Bootstrapping the SAB 88C166 for In-System FLASH Programming

Introduction

Some applications deal with low-level assembly programming while some require high-level host PC code. This project contains both as we discuss bootstrapping the 88C166 and programming its internal FLASH using a host PC.

Almost all microcontrollers have some internal (on-chip) RAM. The capability to run code from internal RAM distinguishes the C16x family of microcontrollers. Moreover, the bootstrap feature of the family allows code to be placed in internal RAM through the serial port upon reset. This provides a very versatile means to accomplish system-level tasks. Perhaps the best example is the use of the bootstrap feature to load and execute code to program the internal FLASH. This way, we may program the internal FLASH in a stand-alone fashion involving only the processor and no other peripherals. This is sometimes referred to as In-System Programming (ISP), implying that the internal FLASH may be programmed even after the processor is placed into a product.

We will review the basics of bootstrapping and FLASH programming of the SAB 88C166 microcontroller. Although the 88C166 microcontroller is soon to be replaced by more advanced and easier to use members of the family, the ideas and code developed in this project are still useful in illustrating the approach. Many of the elements presented here may be used for the future members of the microcontroller family. We will write low-level assembly code to be downloaded to the target processor from a PC. We also discuss Win32 code that runs on the PC host to bootstrap and program the FLASH. This code is written in a modular fashion. More specifically, the user interface and the low-level host-to-board communications tasks are separated. The latter are handled by two Dynamic Link Libraries (DLLs), provided courtesy of Rigel Corporation. The DLLs allow users to incorporate the specific low-level functions into their own programs, without the burden of writing and debugging low-level code such as Windows serial port drivers.

Finally, it should also be mentioned that bootstrapping and programming the 88C166 FLASH is by no means a new concept. There is a collection of software (see reference 1) that programs the 88C166 FLASH. Recent versions of the software have been available on the Siemens BBS and web sites. Some utilities are designed to work in systems with external RAM to hold a larger special-purpose monitor which handles all aspects of FLASH programming (see reference 5 for a free utility of this type). There are also several commercially available FLASH programmers for the C166 family.

Although our task is to eventually program the internal FLASH of the 88C166, you will notice that we spend most of our effort in constructing a robust and flexible framework with enough formalism and abstraction. This organized approach helps us to accomplish various tasks, not just programming internal FLASH. For example, we want to present a general approach to developing bootstrap code, which may be modified and used in various cases. It is such attention to a more formal structure that makes code development more of a science rather than a trial-and-error effort. The formalism and structure not only allow us to reuse our code in similar tasks, but also reduce the debugging effort and facilitate multi-programmer code development by establishing practice standards.

Bootstrapping

The SAB 88C166 microcontroller bootstraps the same way as the its ROM-less counterpart, the SAB C166. Given certain hardware conditions are met, upon a hardware reset, the processor goes into the bootstrap mode. Once in the bootstrap mode, the processor activates the asynchronous serial port S0 and follows the steps below (see reference 3):

1. Wait for a 0 byte
2. When received, send the handshake byte 0x55
3. Wait for 32 more bytes and place them in internal RAM starting at address 0FA40h
4. Once all 32 bytes are received, start execution from 0FA40h

Bootstrapping can be viewed as a procedure to serially load a 32-byte program into internal RAM and execute it. Clearly, 32 bytes is not enough to program the internal FLASH. We use the 32-byte program to load another program, which in turn may load the larger programs needed to program the internal FLASH. This multi-stage loading is typical in all C166 family bootstrapping operations.

The first 32 bytes is perhaps the most important. Here is the code that we will use.

```
#define CODESIZE 176

      org   0FA40h

; --- load code immediately following the boot script ---
      mov   r0, #0FA60h

; --- loop to load the received code ---
LOOP:
      jnb   S0RIR, W0              ; wait for byte
      movb  [r0], S0RBUF           ; store byte
      bclr  S0RIR                  ; clear receive flag
      movb  S0TBUF, [r0]           ; echo the byte
      jnb   S0TIR, $
      bclr  S0TIR

; increment pointer and check size
      cmpi1 R0, #(0FA60h + CODESIZE - 1)

      jmpr  cc_NE, LOOP            ; repeat if more bytes
      nop                          ; filler bytes

#include <sfr166.inc>              ; SFRs are defined here
```

Listing 1. The Bootstrap Code Source

The bootstrap code constructs a loop of instructions to receive a specified number of bytes from the serial port and place them starting at FA60h. The address FA60h is not arbitrary. The 32-byte code is downloaded and placed starting from FA40h. The last of the 32-byte code is thus placed in FA5Fh. Consequently, FA60h is the address of the byte (or word) immediately following the 32-byte block of bootstrap code. Provided that the bootstrap code does not branch away, the instruction at FA60h is the first instruction to be executed once the bootstrap code is finished.

Let us dissect the code. The code specifies the origin to be FA40h. Note that such address information will be stripped away while only the 32 bytes of instructions will be sent to the processor. The origin is useful, though, in developing the bootstrap code. Once assembled, the resultant list file shows the memory locations the various instructions occupy. Register R0 is used as a pointer, initialized to 0FA60h. The loop starts with label LOOP and ends with the instruction "jmpr cc_NE, LOOP". The loop waits for the receive interrupt flag of serial port 0 (S0RIR) to be set. This condition indicates that a byte has been received and is available at the S0 receive buffer. The byte just received is placed in internal RAM at the address held by the pointer R0. Next, the flag S0RIR is cleared so that it may similarly be interrogated for the next byte. The byte just received is also echoed back to the host. This allows the host to verify that all bytes sent are

correctly received by the processor. The byte is echoed back simply by copying it into the S0 transmit buffer S0TBUF. The program waits until the byte is actually transmitted. That is, the code waits for the S0 transmit interrupt request (S0TIR) flag to be set. Once set, the flag is cleared, making it ready for the next transmission. The next task is to see if all bytes are received. The macro CODESIZE is initialized to the length of the program that the loop downloads. The last byte in the program is loaded into the memory location (0FA60h + CODESIZE – 1). Note that the "cmpi1" instruction not only compares register R0 to the constant (0FA60h + CODESIZE – 1), but also increments the register R0. This updates the pointer R0 so that the bytes received are stored at consecutive locations.

The loop terminates when CODESIZE number of bytes are received. The last instruction is a "no operation" or "nop" instruction. This is needed since the bootstrap code must be exactly 32 bytes long. The initialization of register R0 and the loop collectively take 30 bytes. The last instruction brings the bootstrap code length to 32 bytes. This is where the list file, mentioned above, comes in handy. You may verify the size of the bootstrap program by inspecting the list file.

Note that the code to be downloaded should not overwrite the Special Function Registers (SFR) area immediately following the internal RAM. The C166 SFRs start at FE00h. If the bootstrap code uses Peripheral Event Controllers (PECs), the PEC pointers in the region [FDE0..FDFFh] should also be off limits to the bootstrap code. Using this scheme we have a code space of [FA60h..FDDFh] or 896 bytes. Internal RAM is also used for the registers and the stack. Fortunately, we have a total of 64 more bytes at the low end of internal RAM, namely the block [0FA00h..0FA5Fh]. Half of this ([0FA40h-FA5Fh] is where the 32-byte bootstrap code was loaded. The 32-byte bootstrap code is discardable. That is, once loaded and used, this memory is available for other use. We will use the memory below 0FA60h for the stack and the registers.

Of course, if the system has external RAM, the loop could be initialized to download bytes directly into external memory. Since external RAM is typically much larger than internal RAM, very large programs can be downloaded and run with this approach. Most commercial C166 monitors are downloaded in this manner.

If all FLASH programming routines would fit in a 896-byte block, we could have simply downloaded the routines and run them. Unfortunately, the 88C166 FLASH programming is a bit more involved. Moreover, we would like our scheme to be versatile to accommodate downloading and running routines other than just those for FLASH programming. These considerations lead to the adoption of a standard way of loading and running programs. We will call these programs "applications." This is typically the job of an operating system. So, by all practical means, we need to write an operating system for our internal-RAM-only operations. You may have noticed that in the bootstrap loop, a code size of 176 bytes is specified, much less than the 896-byte or so block available. This 176-byte code is actually a very small operating system.

A Very Small Operating System (VSOS)

Operating systems are responsible for keeping the computer hardware and software in stable and known states so that application programs may reliably be loaded and run. An operating system almost always comes with a set of operational rules, such as how memory is used, or which system resources are allocated to the operating system functions. It seems a little presumptuous to call our simple application loader and executor an operating system. After all, what is a 176-byte program compared to UNIX or OS/2? Nonetheless that is what our program does, operate the system so that other applications may be loaded and executed. It is small because we only have a few system resources to deal with, namely a few kilobytes of internal RAM, internal FLASH, a timer, and a serial port. Moreover, our program is not real-time, and runs only one application at a time.

VSOS follows a simple flow chart as shown in Figure 1. It consists of an initialization phase followed by a command loop. Four commands are recognized: Cmd_SetAppID, Cmd_GetAppID,

Cmd_LoadApp, Cmd_RunApp, which are defined to be ‘1’, ‘2’, ‘3’, and ‘4’, respectively. There are three utility functions used by the VSOS. GetByte and SendByte poll serial port S0 to receive and transmit a single byte. LoadApp is a simple loader loop that receives bytes and places them consecutively in memory. VSOS is given in Listing 2.

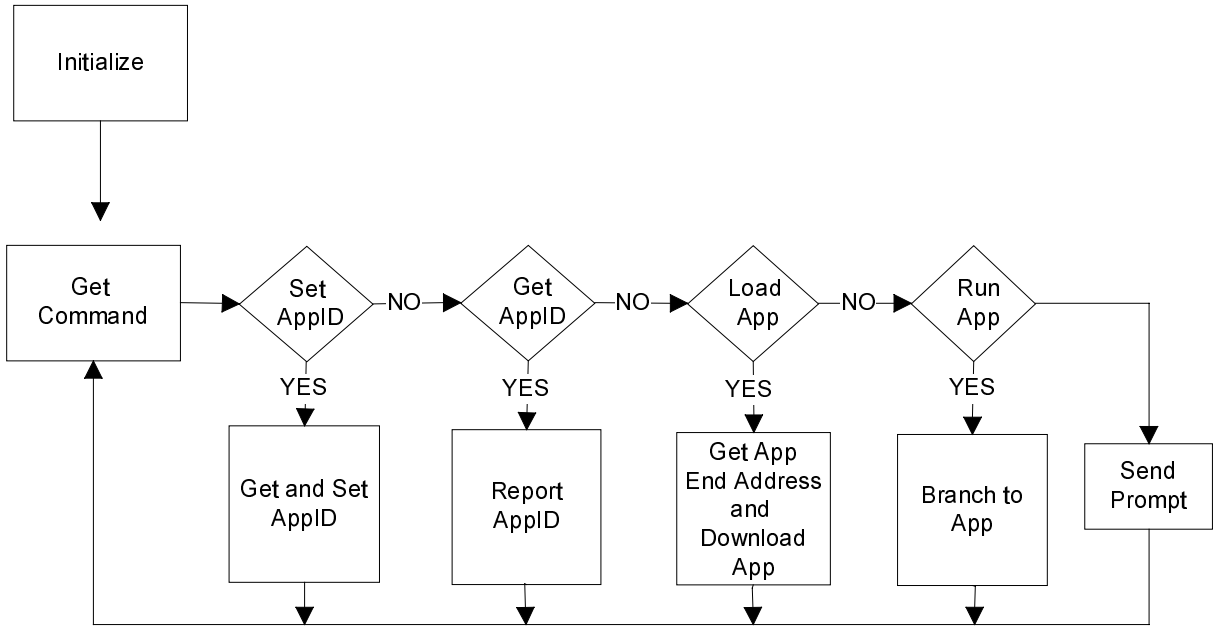

Figure 1. VSOS Logic

Initialization and Memory Use

The system initialization sets the SYSCON, DPP, CSP, S0CON, and S0BG registers. DPP0 and DPP1 registers are set to have access to the internal FLASH. In other applications, DPP registers may be set to other data pages, e.g. 0 and 1. Internal FLASH is enabled by configuring the associated fields in the SYSCON register. The Context Pointer (CP) is set to FA00h allocating the memory block [FA00h..FA1F] to the registers. Similarly, the Stack Pointer (SP) is set to FA40h, which allows a 32-byte stack located in [FA20h..FA3Fh]. The applications must limit their stack use to 32 bytes. The serial port is set to run at 57600 Baud with a 20MHz CPU frequency.

Note that the memory block [FA40h..FA5Fh], which originally held the 32-byte bootstrap code, is available for VSOS and the applications. VSOS uses two integers at FA40h and FA42h to store two integers AppID and AppEnd. AppID is an auxiliary number that may be used to identify the current application or to pass information form one application to the next. It is also useful in debugging. AppEnd is used while loading applications. It specifies first available memory location after the application. Effectively, (AppEnd-FB10h) is the size of the application. Note that AppEnd must be between FB10h and FDE0h.

The remaining 28 bytes from FA44h to FA5Fh are available to hold run-time variables of the currently loaded application.

Setting and Getting the AppID

The first two VSOS commands simply allow you to set the AppID and ask for and receive the AppID. Although our FLASH programming applications do not use the AppID, this feature was used in debugging the system. These two functions were written first and tested before the application loader was developed. Similarly, while debugging the application routines, key information was stored in AppID, which later was observed by simply using the GetAppID

command. You may also use this feature in run-time to set and get numbers to see if VSOS is still functioning properly.

Loading and Running Applications

VSOS imposes a few rules concerning how applications are loaded and run. All applications are loaded starting at FB10h. This is the first available memory location following VSOS. A loaded application is executed by a call to the address FB10h. Thus, applications must terminate with a return statement.

Loading an application starts by sending two bytes, the applications end address. This value is copied by the loader into AppEnd. The LoadApp routine restricts the applications to be contiguous blocks of code or data. Each byte received by the loader is placed into consecutive internal RAM locations. All bytes are sent in binary. Note that, if the application needs memory reserved for data, dummy bytes must be downloaded into these memory locations. The DS or DSW pseudo-ops are useful in defining such initialized data segments.

Loading applications in binary is faster than loading them in, say the Intel Hex format. It also consumes far less code in VSOS. A drawback, however, is the lack of verification, such as a cyclic redundancy check (CRC). Two considerations were noted during the development. First, each downloaded byte is echoed by the loader, so a byte-by-byte verification exists. Secondly, the loader may compute the CRC and report it at the end of the download. The latter is left for future work.

Once the application is loaded into memory, issuing the Cmd_RunApp command causes the VSOS to branch to FB10h and effectively run the application.

```
; --- constants ---
#define OS_SIZE    176          ; (FB10-FA60)
#define APPSTART 0FB10h         ; (0FA60h+OS_SIZE)

#define Cmd_SetAppID '1'
#define Cmd_GetAppID '2'
#define Cmd_LoadApp  '3
#define Cmd_RunApp   '4

; --- memory variables are stored in FA40..FA5E ---
     org   0FA40h
AppID:
     org   0FA42h
AppEnd:
; --- code is not relocatable due to the (absolute)
; --- intersegment jump ---
     org   0FA60h
; --- initialize system registers ---
     mov   SYSCON, #004Eh       ; map FLASH to segment 1
     jmps  0, next              ; initialize CSP
next:
     mov   DPP0,#4h             ; DPP0, DPP1 to access FLASH
     mov   DPP1,#5h
     mov   DPP2,#2h
     mov   DPP3,#3h
; --- initialize the stack and register space ---
     mov   CP, #0FA00h
     mov   SP, #0FA40h
     mov   S0CON, #8011h
```

```
     mov   S0BG, #10
     diswdt                    ; watch doggie sleeps
     einit                     ; end of initialization

CommandLoop:
     callr GetByte             ; get command

     cmp   r0, #Cmd_SetAppID   ; set AppID
     jmpr  cc_NE, Loop01
     callr GetByte
     mov   AppID, r0
     jmpr  cc_UC, CommandLoop

Loop01:
     cmp   r0, #Cmd_GetAppID
     jmpr  cc_NE, Loop02
     mov   r0, AppID
     callr SendByte
     jmpr  cc_UC, CommandLoop

Loop02:
     cmp   r0, #Cmd_LoadApp
     jmpr  cc_NE, Loop03
     callr GetByte             ; end address high byte
     movb  rH1, rL0
     callr GetByte             ; end address low byte
     movb  rL1, rL0
     sub   r1, #1              ; adjust code end
     mov   CodeEnd, r1

     callr LoadApp
     jmpr  cc_UC, CommandLoop

Loop03:
     cmp   r0, #Cmd_RunApp
     jmpr  cc_NE, Loop04
     callr APPSTART
     jmpr  cc_UC, CommandLoop

Loop04:
     mov   r0, #'>'            ; the prompt
     callr SendByte
     jmpr  cc_UC, CommandLoop

; --- utilities ---
; CodeEnd holds the end address
;
LoadApp:
     mov   r0, #APPSTART       ; always load apps here
LoadApp01:
     jnb   S0RIR, $            ; wait for byte
     movb  [r0], S0RBUF        ; store byte
     bclr  S0RIR               ; clear receive flag
     movb  S0TBUF, [r0]        ; echo the byte
     jnb   S0TIR, $
     bclr  S0TIR
     cmpi1 r0, CodeEnd         ; inc pointer, check size
```

```
     jmpr  CC_NE, LoadApp01    ; repeat if more bytes
     ret

SendByte:
     mov    S0TBUF, r0         ; transmit char
     jnb    S0TIR, $
     bclr   S0TIR
     ret

GetByte:
     jnb    S0RIR, $
     mov    r0, S0RBUF         ; receive char
     bclr   S0RIR
     ret

; --- this is where apps are to be loaded ---
; --- (OS_SIZE = AppBegin - 0FA60h) ---
AppBegin:

#include <sfr166.inc>          ; SFRs are defined here
```

Listing 2. VSOS Source

VSOS gives us the necessary infrastructure to load and run small applications in internal RAM. We next review three such applications, which collectively provide all the necessary functionality to program internal FLASH.

Programming the Internal FLASH

The 88C166 has 32K of internal FLASH, which when mapped to segment 1 by configuring the associated fields in the SYSCON register, occupies the memory block 10000h to 17FFFh. Internal FLASH is arranged in four blocks. Each block may be erased separately.

The three applications return common error codes as well as the value of the Flash Control Register (FCR). Each application stores the error code and FCR values in internal RAM at locations above FA44h, as discussed above. These variables are called ErrorCode and FcrValue, respectively. The error codes are defined as follows.

```
      ecNoError      equ      0
      ecNoVPP        equ      1
      ecUnstableVPP  equ      2
      ecNoMoreTrials equ      3
```

We now briefly review each application. Only the critical portions of the code are listed for sake of brevity. It is best to download the source code and read the following sections while referring to the source code. In order to download the source code, please (see reference 4.

Checking FLASH Status

Our first utility checks the internal FLASH banks and reports if each bank is erased (all FFs), is cleared (all 00s), or contains data. The code employs a straightforward approach, scanning FLASH one bank at a time. All bytes in a bank are compared to zero as well as FFh. The result of this scan is returned as a single byte. Each pair of bits describe the status of a bank. For example, bits 0 and 1 indicate the status of bank 0. The pair of bits is set to 00, 11, or 01. The bank is cleared (all 00s) if the bits are 00, erased (FFs) if the bits are 11, and contain data if the bits are set to 01.

Clearing and Erasing FLASH

All banks are cleared and erased in one step. Although a more elaborate application could selectively clear or erase a given bank, our application is intended for in-system programming of the FLASH in a production environment. Thus, erasing the entire FLASH is a reasonable approach.

The VPP voltage of 12 Volts must be applied to the 88C166. The application first writes zeros into all FLASH locations. The error code and the FCR value are reported at the end of the clear operation. If any error is encountered, the application terminates. Otherwise, the application proceeds to erase all banks by invoking the erase algorithm. Again, the error code and FCR value are reported back to the host. The host PC declares the operation to be successful if both operations return the error code ecNoError, in which case the FCR value need not be inspected. If an error is encountered, the host PC may further inspect the FCR value.

Besides APPSTART and the error codes, the application defines the start and end addresses of each of the four banks. The application uses two words in internal RAM, at FA44h and FA46h, to store the variables FcrValue and ErrorCode. The major tasks of the application are arranged into separate subroutines: ClearFlash, EraseFlash, EraseFlashBank, Report, Burn, Unlock, Delay4, Delay10, and SendByte. The subroutine Report, using the subroutine SendByte sends the error code and FCR value as described above.

The subroutine Burn programs a single word into FLASH. It returns an error code depending on the result of the attempted operation. The subroutine Burn is called by ClearFlash to program the zeros into FLASH. Note that Burn is also used in the FLASH programming application. The routines Burn and EraseFlash call the utility functions Unlock, Delay4 and Delay10 to handle the specific requirements of FLASH programming.

Subroutines Delay4 and Delay10 set up Timer 3 and wait until a timeout. They effectively hold for 4 and 10 microseconds, respectively. The subroutine Unlock executes the specific steps to gain access to the FLASH programming logic and the FCR register. Please refer to the 88C166 data sheet (see reference 2) for the specific procedures concerning FLASH programming and erasure operations.

The organization of the code into a hierarchy of functional blocks, each implemented as a separate subroutine, usually leads to a simple main routine.

```
        org    APPSTART             ; FB10h

        callr  ClearFlash           ; clear flash
        callr  Report               ; error code and FCR value

        cmp    ZEROS, ErrorCode     ; check error code
        calla  cc_EQ, EraseFlash    ; erase flash if no error
        callr  Report               ; error code and FCR value

        ret                         ; done
```

Subroutine ClearFlash is a simple loop that repeatedly calls Burn for all the words in the range 0 to 7FFEh. The address and data are placed in registers R13 and R14 before calling Burn.

Subroutine EraseFlash stuffs registers R10, R11, and R13 with the start address, end address, and appropriate FCR value, and calls EraseFlashBank. The FCR value determines the bank to be erased as well as some timing information. The start and end addresses of the bank are used in verifying that this range of FLASH locations contain FFs after the operation. Note that the

erase operation is repeated, up to a limit, if not all memory locations are erased. This limit is hardcoded into the program.

The subroutine EraseFlashBank facilitates modularity and code reusability. For example, a future application may erase a given FLASH bank rather than all banks. In this case, EraseFlashBank may be used without any modification. The subroutine Burn is the workhorse for both clear and program operations. The address and data are placed in registers R13 and R14 by the calling program. Burn uses register R12 to keep the number of programming trails left and register R15 to access the FCR register. Burn writes the variable ErrorCode depending on the result of the operation. The specific operations in EraseFlashBank and Burn are dictated by the FLASH hardware (see reference 2).

Programming FLASH

Many of the subroutines used in clearing and erasing FLASH are also used in programming the FLASH. The key subroutine is Burn, which calls Unlock, Delay4, and Delay10. The application expects the data bytes to be received one record at a time. First the size of the record is expected as a single byte, followed by the address at which the bytes of the record are to be consecutively placed. Records are assumed to be 32 bytes or less. Each such record is first placed in memory. The application reserves a block of memory after all its routines using the define segment pseudo-op, as shown below.

```
Record:
       dsw    10h (0)          ; 32 byte buffer
```

Note that the 32-byte data block may or may not be included while downloading the application, provided that the data block is placed after all other code.

The application uses three variables, RecordSize, RecordAddress, and RecordCRC, defined to occupy memory locations FA48h, FA4Ah, and FA4Ch. The application loads the variables RecordSize and RecordAddress, and then proceeds to receive the data bytes. Once the data bytes are placed in internal memory, they are organized into words to be programmed by the Burn routine. After the entire record is programmed, the application reports the error code and the FCR value, followed by a single byte CRC. These may be used by the host PC program to verify the programming operation. Again, all data bytes so received are echoed back to the host for further verification. The application terminates when a record of zero length is received.

Host PC Software

We start with a few design considerations. First, we would like the user interface to be a Win32 (Windows 95 or Windows NT) application. More importantly, though, we would like the FLASH programming functionality of the host software to be as independent of the user interface as possible. This stems from the need to call the FLASH programming software from other user interfaces or applications, particularly from production line control software which may oversee the programming of commercial products built around the 88C166.

Design considerations lead to a software architecture built around two Dynamic Link Libraries (DLLs), provided courtesy of Rigel Corporation. The first DLL, rTTY is responsible for opening and configuring a serial communications port, and providing the basic serial transmission and receive functions. The second DLL, rISP, contains the high-level functions to bootstrap the target processor, interrogate the status of its FLASH, erase its FLASH, and program its FLASH. The following header file defines the exported DLL functions for any Win32 application to use.

```
// --- function return status codes ---
#define     Stat_OK                 0
#define     Stat_Timeout            1
#define     Stat_NoData             2
#define     Stat_BadData            3
#define     Stat_NoFile             4
#define     Stat_NoHandshake        5
#define     Stat_BadHexRecord       6

// --- error codes ---
#define     ecOK           0
#define     ecNoVpp        1
#define     ecUnstableVPP  2
#define     ecNoMoreTrials 3

extern "C" {
__declspec(dllimport)
int Bootstrap(char nHandshake, LPCSTR szBootFile,
              LPCSTR szOsFile);
```

```
__declspec(dllimport)
int  Erase(UINT &uErrorCode, UINT &uFCR);

__declspec(dllimport)
int  GetFlashStatus(int &nBank0, int &nBank1,
                    int &nBank2, int &nBank3);

__declspec(dllimport)
int  ProgramHexFile(LPCSTR szFileName, UINT &uErrorCode,
                    UINT &uFCR);

__declspec(dllimport) void rTtySetBaudRate(int nDivisor);

__declspec(dllimport) void rTtySetPort(int nPort);

__declspec(dllimport) int  rTtyConnect(void);

__declspec(dllimport) int  rTtyDisconnect(void);
}
```

The high-level functions Bootstrap, Erase, GetFlashStatus, and ProgramHexFile return a status code (Stat_XXXX), as defined in the header file. The error codes and FCR values are passed to the functions by reference, so these values are also available when the functions return.

The handshake parameter nHandshake should be 0x55 when bootstrapping the 88C166. Similarly, the parameters szBootFile and szOsFile are the names of the 32-byte bootstrap code file and the name of the VSOS code file, both in Intel Hex format. The bank status returned by the function GetFlashStatus is integer: 0 (cleared), 1 (has data), or 2 (erased).

The low-level function rTtySetPort specifies the serial port to be used. The port number must be between 1 and 4. The Baud rate is set by the function rTtySetBaudRate, whose argument is the standard DOS divisor for the UART. Use a divisor of 2 to specify 56700 Baud communications. Once the port and its Baud rate are specified, the serial port may be initiated by calling rTtyConnect. The other low-level function rTtyDisconnect releases the serial port back to the Windows operating system. Two of the low level functions rTtyConnect and rTtyDisconnect return Boolean results, TRUE (1) if the operation is successful, or FALSE (0) otherwise. For example, a call to rTtyConnect will return FALSE if an invalid port is specified.

A sample user interface was written in Microsoft VC++ to complete the project. The user interface is a very simple application with four buttons, each calling a DLL function. A drop-down list box selects the port. The buttons are enabled only if the port is valid. A read-only scrollable edit box reports the results of the operations. A sample session is shown in Figure 2 below. The source code of the demo user interface (written in Microsoft VC++) is available (see reference 4).

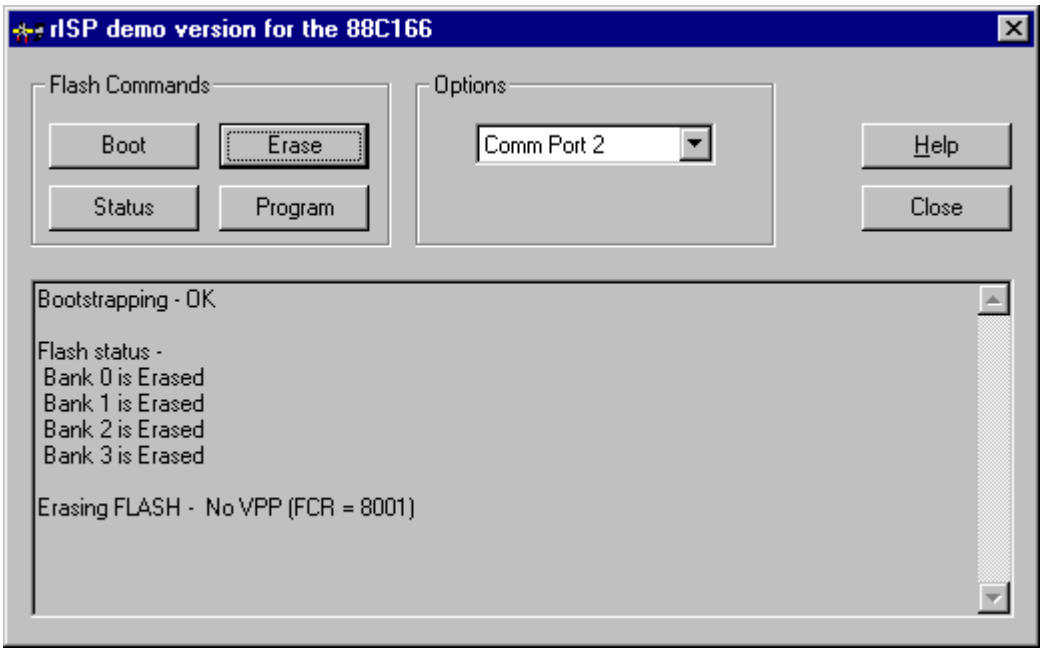

Figure 2. The User Interface for the In-System FLASH Programming of the 88C166.

Conclusions

In retrospect, one can safely say that the project benefited from the modular approach taken. Once the bootstrap code and VSOS were running, work focused on writing FLASH programming utilities. These utility applications were individually developed and tested by loading them under VSOS. Care was given to writing separate routines for distinct logical tasks. This further simplified debugging and project management. Moreover, such routines are more easily reused, as it was observed with the routine Burn, increasing their value-to-development-effort ratio. A similar view was taken in developing Win32 code. Importing the serial port support from an existing DLL (rTTY) reduced much of the host-level debugging effort. Similarly, the bootstrap, erase, get status, and program functions were collected in a new DLL (rISP), rather than simply including them along with the user interface code in the final application. These DLLs not only allow the user interface to be written independently in a number of Win32 languages, but also allow incorporating FLASH programming functionality to existing software environments. One user has already incorporated the DLLs into his own WindowsNT-based production control environment so that the 88C166-based products may be programmed on the production line.

Downloading Source Code

The assembly code for the 88C166 was developed using the demo version of Reads166 V3.00. You may download Reads166 from Rigel Corporation's web site (see reference 4). The assembly code used in this article as well as the demo user interface (written in Microsoft VC++) is also available on the web site.

References

1. Stengl. D. and WS T. Nicolai, "SAB 88C166-5S Flash-EPROM Programming", a collection of bootstrapping, FLASH programming, and MS DOS monitor code, internal document, Siemens AG, 1992.
2. SAB 88C166/83C166 Data Sheet, Edition 5.94, Siemens, AG.
3. Bootstrap Loader, Section 2.1.1., Collection of Application Notes for the C166-Family, Siemens Components, Inc. Cupertino, CA, 95014-0716,
4. Complete source code and further information is available from www.rigelcorp.com.

5. The FLASH utility, rFLI, for both the 88C166 and the C167CR-FM16, may be downloaded from www.rigelcorp.com.

Sencer Yeralan, P.E., Ph. D.

About the author
Dr. Yeralan has been teaching industrial automation and control at the University of Florida for the last ten years. He designs the C166 family evaluation and OEM boards made by Rigel Corporation. He is currently working on the textbook “Programming and Interfacing the C166 Microcontroller.”