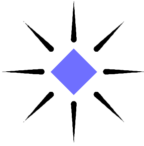

Amulet Technologies

GUI Engines for Embedded Systems

THE AMULET PROTOCOL FOR THE MOTOROLA MC68HC11

INTRO

When choosing a technology to incorporate into a project, the robustness of that technology is highly valuable. The Amulet EasyGUI Browser Chip (and of course the Starter Kit which includes the chip) has the robustness that any disruptive technology demands (click here for more about disruptive technology). It can be interfaced to any microprocessor in any programming language that the user feels most comfortable with. This experiment illustrates the porting process of the Amulet serial protocol on the Motorola MC68HC11E0 (housed in an Adapt11C75DX proto board), specifically dealing with the C and assembly programming languages. Since this experiment is not dealing with any peripheral I/O devices, this document only touches on the workings of the protocol in each language, i.e. we do not have to worry about interfacing the device to the micro and getting information it. Keep in mind that if the desired application demanded it, an even more complex routine for handling the serial information such as a Real Time Operating System (RTOS) could be implemented.

NOTES: Both of these programs are not stand-alone; the NOICE debugger (http://www.noicedebugger.com) had to be ported over for use as the monitor in the HC11, allowing for the NOICE app to be used for program loading, debugging, and running. Changes must be made to each of the protocol examples for the programs to run on reset. Resultantly, this document deals solely with the workings of the protocol, not the loading and running of the protocol from reset or from the debugger.

Also, the Adapt11C75DX board that was used in this experiment is no longer available for purchase (from http://www.technologicalarts.com/) due to the fact that the Xicor X68C75 micro-peripheral chip was discontinued in February 2000. The board has been replaced with the Adapt11C24DX, which is similar to the board used in this experiment but has differences in the mapping of the ports and also in their behavior. The 68HC24 port replacement chip used in the Adapt11C24DX emulates the 68HC11 ports B and C exactly. On the 68HC11, port B is an “output only” port, and port C is bit-wise selectable as input or output via the DDRC register. This differs significantly from the X68C75 that is used on the Adapt11C75DX board. If you are planning to use these ports you’ll have to make a number of changes to the protocol examples below (note that the examples below do not use these ports, so no changes should be necessary).

AMULET PROTOCOL (taken from http://www.amulettechnologies.com/)

The communications protocol is half-duplex, with the Easy GUI client acting as master. The Amulet GUI can send five different types of messages:

- A "Get" byte variable request (**Amulet:UART.byteValue()**)
- A "Get" word variable request (word = 2 bytes) (**Amulet:UART.wordValue()**)
- A "Get" string variable request (**Amulet:UART.stringValue()**)
- An "Invoke" Remote Procedure Call (RPC) (**Amulet:UART.invokeRPC()**)
- A "Set" byte variable command. (**Amulet:UART.setByteValue()**)

If the message is valid, the Stamp should either return the requested data (if a "Get" request) or acknowledge the message (if an "Invoke" or "Set" command). If the message is not valid, the server should respond with an acknowledge (0x10).

Table 1 in Appendix A defines the five types of messages that can be sent between the GUI and the Stamp. The valid range of variables and Remote Procedure Calls is 0-0xFF. The valid range for variable values returned from the Stamp (in response to the "Get" variable request) is also 0-0xFF. Since this is an ASCII protocol, it takes two bytes to send one-byte variables and RPCs. For example, the variable 0x1A would be transmitted as 0x31, 0x41, where 0x31 is the ASCII representation of the high nibble"1" and 0x41 is the ASCII representation of the low nibble "A".

NOTE: The Stamp side must respond to every valid GUI command. **It is always acceptable to respond with an acknowledgment.** When commands are not responded to within 100ms, a time-out will occur, which then clears out all pending requests. This can result in unexpected behavior.

Synchronization--As master, the GUI initiates all communications by sending a message to the Stamp. All valid messages from the GUI to the Stamp start with one of five command bytes: [0x11], [0x12], [0x13], [0x14] or [0x15] -- these are considered the Client Start Of Message (CSOM) characters, with the GUI being the client. **NOTE**: These five CSOM bytes ALWAYS signify the start of a message and they are not allowed in the body of any message. The only valid characters in the body of a message are: ASCII 0-9 (0x31-0x39), and A-F (0x41-0x46), except in the body of the "Get string" response, where all ASCII characters from '0' -' ~' (0x30-0x7e) are valid. If the Stamp receives any character other than those specified, the message should be considered errant, and the Stamp should start over hunting for a new CSOM character.

All Stamp responses must start with the counterpart of the CSOM character that began the message that is being responded to. The valid Server (Stamp) Start Of Message (SSOM) bytes are: [0x21], [0x22], [0x23], [0x24] or [0x25]. The body of the response message starts with the counterpart SSOM and is then followed by any optional response data (in ASCII format).

As noted earlier, after receiving the last byte of a valid message from the GUI, the Stamp then has 100ms to respond to the message before the GUI times out. After 100ms, if there is no response, the GUI will consider the message dropped, flush its buffer, and then be ready to transmit any new messages. The GUI will NOT repeat a message after a TIME OUT.

PROTOCOL IN C

The example C protocol for this experiment implements a polling type approach to gathering data off of the serial line; however, this code could have just as easily been written using interrupts. The key is that the Amulet protocol, no matter what type coding style or language is used, is easy to implement. The layout and flow of this example C protocol is quite simple to follow.

As you can see below the main loop simply calls one function, serIn(), which checks to see if anything has been received into the serial buffer. If something has been received, parseSerial() is called, otherwise the functions returns and will keep being called until a byte is received.

```
int main()
{
	while(1)
	{
		serIn(&buffer);
	}
	return(0);
}

void serIn(RingBuf *buf)
{
	if ((SCSR & RDRF) != 0)
	{
		tail = buf->tail;
		buf->serData[tail++] = SCDR;
		buf->tail = (tail & RB_SIZE_MASK);
		parseSerial();
	}

}
```

The parseSerial() function, called above and implemented below, is a simple state machine. It first checks to see if the first byte (the CSOM) is valid and proceeds to set the serverResp variable to the corresponding SSOM value. Each time a byte is received, a state variable is incremented to keep track of how many of the client bytes have been sent and stored. We use this state variable to determine when and if the correct number of bytes for each function has been received, and once all the bytes for that specific request have been received, we call the functions. However, the setByte() client request takes five bytes as compared to the three of the other function types; therefore, two more bytes must be collected and only then can the setByte handler be called.

```
void parseSerial(void)
{
	static char caseType;
	newByte = byteFromBuf(&buffer);

	if((newByte >= 0x11) && (newByte <= 0x15))
	{
		serverResp = respMake(newByte);
		caseType = serverResp;
	}
	else if(state == 0)
		caseType = 0x00;

	if((state==0) && ((caseType  >= 0x21) && (caseType <= 0x25)))
	{
		state++;
	}
	else if(state == 1)
	{
		hiNib = newByte;
		state++;
	}
	else if(state == 2)
	{
		loNib = newByte;

		if(caseType == 0x23)
		{
			state++;
		}
		else
		{
			state = 0;
		}

		switch(caseType)
		{
			case 0x21:
				getByte();
				break;
			case 0x22:
				getString();
				break;
			case 0x24:
				invoke();
				break;
			case 0x25:
				getWord();
				break;
		}
	}
	else if((state == 3) && (caseType == 0x23))
	{
		setValHi = newByte;
		state++;
	}
```

```
	else if(state == 4)
	{
		setValLo = newByte;
		state = 0;
		setByte();
	}
}
```

As we’ve seen, the handler that parseSerial() calls is dependent on which type of client request is being asked for. These handlers, each seen below, echo back the data that was received from the client, in addition to using that data to either turn an LED on and off or index into an array of bytes, strings, or words. These handlers are the routines that would need to include an I/O device interfacing subroutine if one were being used, but since we aren’t using one, we are simply using hard coded arrays to access information. Once the data has been sent back to the “client,” our code returns to the main loop and waits for another byte to be received.

```
void getByte(void)
{
	char index, byteValue, valueHiNib, valueLoNib;

	index = ascii2hex(hiNib) << 4;
	index |= ascii2hex(loNib);

	byteValue = byteData[index];

	valueHiNib = hex2ascii(byteValue >> 4);
	valueLoNib = hex2ascii(byteValue & 0x0f);

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
	putchar(valueHiNib);
	putchar(valueLoNib);
}

void getString(void)
{
	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);

	if(loNib == 0x30)
	{
		putsting(string1);
	}
	if(loNib == 0x31)
	{
		putstring(string2);
	}
	if(loNib == 0x32)
	{
		putstring(string3);
	}
```

```
	if(loNib == 0x33)
	{
		putstring(string4);
	}
}

void setByte(void)
{
	char index, hexVal;

	index = ascii2hex(hiNib) << 4;
	index |= ascii2hex(loNib);

	hexVal = ascii2hex(setValHi) << 4;
	hexVal |= ascii2hex(setValLo);

	byteData[index] = hexVal;

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
	putchar(setValHi);
	putchar(setValLo);
}

void invoke(void)
{
	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);

	if(loNib == 0x30)
		LED_ON();
	if(loNib == 0x31)
		LED_OFF();

}

void getWord(void)
{
	char index, valMSBhinib, valMSBlonib, valLSBhinib, valLSBlonib;
	unsigned int wordValue;

	index = ascii2hex(hiNib) << 4;
	index |= ascii2hex(loNib);

	wordValue = wordData[index];

	valMSBhinib = hex2ascii((char)((wordValue >> 12) & 0x0f));
	valMSBlonib = hex2ascii((char)((wordValue >> 8) & 0x0f));
	valLSBhinib = hex2ascii((char)((wordValue >> 4) & 0x0f));
	valLSBlonib = hex2ascii((char)(wordValue & 0x0f));

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
```

```
        putchar(valMSBhinib);
        putchar(valMSBlonib);
        putchar(valLSBhinib);
        putchar(valLSBlonib);
}
```

To view the code in its entirety, go to Appendix B.

PROTOCOL IN ASSEMBLY

The assembly version of the example serial protocol also implements a polling type approach to gathering data off of the serial line. Once again, if wanted, the serial communications protocol could have been written with interrupts if desired.

The assembly version's flow is also quite simply and actually handles the bytes coming in roughly the same way that the C version did. By this we mean, the assembly code waits for the first byte to be received as seen in the getch function below, and then uses that byte to determine which handler is going to be used. Note that in the trying routine below, we must use a branch to a jump that jumps to the handlers due to the fact that we need a conditional branch to a subroutine and this is the only way to accomplish this is in the HC11 instruction set.

```
getch:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getch
        LDAA        SCDR
        STAA        SERVERRESP
        RTS

trying:
        JSR         getch
        CMPA        #GETBCOMMAND
        BEQ         bytejump
        CMPA        #STRINGCOMMAND
        BEQ         stringjump
        CMPA        #SETCOMMAND
        BEQ         setjump
        CMPA        #INVOKECOMMAND
        BEQ         invokejump
        CMPA        #GETWCOMMAND
        BEQ         wordjump
        JMP         trying

bytejump:
        JSR         restbyte
        JMP         trying

stringjump:
        JSR         reststring
        JMP         trying
```

```
setjump:
        JSR         restset
        JMP         trying

invokejump:
        JSR         restinvoke
        JMP         trying

wordjump:
        JSR         restword
        JMP         trying
```

After receiving the first byte of the client request, the code now jumps to the specific handler determined by the CSOM. Each handler gathers the rest of the needed bytes (whose implementation are the same as the getch command but with a different variables being populated, these functions can be seen here) and like the C version of the protocol, indexes into hard coded arrays. Once the handler has finished grabbing values from the arrays or turned the LED on or off, the code jumps back to the main loop and begins waiting for another byte to be received. Each individual handler can be seen below. To see the implementation of the subroutines in each of the handlers, click on the links (these links will take you to the to the full version of the code in Appendix C):

```
************************************
*getByte Handler                   *
************************************
restbyte:
        JSR         getHiNib
        JSR         getLoNib
        LDAA        HINIB
        JSR         ascii2hex
        STAA        NIB1
        LDAA        LONIB
        JSR         ascii2hex
        STAA        NIB2
        JSR         two2one
        LDAA        TEMP
        STAA        INDEX
        JSR         putbyte
        JSR         putHiNib
        JSR         putLoNib
        JSR         putval
        RTS

************************************
*getString Handler                 *
************************************
reststring:
        JSR         getHiNib
        JSR         getLoNib
        LDAA        HINIB
        JSR         ascii2hex
        STAA        NIB1
        LDAA        LONIB
        JSR         ascii2hex
```

```
        STAA    NIB2
        JSR     two2one
        LDAA    TEMP
        LSLA
        STAA    INDEX
        JSR     putstring
        JSR     putHiNib
        JSR     putLoNib
        JSR     putstr
        RTS

************************************
*setByte Handler                   *
************************************
restset:
        JSR     getHiNib
        JSR     getLoNib
        JSR     getValHi
        JSR     getValLo
        LDAA    HINIB
        STAA    NIB1
        LDAA    LONIB
        STAA    NIB2
        JSR     two2one
        LDAA    TEMP
        STAA    INDEX
        LDAA    SETVALHI
        STAA    NIB1
        LDAA    SETVALLO
        STAA    NIB2
        JSR     two2one
        JSR     setbytevalue
        JSR     putsetb
        JSR     putHiNib
        JSR     putLoNib
        JSR     putValHi
        JSR     putValLo
        RTS


*************************************
*Invoke handler                     *
*************************************
restinvoke:
        JSR     getHiNib
        JSR     getLoNib
        LDAA    LONIB
        JSR     ascii2hex
        CMPA    #ZERO
        BEQ     ledoff
        CMPA    #ONE
        BEQ     ledon
ledoff:
        LDAB    #ZERO
        STAB    PORTA
        JMP     output
```

```
ledon:
        LDAB        #LED_ON
        STAB        PORTA
output:
        JSR         putinvoke
        JSR         putHiNib
        JSR         putLoNib
        RTS

***********************************
*getWord Handler                              *
***********************************
restword:
        JSR         getHiNib
        JSR         getLoNib
        LDAA        HINIB
        JSR         ascii2hex
        STAA        NIB1
        LDAA        LONIB
        JSR         ascii2hex
        STAA        NIB2
        JSR         two2one
        LDAA        TEMP
        LSLA
        STAA        INDEX
        JSR         putword
        JSR         putHiNib
        JSR         putLoNib
        JSR         putvals
        RTS
```

NOTE: most of the handlers have the same basic format and the only thing that differs are some minor data handling and the type of put functions that are used. The last jump to subroutine calls before the handler returns are the routines that are most involved with handling the hard coded data (i.e. putval, putstr, putvals, in addition to setbytevalue). These are the routines that index into the arrays, pull the data out, and put the data onto the serial line. These can be seen below:

```
*********************************************************
*Puts a byte from the byte table out onto the serial line          *
*********************************************************
putval:
        LDX         #bytetable
        LDAB        INDEX
        ABX
        LDAA        0,X
        STAA        TEMP
        JSR         one2two

        LDAA        NIB1
        JSR         hex2ascii
        STAA        SCDR
putnibone:
```

```
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putnibone

        LDAA        NIB2
        JSR         hex2ascii
        STAA        SCDR
putnibtwo:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putnibtwo
        RTS

****************************************************************************
*Puts a string from the string table out onto the serial line              *
****************************************************************************
putstr:
        LDX         #stringtable
        LDAB        INDEX
        ABX
        LDX         0,X
sendstr:
        LDAA        0,X
        CMPA        #ZERO
        BEQ         putfinish
        JSR         hex2ascii
        STAA        SCDR
putthem:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putthem
        INX
        BRA         sendstr
putfinish:
        STAA        SCDR

finish:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         finish
        RTS
**********************************************************************
*Used to put 4 bytes onto serial line from the bytetable              *
**********************************************************************
putvals:
        LDX         #wordtable
        LDAB        INDEX
        ABX
        LDD         0,X
        STD         TEMP
        JSR         one2four

        LDAA        NIB1
        JSR         hex2ascii
        STAA        SCDR
putn1:
```

```
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn1

        LDAA        NIB2
        JSR         hex2ascii
        STAA        SCDR
putn2:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn2

        LDAA        NIB3
        JSR         hex2ascii
        STAA        SCDR
putn3:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn3

        LDAA        NIB4
        JSR         hex2ascii
        STAA        SCDR
putn4:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn4

        RTS

***********************************************************************
*setByte function, used to set a desired value into bytetable         *
***********************************************************************
setbytevalue:
        LDX         #bytetable
        LDAB        INDEX
        ABX
        LDAA        TEMP
        STAA        0,X
        RTS
```

COMPARISONS AND CONCLUSIONS

Although the two different versions of the code were made to handle the data in basically the same ways, there do exist some differences that the user should be aware of before coding the protocol. The size of the .S19 files for each of the C and assembly protocols varied greatly. The assembly programmer has total control over how tight the assembly code, while the C programmer must depend on how much his compiler can optimize the code. In this experiment the assembly code (which is not tight at all) was 3k, while the ICC11 compiler could only get the .S19 to a size of 10k. This size differential did not produce any noticeable speed differences between the two codes, as both versions of the protocol were able to handle all client requests without any problems.

The purpose of this investigation was to show that the Amulet EasyGUI technology can be hooked up to any micro and in any language, be made to communicate. The Amulet technology is easy enough to work with that the final choice in choosing a micro and programming language comes down to solely what the application specs are and what the programmer feels most comfortable with.

APPENDIX A

Message	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	······· Byte N
Client Get Byte Variable	0x11	Variable Hi Nibble	Variable Lo Nibble	None	None	None	None	None
Server Response	0x21	Variable Hi Nibble	Variable Lo Nibble	Value Hi Nibble	Value Lo Nibble	None	None	None
Client Get String Variable	0x12	Variable Hi Nibble	Variable Lo Nibble	None	None	None	None	None
Server Response	0x22	Variable Hi Nibble	Variable Lo Nibble	ASCII char	ASCII char	ASCII char	ASCII char	······· 0x00
Client Set Byte Variable	0x13	Variable Hi Nibble	Variable Lo Nibble	Value Hi Nibble	Value Lo Nibble	None	None	None
Server Response	0x23	Variable Hi Nibble	Variable Lo Nibble	Value Hi Nibble	Value Lo Nibble	None	None	None
Client Invoke Remote Procedure Call (RPC)	0x14	RPC Hi Nibble	RPC Lo Nibble	None	None	None	None	None
Server Response	0x24	RPC Hi Nibble	RPC Lo Nibble	None	None	None	None	None
Client Get Word Variable	0x15	Variable Hi Nibble	Variable Lo Nibble	None	None	None	None	None
				MS Byte		LS Byte		
Server Response	0x25	Variable Hi Nibble	Variable Lo Nibble	Value Hi Hi Nibble	Value Hi Lo Nibble	Value Lo Hi Nibble	Value Lo Lo Nibble	None

Table 1. Five types of messages can be sent between the client and the server

APPENDIX B

```
/***********************************************************
*Author: Jacob Horn                                          *
*Target: Motorola 68HC11E0FN on an Adapt11C75DX board        *
*                                                            *
*This code is meant to demonstrate the workings of the Amulet protocol    *
***********************************************************/

#define  _SCI
#include <hc11.h>
#include <stdio.h>
#include <string.h>

#define RB_BUF_SIZE     0x08
#define RB_SIZE_MASK (RB_BUF_SIZE-1)
#define NULL     0

#define LED_ON()          PORTA = 0x40
#define LED_OFF()   PORTA = 0x00

typedef struct _ringbuf
{
            char head;
            char tail;
            char serData[RB_BUF_SIZE];
} RingBuf;

#define rbInit(rb) {rb.head = 0 ; rb.tail = 0 ;}

/*********************************
*GLOBAL VARIABLE ALLOCATIONS *
*********************************/

//RingBuffer to be used as "serial buffer"
RingBuf buffer;

char serverResp;                 //first byte of server response back to client
char hiNib;                      //2nd byte of server response to client
char loNib;                      //3rd byte of server response to client
char setValHi;                   //used in setByte routine to hold high nibble value of variable to be set
char setValLo;                   //contains the low nibble value of the variable being set
char state = 0;                  //state variable to keep track of how many bytes have been recieved in
                                 //serIn()
char tail;                       //variable to contain place in buffer of the last recieved byte
char head;                       //variable containing last read byte in the buffer
char newByte;                    //variable to contain byte taken out of the buffer
```

```
//Hard coded arrays used for data in getByte and getWord
char byteData[10] = {48,49,50,51,52,53,54,55,56,57};
unsigned int wordData[10] = {256,257,258,259,260,261,262,263,264,265};
char string1[7] = {'S','T','R','I','N','G',0};
char string2[6] = {'T','R','I','N','G',0};
char string3[5] = {'R','I','N','G',0};
char string4[4] = {'I','N','G',0};

/*************************
*FUNCTION PROTOTYPES   *
*************************/
void serIn(RingBuf *buf);
void parseSerial(void);
char hex2ascii(char hex);
char ascii2hex(char ascii);
void getByte(void);
void getString(void);
void setByte(void);
void invoke(void);
void getWord(void);
char byteFromBuf(RingBuf *buf);
char respMake(char byte);
void putstring(char *str);

/***********************************************************
*MAIN ROUTINE - initializes RingBuffer and sets the baud to 9600, then   *
*stays in an infinite loop polling the serial line to see if anything    *
*has been recieved, and then handling the byte received.                 *
*NOTE:  the serIn() funtion simply checks the serial line to see if      *
*       anything is there, if not it returns.                            *
***********************************************************/
int main()
{
	rbInit(buffer);
	setbaud(BAUD9600);

	while(1)
	{
		serIn(&buffer);
	}

	return(0);
}

/******************************************************
*Checks to see if anything is waiting on the serial line to be handled,   *
*if so, it puts at the end of the Ringbuffer, otherwise it does nothing    *
*and returns                                                               *
******************************************************/
void serIn(RingBuf *buf)
{
	if ((SCSR & RDRF) != 0)
	{
		tail = buf->tail;
		buf->serData[tail++] = SCDR;
		buf->tail = (tail & RB_SIZE_MASK);
```

```
		parseSerial();
	}

}

/*******************************************************
*This function acts as the byte handler to the bytes that serIn() puts	*
*in the Ringbuffer.  Checks to see if valid request type has been	*
*received and then sets server response value.  Using a standard state	*
*machine, the program proceeds to set variables to hold the values of	*
*the next byte received on the serial lines and when the correct number	*
*of bytes have been received (3 bytes for all request types except	*
*setByte which needs 5 bytes) later calls functions that put the	*
*variables back out on the serial line for output	*
*******************************************************/
void parseSerial(void)
{
	static char caseType;

	newByte = byteFromBuf(&buffer);

	if((newByte >= 0x11) && (newByte <= 0x15))
	{
		serverResp = respMake(newByte);
		caseType = serverResp;

	}
	else if(state == 0)
		caseType = 0x00;


	if((state==0) && ((caseType  >= 0x21) && (caseType <= 0x25)))
	{
		state++;
	}
	else if(state == 1)
	{
		hiNib = newByte;
		state++;
	}
	else if(state == 2)
	{
		loNib = newByte;

		if(caseType == 0x23)
		{
			state++;
		}
		else
		{
			state = 0;
		}

		switch(caseType)
		{
			case 0x21:
```

```
                    getByte();
                    break;
                case 0x22:
                    getString();
                    break;
                case 0x24:
                    invoke();
                    break;
                case 0x25:
                    getWord();
                    break;
            }
        }
        else if((state == 3) && (caseType == 0x23))
        {
            setValHi = newByte;
            state++;
        }
        else if(state == 4)
        {
            setValLo = newByte;
            state = 0;
            setByte();
        }
}

/********************************
*Hex to ascii conversion routine        *
*********************************/
char hex2ascii(char hex)
{
        return ((hex < 0x0A) ? (hex + '0') : (hex + ('A' - 0x0A)));
}

/********************************
*Ascii to hex conversion routine        *
*********************************/
char ascii2hex(char ascii)
{
        return((ascii <= '9') ? (ascii - '0') : (ascii - ('A' - 0x0A)));
}

/***********************************************************************
*Handler for a getByte function request                                     *
*Format of request string = 0x11xx, where xx = variable being requested     *
*Returns 0x21xxNN, where NN equals the HEX data of the variable xx          *
***********************************************************************/
void getByte(void)
{
        char index, byteValue, valueHiNib, valueLoNib;

        index = ascii2hex(hiNib) << 4;
        index |= ascii2hex(loNib);

        byteValue = byteData[index];
```

```
	valueHiNib = hex2ascii(byteValue >> 4);
	valueLoNib = hex2ascii(byteValue & 0x0f);

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
	putchar(valueHiNib);
	putchar(valueLoNib);
}

/*************************************************************************
*Handler for a getString function request                                    *
*Format of request string = 0x12xx, where xx = index of string variable      *
*Returns 0x22xxString+Null to client                                         *
*Returns requested data back to the screen                                   *
*                                                                            *
*Uses putchar from ICC C library to put individual characters onto serial line *
*************************************************************************/
void getString(void)
{
	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);

	if(loNib == 0x30)
	{
		putstring(string1);
	}
	if(loNib == 0x31)
	{
		putstring(string2);
	}
	if(loNib == 0x32)
	{
		putstring(string3);
	}
	if(loNib == 0x33)
	{
		putstring(string4);
	}
}

/*******************************************************************
*Handler for a setByte function request                              *
*Format of request string = 0x13xxNN, where xx = variable to be set  *
*and NN = HEX data                                                   *
*Returns 0x23xxNN to client                                          *
*******************************************************************/
void setByte(void)
{
	char index, hexVal;

	index = ascii2hex(hiNib) << 4;
	index |= ascii2hex(loNib);

	hexVal = ascii2hex(setValHi) << 4;
```

```
	hexVal |= ascii2hex(setValLo);

	byteData[index] = hexVal;

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
	putchar(setValHi);
	putchar(setValLo);
}

/************************************************************
*Handler for a invoke function request                        *
*Format of request string = 0x14xx, where xx = variable of function  *
*being requested                                              *
*Returns 0x24xx to client                                     *
*Invoke function turns on/off board's LED                    *
************************************************************/
void invoke(void)
{
	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);

	if(loNib == 0x30)
		LED_ON();
	if(loNib == 0x31)
		LED_OFF();

}

/************************************************************************
*Handler for a getWord function request                                    *
* Format of request string = 0x15xx, where xx = variable being requested    *
*Returns 0x25xxPPNN, where PP = high byte of variable xx, and NN = low byte of  *
*variable xx                                                               *
************************************************************************/
void getWord(void)
{
	char index, valMSBhinib, valMSBlonib, valLSBhinib, valLSBlonib;
	unsigned int wordValue;

	index = ascii2hex(hiNib) << 4;
	index |= ascii2hex(loNib);

	wordValue = wordData[index];

	valMSBhinib = hex2ascii((char)((wordValue >> 12) & 0x0f));
	valMSBlonib = hex2ascii((char)((wordValue >> 8) & 0x0f));
	valLSBhinib = hex2ascii((char)((wordValue >> 4) & 0x0f));
	valLSBlonib = hex2ascii((char)(wordValue & 0x0f));

	putchar(serverResp);
	putchar(hiNib);
	putchar(loNib);
	putchar(valMSBhinib);
```

```
		putchar(valMSBlonib);
		putchar(valLSBhinib);
		putchar(valLSBlonib);
}

/*********************************************
*Function to take a byte out of the buffer			*
*********************************************/
char byteFromBuf(RingBuf *buf)
{
		char byte;
		head = buf->head;
		byte = buf->serData[head];
		buf->head = (head + 1) & RB_SIZE_MASK;

		return byte;
}

/*******************************************************************************
*Function to assign a serverResp value based on the byte taken out of the buffer		*
*******************************************************************************/
char respMake(char byte)
{
		char resp;

		if(byte == 0x11)
		{
				resp = 0X21;
		}
		else if(byte == 0x12)
		{
				resp = 0X22;
		}
		else if(byte == 0x13)
		{
				resp = 0X23;
		}
		else if(byte == 0x14)
		{
				resp = 0X24;
		}
		else if(byte == 0x15)
		{
				resp = 0X25;
		}

		return resp;
}

/***************************************
*Function to put a string onto the serial line		*
***************************************/
void putstring(char *str)
{
		char iloop;
		char index = 0;
```

```
        char value;

        for(iloop=0; iloop <= strlen(str); iloop++)
        {
                value = str[index];
                putchar(value);
                index++;
        }
}
```

APPENDIX C

```
*****************************************************************
*Author: Jacob Horn                                                *
*Company: Amulet Technologies                                      *
*Target: Motorola MCHC11E0 housed on an Adapt11C75DX proto board   *
*Purpose: To demonstrate the workings of the Amulet protocol,      *
*            written in HC11 assembly                              *
*****************************************************************

**********************************************
*I/O pin declarations                        *
**********************************************
PORTA       EQU     $9000
PIOC        EQU     $9002
PORTC       EQU     $9003
PORTB       EQU     $9004
PORTCL      EQU     $9005
DDRC        EQU     $9007
PORTD       EQU     $9008
DDRD        EQU     $9009
PORTE       EQU     $900A
CFORC       EQU     $900B
OC1M        EQU     $900C
OC1D        EQU     $900D
TCNT        EQU     $900E
TIC1        EQU     $9010
TIC2        EQU     $9012
TIC3        EQU     $9014
TOC1        EQU     $9016
TOC2        EQU     $9018
TOC3        EQU     $901A
TOC4        EQU     $901C
TOC5        EQU     $901E
TCTL1       EQU     $9020
TCTL2       EQU     $9021
TMSK1       EQU     $9022
TFLG1       EQU     $9023
TMSK2       EQU     $9024
TFLG2       EQU     $9025
PACTL       EQU     $9026
PACNT       EQU     $9027
SPCR        EQU     $9028
SPSR        EQU     $9029
SPDR        EQU     $902A
BAUD        EQU     $902B
SCCR1       EQU     $902C
SCCR2       EQU     $902D
SCSR        EQU     $902E
```

```
SCDR            EQU     $902F
ADCTL           EQU     $9030
ADR1            EQU     $9031
ADR2            EQU     $9032
ADR3            EQU     $9033
ADR4            EQU     $9034
OPTION          EQU     $9039
COPRST          EQU     $903A
PPROG           EQU     $903B
HPRIO           EQU     $903C
INIT            EQU     $903D
TEST1           EQU     $903E
CONFIG          EQU     $903F

*******************************************
*Client/Server interactions               *
*******************************************
GETBCOMMAND     EQU     $11
GETBRESPONSE    EQU     $21
STRINGCOMMAND   EQU     $12
STRINGRESPONSE  EQU     $22
SETCOMMAND      EQU     $13
SETRESPONSE     EQU     $23
INVOKECOMMAND   EQU     $14
INVOKERESPONSE  EQU     $24
GETWCOMMAND     EQU     $15
GETWRESPONSE    EQU     $25

********************************************
*Constants that need to be defined         *
********************************************
ZERO            EQU     $00
ONE             EQU     $01
NOTSET          EQU     $FF
BAUD9600        EQU     $30
TRENA           EQU     $0C
SERVERRESP      EQU     $50
HINIB           EQU     $51
LONIB           EQU     $52
SETVALHI        EQU     $53
SETVALLO        EQU     $54
INDEX           EQU     $55
VALUE           EQU     $56
HI              EQU     $57
LO              EQU     $58
MSBHI           EQU     $59
MSBLO           EQU     $60
LSBHI           EQU     $61
LSBLO           EQU     $62
TEMP            EQU     $63
TEMP2           EQU     $64
NIB1            EQU     $65
NIB2            EQU     $66
NIB3            EQU     $67
NIB4            EQU     $68
STRINGINDEX     EQU     $69
```

```
LOMASK          EQU     $0F
HIMASK          EQU     $F0
STACKINIT       EQU     $01FF
TDRE            EQU     $80
RDRF            EQU     $20
LED_ON          EQU     $40
ASCIIZERO       EQU     '0'
ASCIIA          EQU     'A'
ASCIININE       EQU     '9'

************************************************************
        ORG     $0200
        LDS     #STACKINIT

**********************************
*initialization of serial port       *
**********************************
        LDAA    #BAUD9600
        STAA    BAUD
        LDAA    #TRENA
        STAA    SCCR2

start:

****************************************************************************
*Main Loop - Get a character and compare to see what type of request it is  *
****************************************************************************
trying:
        JSR     getch
        CMPA    #GETBCOMMAND
        BEQ     bytejump
        CMPA    #STRINGCOMMAND
        BEQ     stringjump
        CMPA    #SETCOMMAND
        BEQ     setjump
        CMPA    #INVOKECOMMAND
        BEQ     invokejump
        CMPA    #GETWCOMMAND
        BEQ     wordjump
        JMP     trying

bytejump:
        JSR     restbyte
        JMP     trying

stringjump:
        JSR     reststring
        JMP     trying

setjump:
        JSR     restset
        JMP     trying

invokejump:
        JSR     restinvoke
        JMP     trying
```

```
wordjump:
        JSR         restword
        JMP         trying

************************************
*getByte Handler                    *
************************************
restbyte:
        JSR         getHiNib
        JSR         getLoNib
        LDAA        HINIB
        JSR         ascii2hex
        STAA        NIB1
        LDAA        LONIB
        JSR         ascii2hex
        STAA        NIB2
        JSR         two2one
        LDAA        TEMP
        STAA        INDEX
        JSR         putbyte
        JSR         putHiNib
        JSR         putLoNib
        JSR         putval
        RTS

************************************
*getString Handler                  *
************************************
reststring:
        JSR         getHiNib
        JSR         getLoNib
        LDAA        HINIB
        JSR         ascii2hex
        STAA        NIB1
        LDAA        LONIB
        JSR         ascii2hex
        STAA        NIB2
        JSR         two2one
        LDAA        TEMP
        LSLA
        STAA        INDEX
        JSR         putstring
        JSR         putHiNib
        JSR         putLoNib
        JSR         putstr
        RTS

************************************
*setByte Handler                    *
************************************
restset:
        JSR         getHiNib
        JSR         getLoNib
        JSR         getValHi
        JSR         getValLo
```

```
        LDAA    HINIB
        STAA    NIB1
        LDAA    LONIB
        STAA    NIB2
        JSR     two2one
        LDAA    TEMP
        STAA    INDEX
        LDAA    SETVALHI
        STAA    NIB1
        LDAA    SETVALLO
        STAA    NIB2
        JSR     two2one
        JSR     setbytevalue
        JSR     putsetb
        JSR     putHiNib
        JSR     putLoNib
        JSR     putValHi
        JSR     putValLo
        RTS

*************************************
*Invoke handler                      *
*************************************
restinvoke:
        JSR     getHiNib
        JSR     getLoNib
        LDAA    LONIB
        JSR     ascii2hex
        CMPA    #ZERO
        BEQ     ledoff
        CMPA    #ONE
        BEQ     ledon
ledoff:
        LDAB    #ZERO
        STAB    PORTA
        JMP     output
ledon:
        LDAB    #LED_ON
        STAB    PORTA
output:
        JSR     putinvoke
        JSR     putHiNib
        JSR     putLoNib
        RTS

************************************
*getWord Handler                    *
************************************
restword:
        JSR     getHiNib
        JSR     getLoNib
        LDAA    HINIB
        JSR     ascii2hex
        STAA    NIB1
        LDAA    LONIB
        JSR     ascii2hex
```

```
        STAA    NIB2
        JSR     two2one
        LDAA    TEMP
        LSLA
        STAA    INDEX
        JSR     putword
        JSR     putHiNib
        JSR     putLoNib
        JSR     putvals
        RTS

*****************************************************************************
*Puts the corresponding SSOM (server start of message) onto the serial line         *
*****************************************************************************
putbyte:
        LDAA    #GETBRESPONSE
        STAA    SCDR
putb:
        LDAA    SCSR
        ANDA    #TDRE
        BEQ     putb
        RTS

putstring:
        LDAA    #STRINGRESPONSE
        STAA    SCDR
puts:
        LDAA    SCSR
        ANDA    #TDRE
        BEQ     puts
        RTS

putsetb:
        LDAA    #SETRESPONSE
        STAA    SCDR
putsb:
        LDAA    SCSR
        ANDA    #TDRE
        BEQ     putsb
        RTS

putinvoke:
        LDAA    #INVOKERESPONSE
        STAA    SCDR
puti:
        LDAA    SCSR
        ANDA    #TDRE
        BEQ     puti
        RTS

putword:
        LDAA    #GETWRESPONSE
        STAA    SCDR
putw:
        LDAA    SCSR
        ANDA    #TDRE
```

```
        BEQ         putw
        RTS

*********************************************************************
*Puts the 2nd and 3rd byte (HINIB and LONIB) onto the serial line      *
*********************************************************************
putHiNib:
        LDAA        HINIB
        STAA        SCDR
puthn:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         puthn
        RTS

putLoNib:
        LDAA        LONIB
        STAA        SCDR
putln:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putln
        RTS

***************************************************************
*Puts a byte from the byte table out onto the serial line        *
***************************************************************
putval:
        LDX         #bytetable
        LDAB        INDEX
        ABX
        LDAA        0,X
        STAA        TEMP
        JSR         one2two

        LDAA        NIB1
        JSR         hex2ascii
        STAA        SCDR
putnibone:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putnibone

        LDAA        NIB2
        JSR         hex2ascii
        STAA        SCDR
putnibtwo:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putnibtwo
        RTS
```

```
******************************************************************************
*Puts a string from the string table out onto the serial line                    *
******************************************************************************
putstr:
        LDX         #stringtable
        LDAB        INDEX
        ABX
        LDX         0,X
sendstr:
        LDAA        0,X
        CMPA        #ZERO
        BEQ         putfinish
        JSR         hex2ascii
        STAA        SCDR
putthem:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putthem
        INX
        BRA         sendstr
putfinish:
        STAA        SCDR
finish:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         finish
        RTS

***********************************************************************************
*Used to put third and fourth bytes onto serial line for a setByte response         *
***********************************************************************************
putValHi:
        LDAA        SETVALHI
        STAA        SCDR
putvh:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putvh
        RTS

putValLo:
        LDAA        SETVALLO
        STAA        SCDR
putvl:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putvl
        RTS

*************************************************************************
*Used to put 4 bytes onto serial line from the wordtable                   *
*************************************************************************
putvals:
        LDX         #wordtable
        LDAB        INDEX
        ABX
```

```
        LDD         0,X
        STD         TEMP
        JSR         one2four

        LDAA        NIB1
        JSR         hex2ascii
        STAA        SCDR
putn1:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn1

        LDAA        NIB2
        JSR         hex2ascii
        STAA        SCDR
putn2:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn2

        LDAA        NIB3
        JSR         hex2ascii
        STAA        SCDR
putn3:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn3

        LDAA        NIB4
        JSR         hex2ascii
        STAA        SCDR
putn4:
        LDAA        SCSR
        ANDA        #TDRE
        BEQ         putn4

        RTS

**************************************************************************
*get functions, used to grab bytes from the serial line                  *
**************************************************************************
getch:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getch
        LDAA        SCDR
        STAA        SERVERRESP
        RTS

getHiNib:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getHiNib
        LDAA        SCDR
        STAA        HINIB
        RTS
```

```
getLoNib:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getLoNib
        LDAA        SCDR
        STAA        LONIB
        RTS

getValHi:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getValHi
        LDAA        SCDR
        STAA        SETVALHI
        RTS

getValLo:
        LDAA        SCSR
        ANDA        #RDRF
        BEQ         getValLo
        LDAA        SCDR
        STAA        SETVALLO
        RTS

************************************************************************
*setByte function, used to set a desired value in the bytetable        *
************************************************************************
setbytevalue:
        LDX         #bytetable
        LDAB        INDEX
        ABX
        LDAA        TEMP
        STAA        0,X
        RTS

********************************************************************************
*requires accumulator A to be loaded with value that is going to converted from hex to ascii    *
********************************************************************************
hex2ascii:
        CMPA        #$0A
        BHI         higher
        ADDA        #ASCIIZERO
        RTS
higher:
        LDAB        #(ASCIIA - $0A)
        ABA
        RTS

***********************************************************************************
*requires accumulator A to be loaded with value that is going to be converted  from ascii to hex    *
***********************************************************************************
ascii2hex:
        CMPA        #ASCIININE
        BHS         highersame
        SUBA        #ASCIIZERO
```

```
        RTS
highersame:
        LDAB        #(ASCIIA - $0A)
        SBA
        RTS

*****************************************************************************************
*requires TEMP to be loaded with variable that is going to be broken apart into two bytes   *
*****************************************************************************************
one2two:
        LDAA        TEMP
        LDAB        TEMP
        LSRA
        LSRA
        LSRA
        LSRA
        STAA        NIB1
        ANDB        #LOMASK
        STAB        NIB2
        RTS

*****************************************************************************************
*requires NIB1 and NIB2 to be loaded with the two bytes that are going to be made into one   *
*****************************************************************************************
two2one:
        LDAA        NIB1
        LSLA
        LSLA
        LSLA
        LSLA
        LDAB        NIB2
        ANDB        #LOMASK
        STAB        TEMP
        ORAA        TEMP
        STAA        TEMP
        RTS

*****************************************************************************************
*requires TEMP (and  TEMP2 – memory locations $63 and $64) to be loaded with byte that will be   *
*broken into four nibs                                                                          *
*****************************************************************************************
one2four:
        LDAA        TEMP
        LDAB        TEMP
        LSRA
        LSRA
        LSRA
        LSRA
        STAA        NIB1
        ANDB        #LOMASK
        STAB        NIB2

        LDAA        TEMP2
        LDAB        TEMP2
        LSRA
        LSRA
```

```
        LSRA
        LSRA
        STAA        NIB3
        ANDB        #LOMASK
        STAB        NIB4
        RTS

****************************************************************************
*look-up tables used for getByte, getString, and getWord requests                    *
****************************************************************************
bytetable:
        FCB         $0, $1, $2, $3, $4, $5, $6, $7, $8

wordtable:
        FDB         $0100, $0101, $0102, $0103, $0104, $0105,$0106, $0107, $0108

stringtable:
        FDB         string1
        FDB         string2
        FDB         string3
        FDB         string4

string1:
        FCB         'S'
        FCB         't'
        FCB         'r'
        FCB         'i'
        FCB         'n'
        FCB         'g'
        FCB         $0

string2:
        FCB         'T'
        FCB         'r'
        FCB         'i'
        FCB         'n'
        FCB         'g'
        FCB         $0

string3:
        FCB         'R'
        FCB         'i'
        FCB         'n'
        FCB         'g'
        FCB         $0

string4:
        FCB         'I'
        FCB         'n'
        FCB         'g'
        FCB         $0
```