string to the Amulet display to reset it to your application's
start page.

const char achReset[] = { 0xA0, 0x02, 0x00, 0x20, 0x3E };

///////////////////////////////////////////////////////////////////////

xmem void LcdInitialize()
{
      int i;

      printf( "lcd initialization begins\n" );

      giCommState = 0;

      ///////////////////////////////////////////////////////////////////////
      //    initialize port C bits 0 and 6 to 0 so that Amulet will power up
      //    in "normal" mode - your implementation may need to change this

      i = PCDRShadow & 0xBE;
      WrPortI( PCDR, &PCDRShadow, i );

      ///////////////////////////////////////////////////////////////////////
      // initialize port D bit 3 for output - set to 0
      //    to turn on the Amulet backlight

      i = PDDDRShadow | 8;                      // ensure output
      WrPortI( PDDDR, &PDDDRShadow, i );

      i = PDDCRShadow & 0xF7;             // ensure not open drain
      WrPortI( PDDCR, &PDDCRShadow, i );

      i = PDDRShadow & 0xF7;
      WrPortI( PDDR, &PDDRShadow, i );

      printf( "lcd bit operations complete - about to set up serial port\n" );

      ////////////////////////////////////////////////////////////////////
```

```
	// initialize comport C to 115200/n/8/1

	serCopen( 115200 );
	serCwrite( &achReset, sizeof( achReset ));
	printf( "lcd initialization complete\n" );
}

root void LcdResetBacklightTimer()
{
	if ( giBacklightTimer == 0 )		// if timer is stopped, backlight is off
	{
		LcdBacklightOn();				// so turn it on
	}
	giBacklightTimer = BACKLIGHT_TIMEOUT;
}

xmem void LcdBacklightOn()
{
	BitWrPortI( PDDR, &PDDRShadow, 0, 3 );
}

xmem void LcdBacklightOff()
{
	BitWrPortI( PDDR, &PDDRShadow, 1, 3 );
}

// The next few values are taken from the Amulet online doc.
// Why const char variables instead of #defines?  Originally I
// wrote the serial transmit code to write directly out of these
// memory locations (you can't pass a pointer to a numeric constant
// as an argument; it must exist in memory if you want to pass a
// pointer to it).  guchAck is still used this way, but the others
// are not.  You can change them to constants now if you like.

const unsigned char guchAck = 0x10;
const unsigned char guchGetByteVarReply   = 0x21;
const unsigned char guchGetStringVarReply = 0x22;
const unsigned char guchPutByteVarReply   = 0x23;
const unsigned char guchPerformRpcReply   = 0x24;
const unsigned char guchGetWordVarReply   = 0x25;

root int HexToNibble( char c )
{
	//    On entry, c is an ASCII alphanumeric hex character.
	// The function returns the byte value of the char, or 0 if invalid.

	if (( c >= '0' ) && ( c <= '9' ))
	{
		return ( c - '0' );
	}
	else if (( c >= 'A' ) && ( c <= 'F'))
	{
		return( 10 + c - 'A' );
	}
	else if (( c >= 'a' ) && ( c <= 'f'))
	{
		return( 10 + c - 'a' );
	}
	return 0;
}
```

```
root int HexToByte( char * ach )
{
	// On entry, achHex points to two ASCII alphanumeric hex characters.
	// I don't even care if they are null terminated or not.
	// The function returns the byte value of the two chars.  No validation is
performed.

	return( HexToNibble( *ach ) << 4 | HexToNibble( *(ach + 1) ) );
}

xmem void LcdGetByteVar( unsigned char uchWhich )
{
	unsigned char uchValue;
	char achMsg[ 6 ];

	if ( !VarGetByte( uchWhich, &uchValue ))		// caller did not supply a
value,
	{
		serCwrite( &guchAck, sizeof( guchAck ));	// so just nak the Amulet
	}
	else
	// caller did supply a value
	{
		sprintf( achMsg, "%c%02X%02X", guchGetByteVarReply, uchWhich, uchValue );
		serCwrite( achMsg, 5 );
	}
}

xmem void LcdSetByteVar( unsigned char uchWhich, unsigned char uchValue )
{
	char achMsg[ 6 ];

	sprintf( achMsg, "%c%02X%02X", guchGetByteVarReply, uchWhich, uchValue );
	serCwrite( achMsg, 5 );
	VarPutByte( uchWhich, uchValue );
}

xmem void LcdGetWordVar( unsigned char uchWhich )
{
	unsigned int uiValue;
	char achMsg[ 8 ];

	if ( !VarGetWord( uchWhich, &uiValue ))		// caller did not supply a
value,
	{
		serCwrite( &guchAck, sizeof( guchAck ));	// so just nak the Amulet
	}
	else
	// caller did supply a value
	{
		sprintf( achMsg, "%c%02X%04X", guchGetWordVarReply, uchWhich, uiValue );
		serCwrite( achMsg, 7 );
	}
}

xmem void LcdGetStringVar( unsigned char uchWhich )
{
	unsigned char * psz;
	char achMsg[ 255 + 5 ];
	int iLen, iWritten;
```

```
	psz = VarGetString( uchWhich );
	if ( psz == NULL )                              // if caller has no value to
supply, just ACK the msg
	{
		serCwrite( &guchAck, sizeof( guchAck ));
	}
	else                                                        // caller supplied
a value, so send it over
	{
		iLen = sprintf( achMsg, "%c%02X%s%c", guchGetStringVarReply, uchWhich,
psz, 0x00 );
		iWritten = serCwrite( achMsg, iLen );
		if ( iWritten != iLen )
		{
			printf( "warning: in LcdGetStringVar, iLen=%d, iWritten=%d\n",
iLen, iWritten );
		}
	}
}

xmem void LcdPerformRpcCall( unsigned char uchWhich )
{
	// first things first - reply to the Amulet to get it on its way

	char achMsg[ 5 ];
	sprintf( achMsg, "%c%02X", guchPerformRpcReply, uchWhich );
	serCwrite( achMsg, 3 );

	VarCallRPC( uchWhich );

	//	was the call between 128 and 255?  this would mean that we have switched
screens -
	//	if so, reset the backlight timer and turn off the rest timer

	if ( uchWhich >= 0x80 )
	{
		guchCurrentScreen = uchWhich;
//		printf( "now on screen %02X... ", guchCurrentScreen );
		LcdResetBacklightTimer();
	}
}

xmem void LcdProcessReceivedCharacters()
{
	unsigned char c;
	unsigned char uchWhich;
	static unsigned char ach[ 2 ];
	unsigned char achMsg[ 32 ];
	int i, j;
	BOOL bDone;

	while ( serCread( &c, 1, 0 ) )
	{
		switch( giCommState )
		{
			case LCD_AWAITING_SOM: // expecting CSOM
				if ( c == 0x11 )              // get byte
				{
					giCommState = LCD_AWAITING_GETBYTE_ADR_1;
				}
				else if ( c == 0x12 )   // get string
```

```
                        {
                              giCommState = LCD_AWAITING_GETSTRING_ADR_1;
                        }
                        else if ( c == 0x13 )   // put byte
                        {
                              giCommState = LCD_AWAITING_PUTBYTE_ADR_1;
                        }
                        else if ( c == 0x14 )   // perform RPC call
                        {
                              giCommState = LCD_AWAITING_RPC_ADR_1;
                        }
                        else if ( c == 0x15 )   // get word
                        {
                              giCommState = LCD_AWAITING_GETWORD_ADR_1;
                        }
                        // else stay parked in LCD_AWAITING_SOM state
                  break;

                  case LCD_AWAITING_GETBYTE_ADR_1:
                        ach[ 0 ] = c;                       // store the first nibble
until the next char comes along
                        giCommState = LCD_AWAITING_GETBYTE_ADR_2;
                  break;

                  case LCD_AWAITING_GETBYTE_ADR_2:
                        ach[ 1 ] = c;
                        uchWhich = HexToByte( ach );
                        LcdGetByteVar( uchWhich );
                        giCommState = LCD_AWAITING_SOM;
                  break;

                  case LCD_AWAITING_GETSTRING_ADR_1:
                        ach[ 0 ] = c;
                        giCommState = LCD_AWAITING_GETSTRING_ADR_2;
                  break;

                  case LCD_AWAITING_GETSTRING_ADR_2:
                        ach[ 1 ] = c;
                        uchWhich = HexToByte( ach );
                        LcdGetStringVar( uchWhich );
                        giCommState = LCD_AWAITING_SOM;
                  break;

                  case LCD_AWAITING_RPC_ADR_1:
                        ach[ 0 ] = c;
                        giCommState = LCD_AWAITING_RPC_ADR_2;
                  break;

                  case LCD_AWAITING_RPC_ADR_2:
                        ach[ 1 ] = c;
                        uchWhich = HexToByte( ach );
                        LcdPerformRpcCall( uchWhich );
                        giCommState = LCD_AWAITING_SOM;
                  break;

                  case LCD_AWAITING_PUTBYTE_ADR_1:
                        ach[ 0 ] = c;                       // store the first nibble
until the next char comes along
                        giCommState = LCD_AWAITING_PUTBYTE_ADR_2;
                  break;
```

```
                    case LCD_AWAITING_PUTBYTE_ADR_2:
                          ach[ 1 ] = c;
                          uchWhich = HexToByte( ach );
                          giCommState = LCD_AWAITING_PUTBYTE_VAL_1;
                    break;

                    case LCD_AWAITING_PUTBYTE_VAL_1:
                          ach[ 0 ] = c;                         // store the first nibble
until the next char comes along
                          giCommState = LCD_AWAITING_PUTBYTE_VAL_2;
                    break;

                    case LCD_AWAITING_PUTBYTE_VAL_2:
                          ach[ 1 ] = c;
                          j = HexToByte( ach );
                          LcdSetByteVar( uchWhich, j );
                          giCommState = LCD_AWAITING_SOM;
                    break;

                    case LCD_AWAITING_GETWORD_ADR_1:
                          ach[ 0 ] = c;                         // store the first nibble
until the next char comes along
                          giCommState = LCD_AWAITING_GETWORD_ADR_2;
                    break;

                    case LCD_AWAITING_GETWORD_ADR_2:
                          ach[ 1 ] = c;
                          uchWhich = HexToByte( ach );
                          LcdGetWordVar( uchWhich );
                          giCommState = LCD_AWAITING_SOM;
                    break;

                    default:
                          printf( "\nunknown comm state %d - rcvd char (%02X)\n",
giCommState, c );
                          giCommState = LCD_AWAITING_SOM;
                    break;
              }
      }
}     // end of ProcessReceivedCharacters

// end of lcd.c
```

Appendix B: Variable Pool Interface Source Code — Rabbit/Amulet

```
/*** BeginHeader VarPoolInitialize, VarGetByte, VarGetWord, VarPutByte, VarPutWord */

      void VarPoolInitialize();

      BOOL VarGetByte( unsigned char uchWhichByte, unsigned char *puchValue );
      BOOL VarGetWord( unsigned char uchWhichWord, unsigned int *puiValue );
      psz VarGetString( unsigned char uchWhichString );
      void VarPutByte( unsigned char uchWhichByte, unsigned char uchValue );
      void VarPutWord( unsigned char uchWhichWord, unsigned int uiValue );
      void VarPutString( unsigned char uchWhichString, psz pszString );

      void VarSetBytePtr( unsigned char uchWhich, unsigned char *puchWhere );
      void VarSetWordPtr( unsigned char uchWhich, unsigned int *puiWhere );

      typedef BOOL (*pfGetByte)();
      typedef BOOL (*pfGetWord)();
      typedef psz (*pfGetString)();
      typedef void (*pfPutByte)();
      typedef void (*pfPutWord)();
      typedef void (*pfPutString)();
      typedef void (*pfRPC)();

      void VarSetFuncGetByte( unsigned char uchWhich, pfGetByte pf );
      void VarSetFuncGetWord( unsigned char uchWhich, pfGetWord pf );
      void VarSetFuncGetString( unsigned char uchWhich, pfGetString pf );
      void VarSetFuncPutByte( unsigned char uchWhich, pfPutByte pf );
      void VarSetFuncPutWord( unsigned char uchWhich, pfPutWord pf );
      void VarSetFuncPutString( unsigned char uchWhich, pfPutString pf );
      void VarSetFuncRPC( unsigned char uchWhich, pfRPC pf);

      void VarCallRPC( unsigned char uchWhich );

/*** EndHeader */

////////////////////////////////////////////////////////////////////////////////
//
// varpool.c - variable space accessible to both Amulet and Rabbit
//    (it also includes the RPC calls too)
//
// $Header: $
//
// $History: $
//
////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////
//    Internal storage for elements of each main data type
// (bytes, words, strings, RPC funcs).

unsigned char auchBytes[ 256 ];
unsigned int auiWords[ 256 ];
psz apszStrings[ 256 ];
pfRPC apfRPCs[ 256 ];

///////////////////////////////////////////////////////////////////////
// Internal storage for ptrs to bytes and words.
//    Not needed for strings or RPCs since these are already pointers.
// On init, these are set to point to the elements of the arrays above,
// but an application can set any element's pointer to point
```

```
//    at any location it chooses

unsigned char *apuchBytePtrs[ 256 ];
unsigned int *apuiWordPtrs[ 256 ];

////////////////////////////////////////////////////////////////////////
// Internal storage for pointers to accessor functions
//    for each main data type.
//    On init, these are set to point to default accessor functions,
//    but an application can set any element's accessor functions
// to point to any compatible functions that it chooses

pfGetByte apfGetByte[ 256 ];
pfPutByte apfPutByte[ 256 ];
pfGetWord apfGetWord[ 256 ];
pfPutWord apfPutWord[ 256 ];
pfGetString apfGetString[ 256 ];
pfPutString apfPutString[ 256 ];

////////////////////////////////////////////////////////////////////////
// Next are the APIs to set the accessor functions above.
// Call these functions to override the built-in accessor functions
// with your own.  They are defined as xmem because they are most
// likely to be called only once during initialization.

/* START FUNCTION DESCRIPTION ****************************************
VarSetFuncGetByte                                          <VarPool.c>

SYNTAX: void VarSetFuncGetByte( unsigned char uchWhich, pfGetByte pf );

DESCRIPTION:
When an Amulet GET_BYTE request cannot be retrieved from a static
variable, but must instead be computed dynamically, use this
function to register your dynamic computation function in the
variable pool interface.

A typical application is a date/time panel, where it is not practical
to constantly update variables in memory with the current time.
Instead, register a function to read the clock and return the current
values of interest only when the Amulet display asks for them.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being requested.  In the date/time example, you could register
the same function for five different byte variables, and then based
on which value is requested, your function can supply the day, month,
hour, minute, or second.

PARAMETER 1: uchWhich - determines which byte variable
requests will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all requests for byte variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

BOOL YourGetByteFunc( unsigned char uchWhich, unsigned char *puchValue );
```

```
although your function can of course have any name.

In your callback function, uchWhich lets your function know which variable
is being requested.  Typically you only need to know this if you are
using the same function to access related variables.

puchValue points to the location where you should store the value
to be returned.

Your function should return TRUE if there is a value to transmit
back to the Amulet display controller.  If your function returns
FALSE, an ACK will be transmitted with no further data.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncPutByte, VarSetFuncGetWord, VarSetFuncGetString
END DESCRIPTION *******************************************************/

xmem void VarSetFuncGetByte( unsigned char uchWhich, pfGetByte pf )
{
      apfGetByte[ uchWhich ] = pf;
}

/* START FUNCTION DESCRIPTION *****************************************
VarSetFuncGetWord                                             <VarPool.c>

SYNTAX: void VarSetFuncGetWord( unsigned char uchWhich, pfGetWord pf );

DESCRIPTION:
When an Amulet GET_WORD request cannot be retrieved from a static
variable, but must instead be computed dynamically, use this
function to register your dynamic computation function in the
variable pool interface.

A typical application is a date/time panel, where it is not practical
to constantly update variables in memory with the current time.
Instead, register a function to read the clock and return the current
values of interest only when the Amulet display asks for them.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being requested.  In the date/time example, you could register
the same function for five different word variables, and then based
on which value is requested, your function can supply the year, month,
day, hour, or minute.

PARAMETER 1: uchWhich - determines which word variable
requests will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all requests for word variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

BOOL YourGetWordFunc( unsigned char uchWhich, unsigned int *puiValue );
```

```
although your function can of course have any name.

In your callback function, uchWhich lets your function know which variable
is being requested.  Typically you only need to know this if you are
using the same function to access related variables.

puiValue points to the location where you should store the value
to be returned.

Your function should return TRUE if there is a value to transmit
back to the Amulet display controller.  If your function returns
FALSE, an ACK will be transmitted with no further data.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncPutWord, VarSetFuncGetByte, VarSetFuncGetString
END DESCRIPTION ******************************************************/

xmem void VarSetFuncGetWord( unsigned char uchWhich, pfGetWord pf )
{
      apfGetWord[ uchWhich ] = pf;
}

/* START FUNCTION DESCRIPTION *****************************************
VarSetFuncGetString                                                  <VarPool.c>

SYNTAX: void VarSetFuncGetString( unsigned char uchWhich, pfGetString pf );

DESCRIPTION:
When an Amulet GET_STRING request cannot be retrieved from a static
variable, but must instead be computed dynamically, use this
function to register your dynamic computation function in the
variable pool interface.

A typical application is a date/time panel, where it is not practical
to constantly update variables in memory with the current time.
Instead, register a function to read the clock and return the current
values of interest only when the Amulet display asks for them.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being requested.  In the date/time example, you could register
the same function for two different word variables, and then based
on which value is requested, your function can supply the year/month/day
string or the hour/minute string.

PARAMETER 1: uchWhich - determines which string variable
requests will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all requests for string variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

psz YourGetStringFunc( unsigned char uchWhich );

although your function can of course have any name.  Your program
```

```
should include a type definition of the form:

typedef unsigned char * psz;

In your callback function, uchWhich lets your function know which variable
is being requested.  Typically you only need to know this if you are
using the same function to access related variables.

Your function should return a pointer to a string if there is a value to
transmit back to the Amulet display controller.  Your string cannot be
stored in an automatic variable, or its value may be indeterminate when
your function returns control to its caller.  Use a static variable to
hold your string.

If your function returns NULL, an ACK will be transmitted
with no further data.

The Amulet has a limit of 255 characters per string variable.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncPutString, VarSetFuncGetByte, VarSetFuncGetWord
END DESCRIPTION ******************************************************/

xmem void VarSetFuncGetString( unsigned char uchWhich, pfGetString pf )
{
      apfGetString[ uchWhich ] = pf;
}

/* START FUNCTION DESCRIPTION ****************************************
VarSetFuncPutByte                                             <VarPool.c>

SYNTAX: void VarSetFuncPutByte( unsigned char uchWhich, pfPutByte pf );

DESCRIPTION:
When an Amulet PUT_BYTE request cannot be stored into a static
variable, but must instead be managed dynamically, use this
function to register your handler function in the variable pool
interface.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being updated.

PARAMETER 1: uchWhich - determines which byte variable
updates will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all updates to byte variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

void YourPutByteFunc( unsigned char uchWhich, unsigned char uchValue );

although your function can of course have any name.

In your callback function, uchWhich lets your function know which variable
```

```
is being updated.  Typically you only need to know this if you are
using the same function to update related variables.

uchValue is the value to be written to your byte variable.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncGetByte, VarSetFuncPutWord, VarSetFuncPutString
END DESCRIPTION *****************************************************/

xmem void VarSetFuncPutByte( unsigned char uchWhich, pfPutByte pf )
{
      apfPutByte[ uchWhich ] = pf;
}

/* START FUNCTION DESCRIPTION *****************************************
VarSetFuncPutWord                                                 <VarPool.c>

SYNTAX: void VarSetFuncPutWord( unsigned char uchWhich, pfPutWord pf );

DESCRIPTION:
Although the Amulet display cannot directly update a word variable,
this function is available in the interest of orthogonality.  If you are
writing modular code and you have an accessor function to read a word
variable, you may need a corresponding accessor function to notify
you of an update to that word variable.  Use the VarSetFuncPutWord
function to register your handler function in the variable pool
interface.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being updated.

PARAMETER 1: uchWhich - determines which word variable
updates will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all updates to word variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

void YourPutWordFunc( unsigned char uchWhich, unsigned int uiValue );

although your function can of course have any name.

In your callback function, uchWhich lets your function know which variable
is being updated.  Typically you only need to know this if you are
using the same function to update related variables.

uiValue is the value to be written into your word variable.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncGetWord, VarSetFuncPutByte, VarSetFuncPutString
END DESCRIPTION *****************************************************/

xmem void VarSetFuncPutWord( unsigned char uchWhich, pfPutWord pf )
```

```
{
      apfPutWord[ uchWhich ] = pf;
}

/* START FUNCTION DESCRIPTION ****************************************
VarSetFuncPutString                                          <VarPool.c>

SYNTAX: void VarSetFuncPutString( unsigned char uchWhich, pfPutString pf );

DESCRIPTION:
Although the Amulet display cannot directly update a string variable,
this function is available in the interest of orthogonality.  If you are
writing modular code and you have an accessor function to read a string
variable, you may need a corresponding accessor function to notify
you of an update to that string variable.  Use the VarSetFuncPutString
function to register your handler function in the variable pool
interface.

You can register the same function for more than one variable.  When
the function is called, the first argument tells the function which
variable is being updated.

PARAMETER 1: uchWhich - determines which string variable
updates will be routed to your function.

PARAMETER 2: pf - points to your callback function.

RETURN VALUE: None.

You need only register your function once; after registration,
all updates to string variable uchWhich will be routed to your function.

The prototype of function pf must take the form:

void YourPutStringFunc( unsigned char uchWhich, psz pszValue );

although your function can of course have any name.

In your callback function, uchWhich lets your function know which variable
is being updated.  Typically you only need to know this if you are
using the same function to update related variables.

pszValue is the pointer to the new value of your word variable.  The pointer
is guaranteed to point to a static location, so you do not normally need to
copy the contents of the string just to preserve it.  Storing the pointer is
a reasonable and safe action.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncGetString, VarSetFuncPutByte, VarSetFuncPutWord
END DESCRIPTION *****************************************************/

xmem void VarSetFuncPutString( unsigned char uchWhich, pfPutString pf )
{
      apfPutString[ uchWhich ] = pf;
}

////////////////////////////////////////////////////////////////////////
//    Next are the default accessor functions.  They are defined as root
// since they will be called regularly during program execution.
```

```
root BOOL VarGetByteDefault( unsigned char uchWhich, unsigned char *puchValue )
{
	if (( puchValue != NULL ) && ( apuchBytePtrs[ uchWhich ] != NULL ))
	{
		*puchValue = *apuchBytePtrs[ uchWhich ];
		return TRUE;
	}
	return FALSE;	// no value is defined
}

root BOOL VarGetWordDefault( unsigned char uchWhich, unsigned int *puiValue )
{
	if (( puiValue != NULL ) && ( apuiWordPtrs[ uchWhich ] != NULL ))
	{
		*puiValue = *apuiWordPtrs[ uchWhich ];
		return TRUE;
	}
	return FALSE;	// no value is defined
}

root psz VarGetStringDefault( unsigned char uchWhich )
{
	return apszStrings[ uchWhich ];
}

root void VarPutByteDefault( unsigned char uchWhich, unsigned char uchValue )
{
	if ( apuchBytePtrs[ uchWhich ] != NULL )
	{
		*apuchBytePtrs[ uchWhich ] = uchValue;
	}
}

root void VarPutWordDefault( unsigned char uchWhich, unsigned int uiValue )
{
	if ( apuiWordPtrs[ uchWhich ] != NULL )
	{
		*apuiWordPtrs[ uchWhich ] = uiValue;
	}
}

root void VarPutStringDefault( unsigned char uchWhich, psz pszValue )
{
	apszStrings[ uchWhich ] = pszValue;
}

//////////////////////////////////////////////////////////////////////
// The next batch of functions are used to get or put an element
// of data.  They "wrap" either the default functions just above
// or your own overriding functions.
// The Amulet communications code calls these.  You can
// and should call them for normal application processing too.
// As before, they are defined as root since they will be called
// frequently during normal program execution.

/* START FUNCTION DESCRIPTION ****************************************
VarGetByte                                            <VarPool.c>

SYNTAX: BOOL VarGetByte( unsigned char uchWhich, unsigned char *puchValue );

DESCRIPTION:
```

```
Call this function to get a byte from the variable pool interface.

PARAMETER 1: uchWhich - determines which variable to get

PARAMETER 2: puchValue - where to store the value of the variable

RETURN VALUE: Returns TRUE if the value was obtained.  Returns FALSE
if no value is available.  If no special handler function has been
attached to the variable pool for the specified variable, the function
always returns TRUE.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarPutByte, VarGetWord, VarGetString
END DESCRIPTION *****************************************************/

root BOOL VarGetByte( unsigned char uchWhich, unsigned char *puchValue )
{
      return (*(apfGetByte[ uchWhich ]))( uchWhich, puchValue );
}

/* START FUNCTION DESCRIPTION *****************************************
VarGetWord                                                <VarPool.c>

SYNTAX: BOOL VarGetWord( unsigned char uchWhich, unsigned int *puiValue );

DESCRIPTION:
Call this function to get a word from the variable pool interface.

PARAMETER 1: uchWhich - determines which variable to get

PARAMETER 2: puiValue - where to store the value of the variable

RETURN VALUE: Returns TRUE if the value was obtained.  Returns FALSE
if no value is available.  If no special handler function has been
attached to the variable pool for the specified variable, the function
always returns TRUE.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarPutWord, VarGetByte, VarGetString
END DESCRIPTION *****************************************************/

root BOOL VarGetWord( unsigned char uchWhich, unsigned int *puiValue )
{
      return (*(apfGetWord[ uchWhich ]))( uchWhich, puiValue );
}

/* START FUNCTION DESCRIPTION *****************************************
VarGetString                                                    <VarPool.c>

SYNTAX: psz VarGetString( unsigned char uchWhich );

DESCRIPTION:
Call this function to get a string from the variable pool interface.

PARAMETER 1: uchWhich - determines which variable to get

RETURN VALUE: Returns a pointer to the string if it exists, or NULL
if the string does not have a value.  If a string is returned, it is
guaranteed to be intransient - you need not make a copy of it just
```

```
to preserve it.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarPutString, VarGetByte, VarGetWord
END DESCRIPTION *******************************************************/

root psz VarGetString( unsigned char uchWhich )
{
      return (*(apfGetString[ uchWhich ]))( uchWhich );
}

/* START FUNCTION DESCRIPTION *****************************************
VarPutByte                                              <VarPool.c>

SYNTAX: void VarPutByte( unsigned char uchWhich, unsigned char uchValue );

DESCRIPTION:
Call this function to put a byte value into the variable pool interface.

PARAMETER 1: uchWhich - determines which variable to put

PARAMETER 2: uchValue - the new value of the variable

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarGetByte, VarPutWord, VarPutString
END DESCRIPTION *******************************************************/

root void VarPutByte( unsigned char uchWhich, unsigned char uchValue )
{
      (*(apfPutByte[ uchWhich ]))( uchWhich, uchValue );
}

/* START FUNCTION DESCRIPTION *****************************************
VarPutWord                                              <VarPool.c>

SYNTAX: void VarPutWord( unsigned char uchWhich, unsigned int uiValue );

DESCRIPTION:
Call this function to put a word into the variable pool interface.

PARAMETER 1: uchWhich - determines which variable to get

PARAMETER 2: uiValue - the new value of the variable

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarGetWord, VarPutByte, VarPutString
END DESCRIPTION *******************************************************/

root void VarPutWord( unsigned char uchWhich, unsigned int uiValue )
{
      (*(apfPutWord[ uchWhich ]))( uchWhich, uiValue );
}

/* START FUNCTION DESCRIPTION *****************************************
```

```
VarPutString                                                <VarPool.c>

SYNTAX: void VarPutString( unsigned char uchWhich, psz pszValue );

DESCRIPTION:
Call this function to put a string into the variable pool interface.
In doing so, you are guaranteeing that the memory referenced by
pszValue will be available indefinitely.  Do not store a pointer
to an automatic variable.

PARAMETER 1: uchWhich - determines which variable to get

PARAMETER 2: pszValue - pointer to the string variable

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarGetString, VarPutByte, VarPutWord
END DESCRIPTION *******************************************************/

root void VarPutString( unsigned char uchWhich, psz pszValue )
{
      (*(apfPutString[ uchWhich ]))( uchWhich, pszValue );
}

///////////////////////////////////////////////////////////////////////
// If you don't need a function to access a byte or a word,
// but you want to set the internal pointer to point at one of
// your own bytes or words, use these routines.  Even if you change
// the pointers, the default accessor routines will work fine.

/* START FUNCTION DESCRIPTION ******************************************
VarSetBytePtr                                               <VarPool.c>

SYNTAX: void VarSetBytePtr( unsigned char uchWhich, unsigned char * puchWhere );

DESCRIPTION:
Call this function to set the address of a byte variable.  For
example, if you have an application screen that displays the
fields of a database record, you can set the pointers for the
appropriate variables to point to fields of a structure of your record
instead of copying data back and forth between your record
and the built-in variable pool.

PARAMETER 1: uchWhich - which variable to set the pointer for

PARAMETER 2: puchWhere - where in memory to point at

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetWordPtr
END DESCRIPTION *******************************************************/

void VarSetBytePtr( unsigned char uchWhich, unsigned char * puchWhere )
{
      apuchBytePtrs[ uchWhich ] = puchWhere;
}
```

```
/* START FUNCTION DESCRIPTION ******************************************
VarSetWordPtr                                                <VarPool.c>

SYNTAX: void VarSetWordPtr( unsigned char uchWhich, unsigned int * puiWhere );

DESCRIPTION:
Call this function to set the address of a word variable.  For
example, if you have an application screen that displays the
fields of a database record, you can set the pointers for the
appropriate variables to point to fields of a structure of your record
instead of copying data back and forth between your record
and the built-in variable pool.

PARAMETER 1: uchWhich - which variable to set the pointer for

PARAMETER 2: puiWhere - where in memory to point at

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetBytePtr
END DESCRIPTION *******************************************************/

void VarSetWordPtr( unsigned char uchWhich, unsigned int * puiWhere )
{
      apuiWordPtrs[ uchWhich ] = puiWhere;
}

///////////////////////////////////////////////////////////////////////
// Next are the RPC related functions

root void VarDefaultFuncRPC( unsigned char uchWhichRPC )
{
      printf( "unexpected call to RPC 0x%02X\n", uchWhichRPC );
}

/* START FUNCTION DESCRIPTION ******************************************
VarSetFuncRPC                                                <VarPool.c>

SYNTAX: void VarSetFuncRPC( unsigned char uchWhich, pfRPC pf );

DESCRIPTION:
Call this function to set the address of a remote procedure call.
When the Amulet display controller initiates an RPC for a given
function number, if your function is registered to process that
function number, your function will be called.

You can register the same function for more than one RPC request.
This may be desirable if you have several different nearly identical
operations to perform.  For example, when programming a soft
keyboard, you can use a range of RPC values that all call the same
function, where the RPC value is the ASCII value of the character
that was tapped.  You would then need only a single handler instead
of separate handlers for each possible key.

If your Amulet uHTML code initiates an RPC request for which no
function handler has been registered, the default behavior is to
print a message on the debug stdio stream indicating which RPC
was requested.
```

```
PARAMETER 1: uchWhich - which variable to set the pointer for

PARAMETER 2: pf - pointer to your handler function.  The function
must take the form:

void MyRpcFunction( unsigned char uchWhich );

although of course you may use any name in place of MyRpcFunc.
The argument allows your function to determine which RPC request
initiated the call; normally this is useful only if you register
the same function for more than one RPC request.  You can override
this behavior by calling

      VarSetFuncRPC( uchWhich, NULL );

which informs the variable pool interface that no handler, including
the default handler, should process the RPC request.

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarCallRPC
END DESCRIPTION ******************************************************/

xmem void VarSetFuncRPC( unsigned char uchWhichRPC, pfRPC pf )
{
      apfRPCs[ uchWhichRPC ] = pf;
}

/* START FUNCTION DESCRIPTION *****************************************
VarCallRPC                                              <VarPool.c>

SYNTAX: void VarCallRPC( unsigned char uchWhich );

DESCRIPTION:
Call this function to perform a remote procedure call.  Normally
the Amulet communications code would call this function in response
to an incoming RPC request, but your application can also call it
directly for other reasons if needed.

PARAMETER 1: uchWhich - which variable to set the pointer for

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarSetFuncRPC
END DESCRIPTION ******************************************************/

root void VarCallRPC( unsigned char uchWhichRPC )
{
      if ( apfRPCs[ uchWhichRPC ] != NULL )
      {
            (*(apfRPCs[ uchWhichRPC ]))( uchWhichRPC );
      }
}

///////////////////////////////////////////////////////////////////////
// The last function is the "constructor" for the variable pool.
```

```
/* START FUNCTION DESCRIPTION ******************************************
VarPoolInitialize                                                <VarPool.c>

SYNTAX: void VarPoolInitialize();

DESCRIPTION:
Call this function exactly once at program startup to initialize
the variable pool interface.

You should call this function before initiating communications with
the Amulet display, as well as before any other requests to the
variable pool.

RETURN VALUE: None.

KEY WORDS: Amulet variable pool interface

SEE ALSO: VarGetByte, VarGetWord, VarGetString
END DESCRIPTION *******************************************************/

xmem void VarPoolInitialize()
{
	int i;
	for ( i = 0; i < 256; i++ )
	{
		apuchBytePtrs[ i ] = &auchBytes[ i ];
		apuiWordPtrs[ i ] = &auiWords[ i ];
		apszStrings[ i ] = NULL;
		apfRPCs[ i ] = VarDefaultFuncRPC;

		apfGetByte[ i ] = &VarGetByteDefault;
		apfGetWord[ i ] = &VarGetWordDefault;
		apfGetString[ i ] = (pfGetString)&VarGetStringDefault;
		apfPutByte[ i ] = &VarPutByteDefault;
		apfPutWord[ i ] = &VarPutWordDefault;
		apfPutString[ i ] = &VarPutStringDefault;
	}
}

// end of varpool.c
```