Basic Application Loading Over Serial Interface for the DaVinci DM644x

Application Report

Literature Number: SPRAAI0
December 2006

Texas Instruments

Contents

List of Figures

List of Tables

Trademarks

Code Composer Studio is a trademark of Texas Instruments.

Linux is a registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft, Microsoft Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.

MontaVista is a registered trademark of MontaVista Software, Inc..

Novell is a registered trademark of Novell, Inc. in the United States and other countries.

Basic Application Loading Over the Serial Interface for the DaVinci TMS320DM644x

Daniel J. Allred

ABSTRACT

This application report describes two related pieces of software that are used together to download an application over the DM644x UART0 serial interface and run it out of the ARM internal memory. The discussion begins with a description of a host application that executes on a user's PC, and includes a description of how to interface to the ROM boot loader of the DM644x device. The discussion of the embedded ARM application that runs on the DM644x describes using the ARM GNU cross-compiler toolchain provided by MontaVista to generate an appropriate binary file for download. The target application that is presented provides examples of how to initialize and use the UART peripheral of the DM644x and the DDR2 memory subsystem that is found on the DVEVM board.

This application report contains project code that can be downloaded from http://www.ti.com/lit/zip/SPRAAI0.

1 UART Application Boot Overview

This section gives an overview of the process of sending a binary application image to the DM644x device over its serial interface for execution. As an example, this scenario would be useful for downloading test code in a production setting as part of quality control procedures. This section mirrors the presentation given in the UART Boot Mode section of *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14). The reader should be familiar with all the information in the Boot Modes section of that document. Additional details can be found in the Bootmode section of the *TMS320DM6446 Digital Media System-on-Chip* data manual (SPRS283).

Upon reset or power up, the DM644x begins executing code from its ROM boot loader (RBL). The RBL determines how to boot based on the logic levels at pins BTSEL[1..0]. These values are latched into the BOOTCFG register bits 7..6 at reset and the RBL reads this register to determine the intended boot mode. On the Spectrum Digital DVEVM board, these pins are attached to switches S3-2 and S3-1 respectively, allowing the user to change the intended boot mode while the board is powered off. Table 1 shows the available boot modes for the DM644x.

The only boot mode of concern for this application report is UART boot. The objective of the UART boot mode is to interface to a host system via a serial cable and download code that the system can use to boot. The traditional intention is that a secondary boot loader, referred to as a User Boot Loader (UBL) in the *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14), would be downloaded. This UBL would then be used to initialize certain subsystems of the chip, in particular the DDR2 memory controller, and then download a tertiary boot loader like U-boot, which has been written with the objective of loading the Linux® kernel onto embedded systems.

Table 1. Boot Modes for DM644x Device

BTSEL0 (DVEVM SW3-1)	BTSEL1 (DVEVM SW3-2)	Boot Mode
0	0	NAND Flash Boot
1	0	NOR Flash Boot
0	1	Reserved
1	1	UART Boot

The code that is downloaded, however, does not have to be a boot loader. It can be an application in its own right. This document presents a simple application that targets the ARM side of the DM644x and is downloaded over the UART at boot time. This application is referenced as the UART application for the remainder of this document and is discussed in Section 4. To assist in the transfer of the UART application, another program runs on the host system (assumed to be a personal computer) to send the required data at the appropriate time. This program is called the host application for the remainder of this document and it is described in Section 3. Section 2 presents details of the RBL's UART boot process and the negotiations that the host application needs to understand to deliver the code to the ARM subsystem of the DM644x device.

2 ROM Boot Loader Negotiations

This section describes the details of the signals the ROM Boot Loader (RBL) of the DM644x transmits and expects in the UART boot mode. Figure 1 provides a graphical overview of the RBL's program flow in UART boot mode. As specified in the *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14), the RBL is set up with a UART timeout of 500 ms, which means this program flow restarts after 500 ms of waiting for an expected byte without receiving it. This timeout is not shown in Figure 1.

The negotiation prompts from the RBL consist of 7-byte strings with an eighth null terminating character. The carat symbol, "^", is used to indicate a space in the prompt sequences. The sequence "\0" is the standard C escape sequence for the NULL (zero) character and therefore represents only a single byte.

2.1 ACK Header

In addition to the prompts sent by the DM644x's RBL, the RBL also expects to receive an eight byte sequence, "^^^^ACK\0", as the beginning of a header describing the application bytes that the user ultimately wishes to transmit. Following Figure 1, this header should be sent in response to the RBL's initiating "^BOOTME\0" sequence. Table 2 gives details of this header and its contents.

Table 2. The ACK Header in Detail

Data	Width in Bytes	Description
"^^^^ACK\0"	8	Expected character sequence to begin the ACK data header.
CRC-32 Checksum	8	Eight ASCII text characters representing the hexadecimal representation of the 32-bit CRC32 checksum of the bytes of the application code intended for transmission.
Byte Count	4	Four ASCII text characters representing the hexadecimal representation of the number of bytes in the binary application code.
Entry Point Address	4	Four ASCII text characters representing the hexadecimal representation of the address to which the RBL should hand execution upon successful transmission.
"0000"	4	Four terminating '0' characters (Not '\0', or NULL, characters)

One key aspect of the data sent after the "^^^^ACK\0" sequence is that it is all ASCII characters that represent hexadecimal digits. That is, each byte sent is drawn from the set [0-9a-fA-f] and represent 4 bits (one hexadecimal digit) of the binary data. As an example, to send a CRC32 checksum of 0x12345678 the host should send the eight bytes 0x31 through 0x38 – the ASCII characters "1" through "8".

The RBL checks the validity of the input byte count and the entry point. The input byte count must be less than 0x3800 = 14 kB. The entirety of the transmitted data is stored in the internal tightly-coupled RAM (TCM) of the ARM processor starting at address 0x0020. The internal RAM is 16 kB in size, leaving some space for the 1024-byte CRC lookup table, as well as the stack and un-initialized variables. If the byte count is too large, the RBL sends the BADCNT sequence. The entry point of the binary application must be between 0x100 and 0x3800. If it is not in this range, the RBL returns the BADADDR sequence.

The difference between the lowest allowed entry point, 0x100, and the starting memory location of the binary, 0x20, allows the inclusion of self-copy code at the beginning of the binary application image for use in multiple circumstances (e.g. NOR boot and UART boot). This is not needed for this example. To make use of this memory space, the example application places data, instead of code, at these lower addresses.

2.2 CRC32 Table

If the header data is accepted as valid, then the RBL replies with the BEGIN sequence. This indicates that the RBL is ready to receive the 1024-byte CRC32 lookup table. The CRC32 value sent in the ACK header is used to verify that the application data received matches the data transmitted. The RBL expects the data to be transmitted as 256 four-byte words (the format in which the table is constructed) converted to their ASCII hexadecimal representation. For example, if the first two words of the table are 0xABCD0123 and 0x1A2B3C4D then the characters to be transmitted are “ABCD01231A2B3C4D”. The actual values that this table holds are found in Appendix A.

There are two potential pitfalls with this checksum scheme. First, the checksum value sent in the header could have been corrupted during transmission. Second, one or more elements of the CRC32 lookup table could be corrupted in transmission, resulting in an incorrect calculation of the checksum on the DM644x. Either of these results in boot failure, even if all of the bytes of the application code are received error-free. Using the limited set of ASCII characters to transmit hexadecimal representations of the binary data helps catch errors of both types. In addition, to protect the transmission of the large CRC32 table, a checksum8 is calculated for all of the bytes of the table. This checksum8 value is the least significant byte of the sum of all bytes in the lookup table and should be equal to 0x00. As Figure 1 shows, if this checksum8 check fails, the RBL replies with the CORRUPT sequence and the boot process begins again with BOOTME.

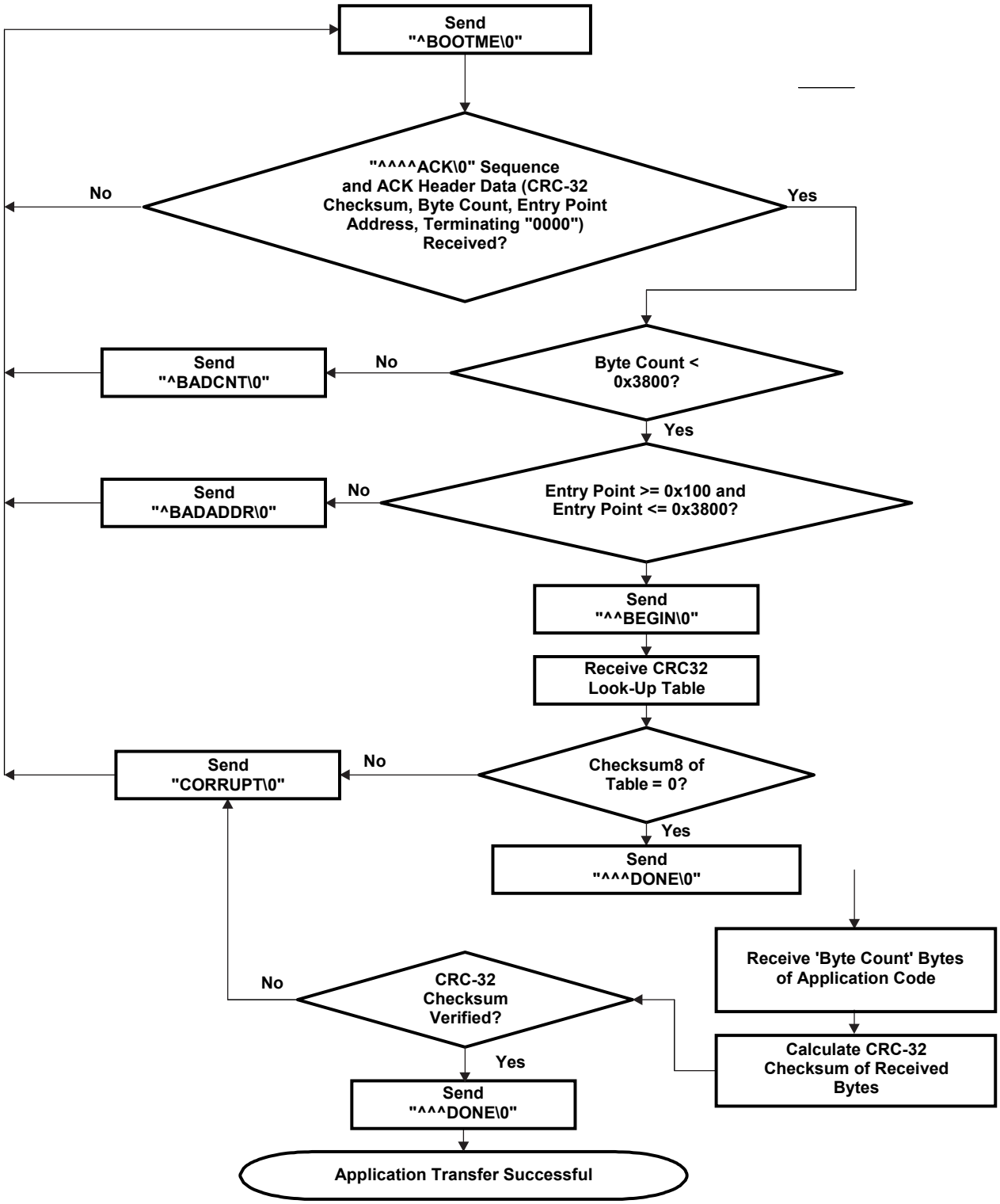

Figure 1. ROM Boot Loader UART Mode Program Flow

2.3 Application Data

If the CRC32 table is accepted as valid, the RBL indicates this with the DONE sequence. The RBL then expects to see the transmission of the application code on the serial line. The RBL waits until all bytes, specified by the byte count sent in the ACK header, are received. As was the case for the CRC32 table, the data is expected to be transmitted as ASCII hexadecimal representations of four-byte words. Consequently the binary application image must be four-byte aligned (byte count must be a multiple of four).

When all expected bytes are received, the CRC32 checksum is calculated on these bytes, starting at the lowest address filled (0x20) through the highest address filled (0x20 + byte count). It is important to remember that the order of the bytes as they are stored in memory must be the same as the order of the bytes as they were used to calculate the CRC32 checksum that was sent in the ACK header. For example, even though 0x12345678 is sent as “12345678”, it is stored in memory as bytes in the following order (from low address to high address): 0x78, 0x56, 0x34, 0x12.

2.4 Important Notes Regarding CRC32 Checksum

Note: The standard CRC32 algorithm (used in Ethernet, PKZIP, FDDI, etc.) uses the reflected, inverted form with the polynomial 0x04C11DB7. But the CRC32 checksum function performed in the ROM of the DM644x **DOES NOT DO THE FINAL BIT INVERSION** when it calculates the CRC on the received data. Consequently, the checksum value that the host should send in the ACK header should be the bitwise inversion of the checksum generated by the standard algorithm that is commonly in use. The CRC32 lookup table that is sent to the DM644x is not affected.

Note: If desired, the CRC32 checksum verification can be bypassed by sending a 1024-byte table of all zeros and a CRC32 checksum value of 0x00000000 in the ACK header. However, for maximum data integrity, the CRC32 checksum should be fully and correctly implemented.

3 Host Application Example

There are two types of host applications that can be created to assist in getting an application binary transmitted to the DM644x internal memory. The first simple type can input the application data and create an output text file with all of the sequences and data already converted to hexadecimal ASCII format. This file could then be downloaded as a text file in its entirety over the serial port using a terminal emulation program like HyperTerminal or Minicom. If this approach is taken, a character delay of at least 1 ms must be set to give the RBL time to do its processing.

The second approach to the host application is to create an active application that communicates over the serial port and responds in real-time to the RBL. In this case, the data transmission can take place at full speed because the data is ensured to be sent at the correct time. This approach is somewhat more complicated, but also does not require the use of an additional terminal program. In addition, it provides the ability to handle errors and gain feedback on the negotiation process that cannot be done with the first passive approach.

The host application attached to this application report follows the second, active paradigm. This section describes some details of this host application that is executing on the PC connected to the DM644x board. The host application handles all the required negotiations described in Section 2, generating appropriate responses at the appropriate times. This section describes, in general, what the host application must do and, in specific, how the provided application does these things.

3.1 Overview

The host application must first check how it was called and make sure all needed command-line parameters were provided. Then, it must open the host's serial port to communicate to the DM644x. Next, it must load the binary application file and calculate its CRC32 checksum. Finally, the host application should perform the negotiation over the serial connection as described in Section 2. The following sections give some details and explanations of the example host application and how it handles these requirements. The most complete documentation, however, is the source code itself.

The host application was written in the C# language. C# is a standardized object-oriented programming language (ECMA and ISO standards) that, like Java, aims to be cross platform by targeting a common language infrastructure (CLI). An implementation of the CLI has been developed for multiple platforms and programs compiled on one platform for one implementation should work equally well on another. This is accomplished partially because the compiled code exists as an intermediate language.

Surrounding the CLI, there can be a precompiled framework of classes that are used to perform common functions and provide basic utilities (much like the standard libraries of C or C++). Microsoft® has provided a framework of this type known as the Microsoft® .Net Framework, currently at version 2.0. An open source project sponsored by Novell®, called the Mono Project, has developed an open-source cross-platform CLI and framework that is mostly compatible (and aims to be fully compatible) with Microsoft's closed .Net Framework. The provided host application has been tested and verified to work on both frameworks. As a result, the application can be used on a host running Microsoft Windows® or any recent Linux distribution. Under Windows, the application can use either the .Net Framework (provided for free at http://www.microsoft.com/) or the Mono Framework (available at http://www.mono-project.com/). Under Linux, the Mono Framework is the only option.

The code is found in the *DVLoader.cs* file and the compiled executable is *DVLoader.exe*. The source code is internally partitioned into two classes – a main program class and a CRC32 class. The program class consists of the Main() function, where program execution begins, the TransmitSerialApp() function, where the serial negotiation takes place, and the ReadSeq() function, used as a helper function by TransmitSerialApp() to wait for the sequences coming from the DM644x RBL. The CRC32 class consists of a public constructor and a public CalculateCRC() function. When the constructor is called, a private BuildTable() function is used to generate the CRC32 lookup table. This table is then used for calculating the CRC locally and is sent to the DM644x's RBL during the serial negotiation. The console output is shown in Figure 2.

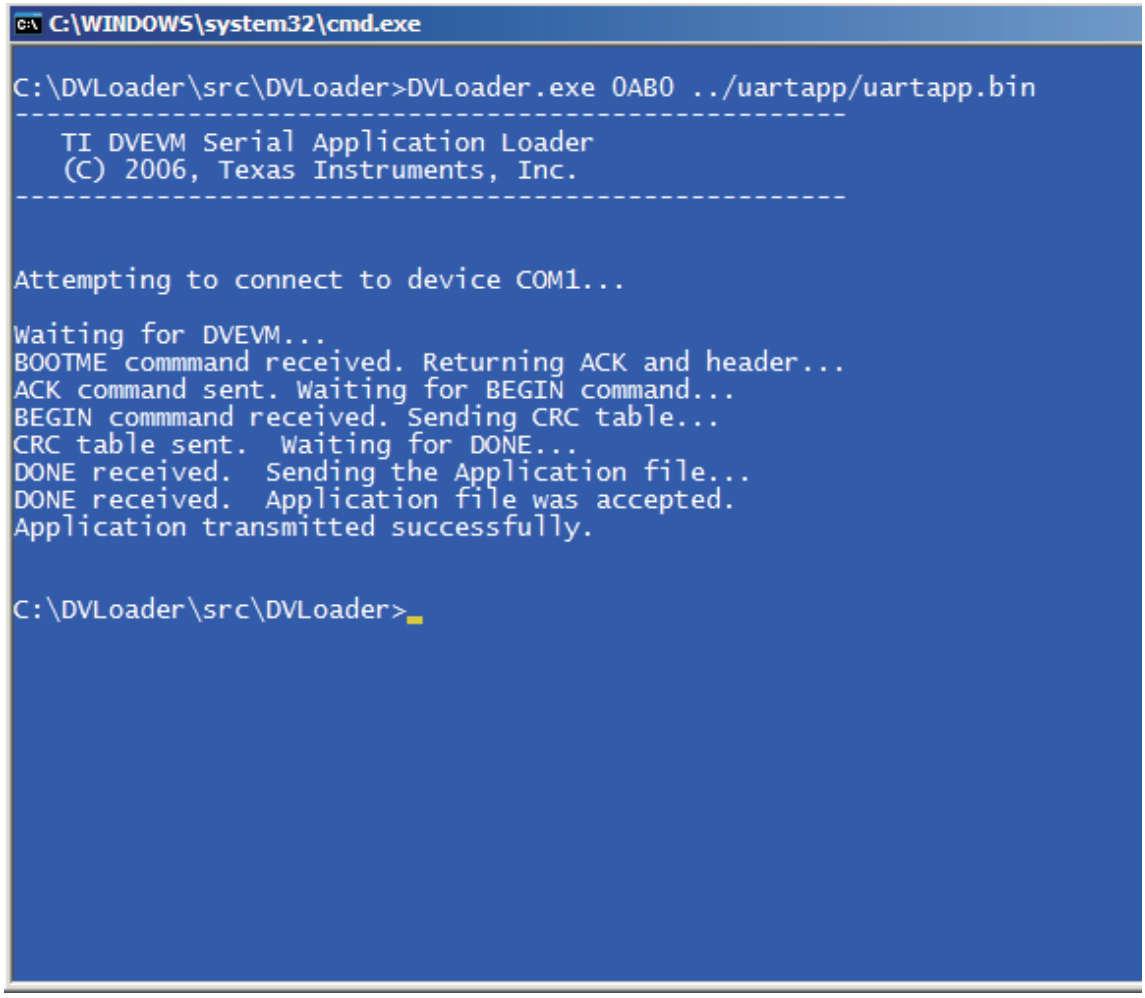

Figure 2. Console Output

3.2 Command-Line Parameters

Calling the host application requires one or two command-line parameters. The first optional parameter is the absolute entry point (memory address) of the application as a four-character string representing the address in hexadecimal format. If not present, the entry point address sent in the ACK header defaults to 0x0100, the lowest possible value the RBL accepts.

The required parameter is the filename of the binary application that is to be sent over the UART. The filename can include a fully qualified or relative path from the current directory. This file should be byte-for-byte the binary data that is intended for transmission to the DM644x. Therefore, the maximum file size is 14 kB.

The Main() function parses the command-line arguments and sets internal variables with the appropriate default or provided values. If the number of command-line arguments is invalid, the program displays a simple help message.

3.3 Serial Port Access

After parsing the command-line parameters, the Main() function needs to open the serial port of the host. Opening the port requires the name of the port, which varies depending on which platform the program is executing. Under Windows, COM1 is the first serial port name. Under Linux, the first serial port is /dev/ttyS0. The Main() function identifies what platform the system is running on and sets the serial port name accordingly.

The serial port object (part of the Framework) is created using the determined name and with the settings of 115200 baud, no parity, 8 data bits, 1 stop bit (written 115200 8N1). Before opening the port, the encoding format is set to ASCII. This ensures that text written to the port is sent as 8-bit ASCII characters. The timeout for read operations is also set to 500 ms, matching the read timeout of the DM644x RBL. Finally the port is opened.

The Main() code catches any exceptions thrown by the open command. Specifically, if an UnauthorizedAccessException occurs, the program warns that the port is most likely already in use by another application and then terminates. This message would appear, for instance, if HyperTerminal has an open connection on the port when this program is executed under Windows. To keep this program simple, there is no command-line option to change the intended host port to a higher number. The first serial port (COM1 or /dev/ttyS0) is likely to work in most situations. If, however, a higher port number must be used, the supplied code easily can be modified and recompiled. The README file supplied in the code archive describes how to do this.

3.4 CRC32 Calculation

If opening the serial port succeeds, the Main() program branches to the TransmitSerialApp() function, passing the entry point address and the filename of the binary application. The binary application file is opened and read as a sequence of 32-bit words. As these words are read, the ASCII hexadecimal representation of the each word is appended to a string. Also, each word is split into its constituent bytes and these are appended to an array of bytes, with the least significant byte first (same order as in the memory of the ARM little-endian architecture).

When all of the words have been read from the file, this array of bytes holds the binary image in the host's memory. Next a CRC32 object is created. The array of bytes is passed to the CRC32 object's CalculateCRC() method and the return 32-bit value is the standard CRC32 checksum. That value is then bitwise inverted to comply with the special note in Section 2.4. This final value is passed to the RBL in the ACK header during the serial negotiations.

3.5 Serial Negotiation With the DM644x RBL

The remainder of the TransmitSerialApp() function serves to implement the serial negotiation described in Section 2. The ReadSeq() function is used to wait for the output sequences of the DM644x RBL. If the expected output sequence is not received within the timeout period, then the negotiation process begins again by checking for the BOOTME sequence. This functionality mirrors the boot failure process that takes place on the DM644x.

The code continues looping indefinitely until the negotiation completes successfully. To end the program prematurely, you must interrupt execution using the Ctrl-C keystroke. The total execution time to complete the negotiation and transmit a 14 kB binary application is approximately three to four seconds. The host application prints status messages indicating how the negotiation process is proceeding. When the binary application data has been successfully transmitted, the TransmitSerialApp() function completes and control returns to the Main() function. There, the serial port is properly closed and the program concludes.

4 ARM Target Application Example

In this section, details of the included example UART application are presented. In general the UART application can do whatever the developer would like, subject to some limitations discussed in the Section 4.1.

To be useful the application needs to do a few things. One of these things is to initialize the DM644x UART module and communicate over it to the host PC. This provides an I/O path to view diagnostic output or show a menu and receive a selection from the user on the host PC. Another useful ability for the UART application is to enable and use the DDR2 memory interface built-in to the DM644x. This could be used to download a more extensive ARM-side application (e.g. U-boot) or to download DSP code and boot the C64+ core out of the DDR2 memory space.

Because these two items form the basis for a useful UART application, the example application included with this document does these two things. It is intended that the user on the host PC interacts with the UART application via a terminal program, such as HyperTerminal or Minicom, after the host application exits. The code was written with the intention of being compiled using the MontaVista® Linux cross-compiler GNU toolchain. Texas Instruments Code Composer Studio™ is neither required nor supported. The included code archive includes a makefile, a linker script, three C files, and three header files. The contents of these files are discussed in Section 4.2 and Section 4.3.

4.1 Limitations

The main limitation of the UART application is its code size. The application is limited to 14 kB of instructions and initialized data. Complex applications may need more memory than this. This is why the UART application can commonly be a User Boot Loader (UBL), whose job is to initialize the external memory system and then return to the host PC to download a bigger, more complex application.

The only way to overcome this limitation and still run the application out of the internal TCM RAM is to modularize the application and write a much more extensive host application to load needed modules as requested over the serial link. Similarly, the application base could be used to initialize the DDR2 memory, download all modules to the external memory, and then swap them in as necessary. But, for a single monolithic application, without these types of workarounds, space is limited.

There is also a possible performance limitation that the developer should be aware of. This limitation can easily be overcome as part of the system initialization (see Section 4.3.1). When the DM644x begins execution of the downloaded UART application, the ARM core is only operating at half of the oscillator input frequency. On the DVEVM, the clock source is 27 MHz, so the ARM core is operating at 13.5 MHz to begin with. This is the processor's safe mode. The ARM core is designed to run up to 297 MHz, and can be set to do so by modifying the settings of the PLL1 peripheral. The example code provided does this as part of the platform initializations.

4.2 GNU Toolchain Specifics

The UART application code is designed to run on the ARM core of the DM644x in what can be called a "raw" format. There is no underlying operating system or other runtime support. There is no loader to decode the file format, place sections in memory, and set up the stack. The compiled code must simply be placed in memory by the RBL and executed as is. This requirement demands particular commands, settings, and linking to create a binary that can be used in this manner. This section details the way in which the GNU tools are used to meet this requirement, principally through the use of a linker script, used to place object code at particular memory locations, and a makefile, used to call certain GNU tools to generate the appropriate binary file format.

4.2.1 Linker Script

The linker script found in the code archive is *uartapp.lds* (*ld* is the GNU linker and *s* stands for script, though neither the filename nor the extension is consequential). The purpose of the linker is to organize the sections of code in the compiled object files into a single executable object file. The linker must resolve symbol names by replacing them with memory addresses. These addresses are ultimately determined by the contents of the linker script.

For this application, the linker script is critical for two reasons. The first reason has to do with the internal memory architecture of the ARM core of the DM644x. The internal RAM consists of two 8 kB pages of RAM and two buses to access these blocks, one for accessing instructions and one for accessing data. The two buses access the same physical memory locations, but they are logically mapped to different address ranges in the system memory space (see Tightly Coupled Memory section of the *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14)). This organization allows simultaneous access to an instruction word in one page and a data word in another page. The instruction bus is mapped to 0x0000-0x3FFF, and the data bus is mapped to 0x8000-0xBFFF. The linker script must be used to appropriately place data sections into the data memory bus range and instruction sections into the instruction memory bus range. Additionally, the linker script must be written to insure that the data sections and instruction sections do not overlap in the physical memory. The linker will not provide any warning to prevent this because it has no knowledge that the two memory ranges access the same memory. The developer must prevent this.

The second reason that the linker script is so critical is that there is an offset of 0x20 bytes between the load address of the binary and the run address of the binary. This is due to the RBL copying the entire image to the internal memory starting at location 0x0020 (actually 0x8020) instead of 0x0000. The binary image can and should have code and/or data starting at location 0x0000 of the file, but when this file is downloaded to the board, the RBL places this code and/or data starting at location 0x0020. The linker script provides the means to handle the discrepancy.

Next, various parts of the linker script are explained. More information on the format and commands for GNU linker scripts can be found in the latest GNU documentation. See Section 5 for the location of the documentation.

This command sets the entry point of the application to be the boot symbol, i.e. the address of the boot() function.

```
ENTRY(boot)
```

The Sections command is used to create the layout of the code and data sections that goes into the output object file. Within this command, the output code and data sections are created by mapping sections of the input object files into these output sections.

The first line below sets the location counter (the '.' symbol) to be 0x8020. This means that the next defined output section, the read-only section `.rodata`, begins at this address. The `AT` keyword specifies that this section should be loaded at the beginning of the output object file, address 0x0. But the linker resolves any symbols in the code that reference data in this section to the address range beginning at 0x8020. The `.rodata` output section consists of the `.rodata` and `.rodata*` sections taken from each of the input object files. These sections consist of read-only or constant data, like strings sent by the UART commands The `ALIGN(4)` statement forces the location counter to be a multiple of four, so this section terminates on a 32-bit word boundary.

```
. = 0x00008020;

.rodata    : AT ( 0x0 )
{
     *(.rodata*)
     *(.rodata)
     . = ALIGN(4);
}
```

This command creates the `.data` section in the output object file using the `.data` sections of each of the input object files. Its runtime location follows immediately after the end of the `.rodata` section. The load address, following the `AT` keyword, is the sum of the `.rodata` section's load address and size. In other words, the `.data` section is loaded immediately after the `.rodata` section. The `.data` section consists of global variables that are initialized in the code.

```
.data : AT ( LOADADDR(.rodata) + SIZEOF(.rodata) )
{
     *(.data)
     . = ALIGN(4);
}
```

The following commands specify the output sections of program instructions. To begin, the location counter is decremented by 0x8000 to move the run-time address of these sections to the instruction bus region of the internal RAM. The load addresses of these sections, however, are made relative to the previous sections, again stacking these sections immediately after the previous ones in the binary output file. The `.boot` section, which holds the entry point boot() function is placed second so that it most likely comes after the 0x0100 lowest allowed entry point address. If that is not the case, then the `.boot` section needs to be forced to begin at 0x100 by editing the linker script.

```
. -= 0x8000;
.text : AT ( LOADADDR(.data) + SIZEOF(.data) )
{
     *(.text)
     . = ALIGN(4);
}
.boot : AT ( LOADADDR(.text) + SIZEOF(.text) )
{
     *(.boot)
     . = ALIGN(4);
}
```

These commands first increment the location counter back up by 0x8000 in order to place this section in the data bus region of the internal RAM. This `.bss` section is used as a location to access un-initialized variables that the code may need to use. Because this section does not actually contain any data at load time, a load address is unnecessary. This space can even be past the 14 kB RBL limit because it does not add any size to the binary image that is downloaded. Of course, no data can exist at or above 0xC000, so the developer needs to be aware of how much space un-initialized data might take up.

```
. += 0x8000;
.bss :
{
     *(.bss) *(COMMON)
     . = ALIGN(4);
}
```

The last part of the Sections command specifies a symbol for the location of the top of the stack, and two other sections corresponding to the AEMIF CS2 memory region (occupied by NOR or NAND flash on the DVEVM) and the DDR2 memory space. Technically, these sections are not required as the addresses could be fixed in the C code, but the linker script seems an appropriate centralized place to modify memory space settings. This is especially true if there is a chance that the sections may move in the future (if a design uses the AEMIF CS3 space instead of the CS2 space, for example).

The `_topstack` symbol is set to the top of the data bus internal RAM region as 0xBFFC. It is used in the boot code to set the stack pointer, which grows down from this location. In this case, care must be taken by the developer to make sure the `.bss` section and the stack do not overflow into each other. There is no built-in mechanism to do this. Figure 3 gives a visual overview of how the different sections are organized, first in the binary application file, and then in memory on the ARM core of the DM644x.

```
__topstack = 0xC000 - 0x4;

. = 0x02000000;
.aemif :
{
     *(.aemif)
}

. = 0x80000000;
.ddrram    :
{
     *(.ddrram)
}
```

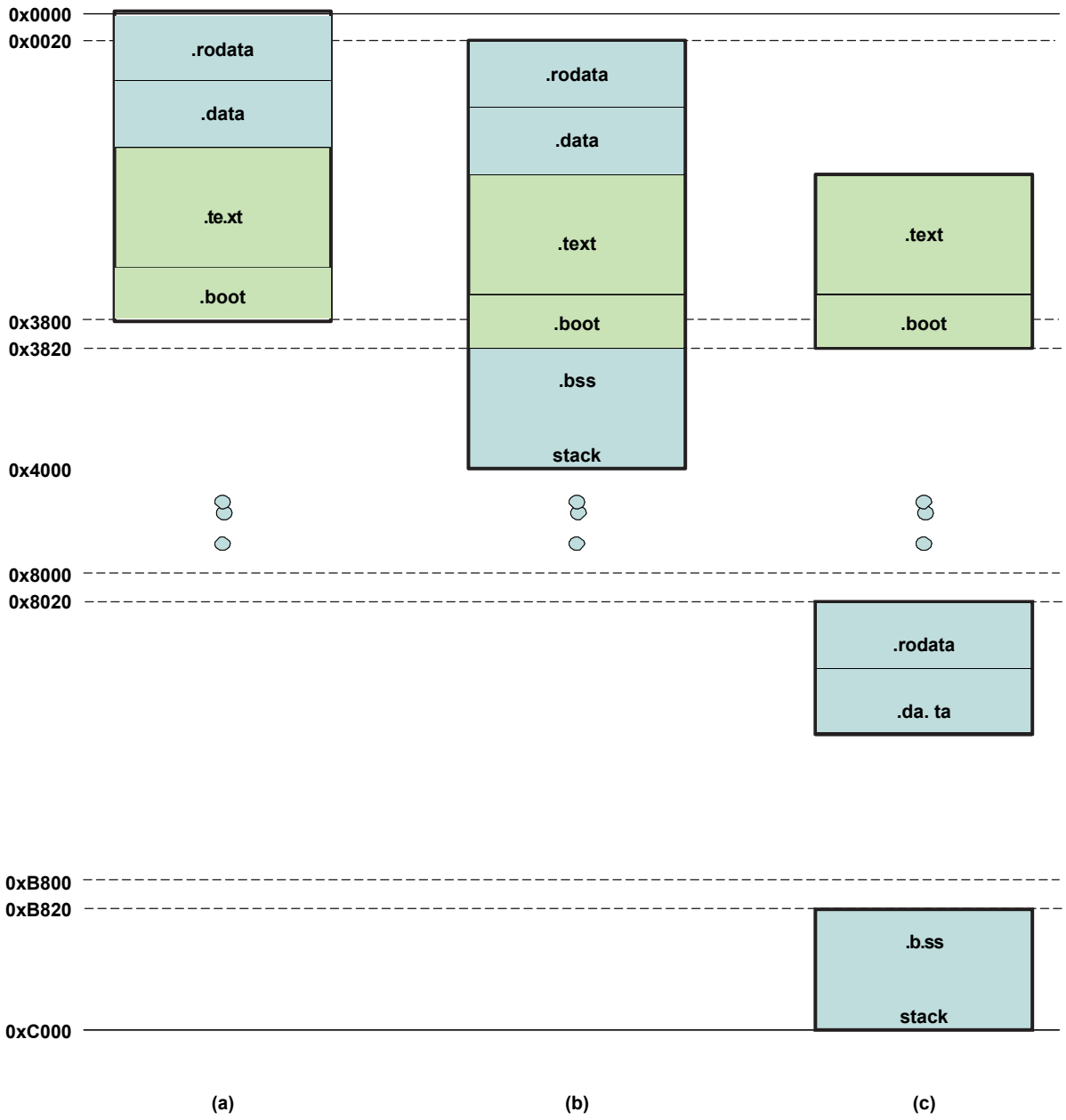

Figure 3. View of Object File Sections

Figure 3 shows the object file sections in the following views:

(a) Binary Application File

(b) Physical Memory

(c) Logical Memory (showing separate data and instruction bus memory spaces)

4.2.2 Makefile

The makefile controls the build process, including calling the linker that utilizes the linker script discussed in Section 4.2.1. The GNU tools that it calls are the key to generating the proper output image. The makefile takes the following steps:

1. Use the cross-compiler gcc executable to compile the *.c files into object files (*.o) files.

   The -c flag indicates that the gcc tool should do only compilation and assembly, no linking. The –Os flag indicates that the compiler should optimize the output for size. The -Wall flag indicates that all warnings should be produced, where applicable.
2. Generate an ELF executable by using the cross-compiler gcc executable (which internally calls the GNU linker ld) to link the objects files.

   Here gcc is used only for linking the already compiled object files. The -Wl flag passes the -T$(LINKERSCRIPT) flag directly to the internal call to the GNU linker ld, so that the linker uses the script in determining the output. The -nostdlib flag is very important for generating a correct “raw” binary image. It prevents the linker from trying to compile in the system startup libraries, which usually provides the entry point to a program. The linker script specifies the entry point, and the application does not need a loader, so these libraries and files are not needed.
3. Use the cross-compiler version of the GNU objcopy tool to convert the ELF executable to the binary format.

   The objcopy command is used to convert from one object file format to another, in this case from ELF to binary. This essentially produces a memory dump starting at the load address of the lowest section in the ELF file. According to the linker script, this is the `.rodata` section at load address 0x0. See Figure 3(a) for a view of the binary application image structure.

   The -R flags are used to remove the .aemif and .ddrram sections because these aren’t in the memory range of the internal RAM. The *--gap-fill 0xFF* command fills any empty bytes between sections with the value 0xFF. The *--pad-to 0x3800* pads the binary output file to be exactly 14 kB in length. This padding is not needed, but does make any binary images consistent. Finally, the -S option tells the objcopy command to strip all debugging, relocation and symbol information.
4. Use the cross-compiler objdump utility to find the address of the entry point boot() function.

   The final step in the makefile is to analyze the ELF executable, which still has its symbol information, to determine the address of the boot symbol, specified as the entry point in the linker script. Running the cross-compiler objdump command on the ELF executable with the –t command displays the symbol table. The makefile pipes this output to the grep command, which finds the line containing the address of boot. This line is then piped to the sed command, to parse the address out by itself, where it is displayed. This value can then be used as a parameter on the host application command-line.

   The online manuals for the GNU tools can be found at http://www.gnu.org/manual/manual.html. The objcopy and objdump commands are part of the binutils package, also documented at the GNU website found at http://www.gnu.org/.

4.3 Discussion of C Code

As with the host application, the best ultimate documentation for the UART application is the code itself. However, this section attempts to explain in some detail the most important aspects of the code. This section is divided into four subsections. The first deals with the boot code that must be executed before the main body of the application can begin. The next section briefly discusses the main program. The following section concerns the platform initialization, which principally consists of starting the DDR2 memory system. The final section presents some simple functions to send and receive data via the UART.

The code consists of three C files and three corresponding header files. The file *dm644x.c* contains the entry point boot() function and the system initialization code. The *dm644x.h* header file also defines a number of structures and preprocessor macros for accessing system and peripheral registers. The *uart.c* and *uart.h* files define the functions to use the UART peripheral to send and receive data, though the UART initialization procedure is found in the *dm644x.c* file. Finally, the *uartapp.c* and *uartapp.h* files contain the uartapp_main() function, which presents the application menu. The action functions that are called based on the user’s menu selection also are found in these files.

4.3.1 Boot Code

The boot() function is prototyped in the “dm644x.h” header file as follows:

```
void boot( void ) __attribute__((naked,section (".boot")));
```

The attributes attached to the function prototype give the gcc compiler additional information about how to convert the C function to assembly code. The first attribute, `naked`, tells the compiler that the function prologue and epilogue should not be generated. These normally contain the commands to save and restore registers. Because this function is intended to be the program entry point, these are not needed. The second attribute specifies that this function should be placed in the `.boot` section of the object file. This is used to by the linker script to place the entry point at an appropriate place in the binary application image.

The boot code must do one important thing so that the further function calls work correctly – it must set the stack pointer. This can only be accomplished through loading the *SP* register of the ARM core with the value given in the linker script.The code below does does this by utilizing the extended ASM capabilities of the GNU compiler.

```
extern int __topstack;
register int* stackpointer asm ("sp");
stackpointer = &(__topstack);
```

The boot code then calls the uartapp_main() function, found in `uartapp.c`, to start executing the true program. When the main function returns, execution enters an infinite loop intended to halt the program execution so that it does not begin executing undefined instructions after the end of the boot() function.

4.3.2 Main Program

The main function, uartapp_main(), first calls PlatformInit() in *dm644x.c* to set up the system peripherals (see Section 4.3.3). Then it displays a prompt for the user to press any key and waits for a single byte to arrive from the UART system. If the UART receive function times out, then the prompt is displayed again and the wait starts over. This allows the user to control when the main menu is displayed. This is needed to give the user time to start a terminal program on the host, after the host console application has terminated.

The main menu of the program is displayed after the keypress loop is passed. The menu offers three options to the user and prompts for a choice. The application loops waiting for the UARTRecvData() function to succeed and return the input choice. A switch-case block checks for the characters 1, 2, or 3 and proceeds to execute the desired choice. If an invalid choice came from the host, a message indicating this is displayed. After the switch block, the uartapp_main() function terminates and execution returns to the boot function, where the program enters an infinite while() loop. An example of the terminal output is shown in Figure 4.

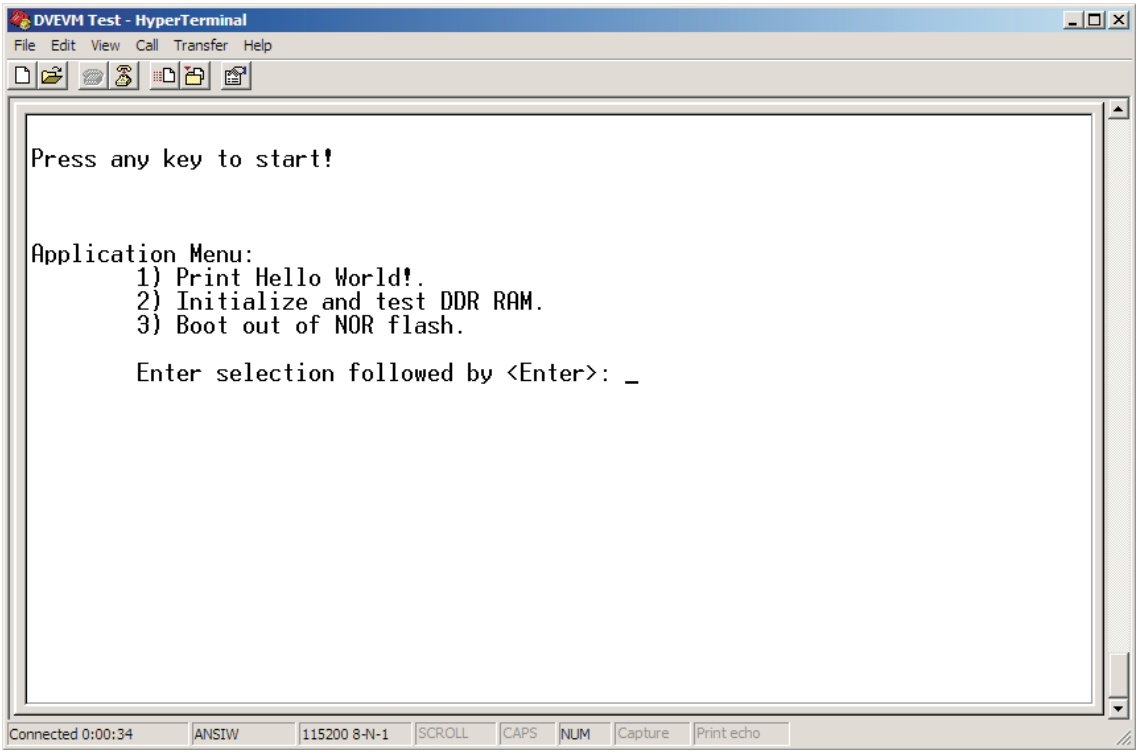

Figure 4. Terminal Output

The menu choices are:

1. Print “Hello World!”

   This displays a simple message over the UART to demonstrate its functionality. This option is somewhat unnecessary because the display of the menu shows that the UART has been initialized and is working correctly.

2. Perform a DDR memory test.

   This choice executes the DDRTest() function which writes an example pattern to all the memory locations of the DDR2 memory space and then verifies that pattern via a simple read-back. If the verification of any memory location fails, then the DDRTest() function exits prematurely returning an error code.

Note: The DDRTest() function is not meant to represent a robust, valid test of memory. It only serves as an example to verify that the DDR2 memory controller has been initialized and is functioning correctly.

3. Jump to the CS2 AEMIF memory space to start execution.

   This is accomplished by simply calling the function aemif_start(). This function is prototyped in the header file as

```
void aemif_start( void )_attribute__((naked,section(".aemif")));
```

   and is declared as

```
void aemif_start(){
     asm(" NOP");
}
```

   This code says that the aemif_start() function is located in the CS2 AEMIF memory space (as per the linker script) and, like the boot() entry point function, is naked. This is because this function acts as an exit point of the application, from which there is no return. The assemply *NOP* command is inserted to make sure the function is not optimized out by the compiler. In the makefile, the objcopy command excludes this function by excluding the .aemif section from the binary output.

Note: The last menu choice requires some executable code to be present at the start of the CS2 AEMIF memory region. On the DVEVM this might be the U-boot code in the NOR flash memory.

4.3.3 Platform Initialization

This section describes the important code that is used to initialize the DM644x DMSoC when booting over the UART. All of this code can be found in the *dm644x.c* file. The initialization requires reading and writing a large number of hardware registers. The addresses of these registers are determined by structures and preprocessor defines in the *dm644x.h* header file. The header file references the associated peripheral and subsystem documentation if more information is desired.

The PlatformInit() function, called by the uartapp_main() function, in turn calls five other initialization functions in order: UARTInit(), PLL1Init(), PLL2Init(), DDR2Init(), and AEMIFInit(). The UARTInit() command is used to set up the UART0 and TIMER0 peripherals. The TIMER0 peripheral is used by the UART commands as a timeout timer. PLL1Init() is used to change the ARM core clock from the slow 13.5 MHz to the fast 297 MHz by changing the PLL1 multiplier. PLL2Init() and DDR2Init() are used to configure the DDR2 memory system. Finally, the AEMIFInit() function is used to set some timing registers for the AEMIF interface to the worst-case settings.

In this section, only the initializations for the DDR2 memory, found in functions PLL2Init() and DDR2Init(), and the UART, found in UARTInit(), are discussed. The PLL1 Initialization is similar to that used for the PLL2. The AEMIF initialization does not require elaboration.

4.3.3.1 DDR2 Memory Initialization

The steps needed to initialize the DDR2 memory are listed in the *TMS320DM644x DMSoC DDR2 Memory Controller User's Guide* (SPRUE22). This section clarifies those listed instructions by providing and discussing a functioning code example.

4.3.3.1.1 PLL2 Peripheral Initialization

The first step to enabling the DDR2 memory system is to set up the DDR2 memory clock that is generated by the PLL2 peripheral. Details of how to do this are given in the PLL Controller section of the *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14). The first step is to make sure that the PLL is receiving the system clock from the proper source by setting or clearing bit 8 of the PLL2 control register.

```
PLL2->PLLCTL &= (~0x00000100);
```

The PLL2 must be put into bypass mode by clearing bit 5 and bit 0 of the PLL control register. Bypass mode sends the input reference clock to the connected subsystems and disconnects the PLL's VCO output from those same systems (i.e. the PLL is bypassed). After the PLL2 is put into bypass mode, a short amount of time is given to ensure that the switch is complete.

```
PLL2->PLLCTL &= (~0x00000021);
waitloop(32*11);
```

Next, the PLL2 is put into reset, disabled, powered-up, and then re-enabled. This leaves the peripheral device still in reset, but enabled with power.

```
PLL2->PLLCTL  &= (~0x00000008);   // Put PLL into reset
PLL2->PLLCTL  |= (0x00000010);    // Disable the PLL
PLL2->PLLCTL  &= (~0x00000002);   // Power-up the PLL
PLL2->PLLCTL  &= (~0x00000010);   // Enable the PLL
```

In this state the PLL multipliers and dividers can be set. Note that only the PLL2 peripheral has programmable dividers. The PLL2 provides the input clock to the video processing sub-system (VPSS) as well as the DDR2 memory controller. The VPSS needs a 54 MHz clock input. The DDR2 memory controller needs a clock operating at twice the physical memory rate. On the DVEVM, the memory layout is intended to work up to a rate of 166 MHz. The closest multiple of 27 MHz is 162 MHz. Therefore, the PLL2 needs to provide 324 MHz to the DDR2 memory controller.

```
PLL2->PLLM = 23;        // 27  Mhz * (23+1) = 648 MHz
PLL2->PLLDIV1 = 11;     // 648 MHz / (11+1) = 54  MHz
PLL2->PLLDIV2 = 1;      // 648 MHz / (1+1 ) = 324 MHz
                           (the PHY DDR rate)
```

Finally, the PLL dividers are enabled and the output clocks are set to undergo phase alignment to the input system clock.

```
PLL2->PLLDIV2 |= (0x00008000);   // Enable DDR divider
PLL2->PLLDIV1 |= (0x00008000);   // Enable VPBE divider
PLL2->PLLCMD |= 0x00000001;      // Tell PLL to do phase alignment
while ((PLL2->PLLSTAT) & 0x1);   // Wait until done
waitloop(256*11);                // Wait for stability
```

With the clocks stable, the PLL2 peripheral can be released from reset and switched out of bypass mode.

```
PLL2->PLLCTL |= (0x00000008);   // Take PLL out of reset
waitloop(2000*11);              // Wait for locking
PLL2->PLLCTL |= (0x00000001);   // Switch out of bypass mode
```

4.3.3.1.2 DDR Memory Controller Initialization

Once the clocks to the DDR2 memory controller peripheral are correctly set, the memory controller itself needs to be initialized. The settings for the memory controller are determined by the type and number of memory devices, as well as the board layout. To begin, the memory controller peripheral is first enabled via the on-chip power and sleep controller (PSC).

```
while (PSC->PTSTAT & 0x00000001);
PSC->MDCTL[13] = ((PSC->MDCTL[13]) & (0xFFFFFFE0)) | (0x00000003);
PSC->PTCMD |= 0x00000001;
while ((PSC->PTSTAT) & 0x00000001);
while (((PSC->MDSTAT[13]) & 0x1F) != 0x00000003);
```

Then, the timing registers need to be programmed with the proper values. The formula for calculating these values is provided in the DDR2 Memory Controller Registers section of SPRUE22. Next, a dummy write and read cycle is performed on the DDR memory space to apply the settings.

```
DDRMem[0] = DDR_TEST_PATTERN;
if (DDRMem[0] == DDR_TEST_PATTERN)
     UARTSendInt(DDRMem[0]);
```

After the timing values are set, the DDR memory controller undergoes a soft reset via the PSC.

```
PSC->MDCTL[13] = ((PSC->MDCTL[13]) & (0xFFFFFFE0)) | (0x00000001);
PSC->PTCMD |= 0x00000001;
while ((PSC->PTSTAT) & 0x00000001);
while (((PSC->MDSTAT[13]) & 0x1F) != 0x00000001);

PSC->MDCTL[13] = ((PSC->MDCTL[13]) & (0xFFFFFFE0)) | (0x00000003);
PSC->PTCMD |= 0x00000001;
while ((PSC->PTSTAT) & 0x00000001);
while (((PSC->MDSTAT[13]) & 0x1F) != 0x00000003);
```

The DDR memory controller can now begin the voltage, temperature, and process (VTP) calibration. This calibration controls the output impedance of the output IO of the DDR pins of the DM644x. This calibration is begun by clearing and then setting bit 15 of the VTPIOCR. After at least 33 VTP clock cycles, the calibration is ensured to be complete. The VTP clock is 27 MHz. The calibration data is stored in the DDRVTPR register, which can only be read by setting the enable bit in the DDRVTPER of the ARM system module. Once the data is read, it should be written back to the VTPIOCR.

```
DDR->VTPIOCR = 0x201F;                          // Clear calibration start bit
DDR->VTPIOCR = 0xA01F;                          // Set calibration start bit
waitloop(11*33);                                // Wait for calibration to complete
SYSTEM->DDRVTPER = 0x1;                         // Enable DDRVTPR access
tempVTPData = 0x3FF & DDRVTPR;                  // Read calibration data
DDR->VTPIOCR = ((DDR->VTPIOCR) & 0xFFFFFC00) | tempVTPData;
DDR->VTPIOCR = (DDR->VTPIOCR) & (~0x00002000); // Clear calibration enable bit
SYSTEM->DDRVTPER = 0x0;                         // Disable DDRVTPR access
```

At this point the DDR2 Memory subsystem is enabled and calibrated. The simple memory test in the uartapp_main() function can verify that the memory system is functioning correctly.

4.3.3.2 UART Initialization

The UART initialization is, appropriately, much simpler than that for the DDR2 memory system. The code to perform the initialization is found in the UARTInit() function. It consists of setting the UART control registers and then the TIMER0 control registers for use as a timeout timer for the UART commands. Because the UART is used to download the code when interfacing to the RBL, it is actually already initialized. But this code is included for completeness in case the user wants to change the communications parameters.

The initialization of the UART needs to be delayed a certain amount of time after the RBL passes control to the downloaded application. This time allows the final bytes of data the RBL placed in the UART's FIFO buffer to be successfully sent. To delay the necessary amount of time, the transmitter empty bit of the UART line status register (LSR) is polled until it indicates that the transmit shift register and transmitter FIFO are both empty.

```
while((UART0->LSR & 0x40) == 0  );
```

The remainder of the UARTInit() function is used to set the appropriate values in the UART configuration registers.

```
SYSTEM->PINMUX[1] |=  1;        // Make sure the pins are in UART mode
UART0->LCR |= 0x80;             // Set DLAB bit to allow setting clock dividers
UART0->DLL = 0x0F;              // Set dividers for the UART baud rate
UART0->DLH = 0x00;
UART0->FCR = 0x07;              // Enable, clear, and reset FIFOs
UART0->MCR = 0x00;              // Disable autoflow control
UART0->PWREMU_MGNT |= 0x8001;   // Enable RX, TX and set to run
UART0->LCR = 0x03;              // Set word length to 8-bits, clear DLAB bit
```

After the UART0 peripheral is set up, the TIMER0 mode and period is set. During the UART send and receive commands, the timer is enabled and its IRQ bit is polled to check if the counter has reached the specified period count. The timer period register holds the number of cycles that pass until an interrupt is thrown. The TIMER0 peripheral is clocked from the 27-MHz input reference clock.

```
TIMER0->TCR = 0x00000000;       // Disable the timer
TIMER0->TGCR = 0x00000003;      // Set to 64-bit GP Timer mode, enable TIMER12 & 34
TIMER0->TIM34 = 0x00000000;     // Reset timer count to zero
TIMER0->TIM12 = 0x00000000;     // Reset timer count to zero
TIMER0->PRD34 = 0x00000000;     // Set timer period registers
TIMER0->PRD12 = 0x080BEFC0;     // 0x080BEFC0 = 5*27x10^6 => 5 seconds
```

4.3.4 UART Commands

This sections details the basic functionality of the UART send and receive code. Details about the registers used in these function can be found in the *TMS320DM644x DMSoC Universal Asynchronous Receiver/Transmitter (UART) User's Guide* (SPRUE33) and in the *TMS320DM644x DMSoC 64-Bit Timer User's Guide* (SPRUE26). The UART communications commands consist of four functions:

- UARTRecvData() – This function is used to receive a specified number of bytes.
- UARTSendData() – This function is used to send a null-terminated string.
- UARTSendInt() – This function is used the ASCII hexadecimal representation of an integer.
- GetStringLen() – This function is used to determine the string length for UARTSendData().

In addition there are two simple inline function (found in dm644x.h) that are used to start the timeout timer and then check its status:

- TIMER0Start() – Clear the timer status, reset the counter, and enable counting.
- TIMER0Status() – Check if the TIMER0 bit of the interrupt controller IRQ1 register is set.

4.3.4.1 UARTRecvData()

The receive function requires the caller to pass the number of bytes that the function should wait to receive. The function loops this number of times, waiting for a byte to arrive or a timeout to occur.

```
TIMER0Start();                                  // Start the timeout timer
do{
     status = (UART0->LSR)&(0x01);              // Check the data-ready bit of LSR
     timerStatus = TIMER0Status();              // Check timout status
} while (!status && timerStatus);               // loop until timout or data ready
```

If a timeout has occurred, then the function should terminate early, returning a timeout error code.

```
if(timerStatus == 0)
     return E_TIMEOUT;
```

Otherwise, a byte has been received and is ready to be read at the UART receive register.

```
seq[i] = UART0->RBR;
```

Finally, this character is echoed back to the host. This step is not required, but does improve the usability for terminal programs without local echo.

```
UART0->THR = seq[i];
do{
     status = (UART0->LSR)&(0x20);             // check transmit register empty bit
} while (!status);
```

4.3.4.2 UARTSendData()

To send data, a pointer to the start of a byte sequence is passed to the UARTSendData() function. Then the function must get the length of the string by calling the GetStringLen() function. The returned value tells the function how many times to loop.

```
numBytes = GetStringLen(seq);
```

Then the function loops numBytes times sending a byte each loop iteration, unless a timeout occurs.

```
TIMER0Start();                                  // Start the timeout timer
do{
     status = (UART0->LSR)&(0x20);              // check transmit register empty bit
     timerStatus = TIMER0Status();              // Check timout status
} while (!status && timerStatus);               // loop until timout or THR empty
```

As with the receive function, if a timeout occurs the function terminates returning a timeout error code.

```
if(timerStatus == 0)
     return E_TIMEOUT;
```

If no timeout occurs, then the UART transmit hold register is empty and can accept the next data byte from our sequence.

```
(UART0->THR) = seq[i];
```

4.3.4.3 UARTSendInt()

This function is useful for sending a hexadecimal text representation of an integer variable (or any variable cast to an integer). This function creates a nine byte, null terminated sequence by examining the integer in 4-bit chunks. Each 4-bit chunk is converted to the appropriate numeral or A-F letter.

```
for( i = 0; i < 8; i++){
     shift = ((7-i)*4);                          // Most sig 4-bit to least sig 4-bit
     temp = ((value>>shift) & (0x0000000F));     // Value of 4-bit chunk
     if (temp > 9){                              // If it should be in A-F
          temp = temp + 55;
     }
     else{                                       // If it should be a numeral
          temp = temp + 48;
     }
     seq[i] = temp;                              // Assign the chractder to seq.
}
seq[8] = 0;                                      // Null-terminate the sequence
```

Finally, the sequence of characters is passed to the UARTSendData() function for transmission.

```
return UARTSendData(seq);
```

4.3.4.4 GetStringLen()

This function helps the send function know how many bytes to send. It returns a zero if no termination is found within a maximum number of bytes. Otherwise, the number of bytes before the null-termination is received.

```
while ((seq[i] != 0) && (i<MAXSTRLEN)){ i++;}
if (i == MAXSTRLEN)
     return 0;
else
     return i;
```

The MAXSTRLEN value is a preprocessor macro defined in the *uart.h* file as 256.

5 References

- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (SPRUE14)
- *TMS320DM644x DMSoC DDR2 Memory Controller User's Guide* (SPRUE22)
- *TMS320DM644x DMSoC Universal Asynchronous Receiver/Transmitter (UART) User's Guide* (SPRUE33)
- *TMS320DM644x DMSoC 64-Bit Timer User's Guide* (SPRUE26)
- GNU Linker Manual, http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html.
- GNU Binutils Manual, http://www.gnu.org/software/binutils/manual/html_chapter/binutils_toc.html.
- GNU gcc Manual, http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/.
- *A Painless Guide to CRC Error Detection Algorithms*, Ross Williams, 1993, http://www.ross.net/crc/download/crc_v3.txt.

Appendix A CRC32 Lookup Table

This appendix includes the values found in Table A-1 that are sent to the RBL in the application download process.

A.1 CRC32 Lookup Table

Table A-1. Lookup Table for CRC32

No.	Value	No.	Value	No.	Value	No.	Value
0	0x00000000	1	0x77073096	2	0xC30C8EA1	3	0x990951BA
4	0x076DC419	5	0x706AF48F	6	0xE963A535	7	0x9E6495A3
8	0x0EDB8832	9	0x79DCB8A4	10	0xE0D5E91E	11	0x97D2D988
12	0x09B64C2B	13	0x7EB17CBD	14	0xE7B82D07	15	0x90BF1D91
16	0x1DB71064	17	0x6AB020F2	18	0xF3B97148	19	0x84BE41DE
20	0x1ADAD47D	21	0x6DDDE4EB	22	0xF4D4B551	23	0x83D385C7
24	0x136C9856	25	0x646BA8C0	26	0xFD62F97A	27	0x8A65C9EC
28	0x14015C4F	29	0x63066CD9	30	0xFA0F3D63	31	0x8D080DF5
32	0x3B6E20C8	33	0x4C69105E	34	0xD56041E4	35	0xA2677172
36	0x3C03E4D1	37	0x4B04D447	38	0xD20D85FD	39	0xA50AB56B
40	0x35B5A8FA	41	0x42B2986C	42	0xDBBBC9D6	43	0xACBCF940
44	0x32D86CE3	45	0x45DF5C75	46	0xDCD60DCF	47	0xABD13D59
48	0x26D930AC	49	0x51DE003A	50	0xC8D75180	51	0xBFD06116
52	0x21B4F4B5	53	0x56B3C423	54	0xCFBA9599	55	0xB8BDA50F
56	0x2802B89E	57	0x5F058808	58	0xC60CD9B2	59	0xB10BE924
60	0x2F6F7C87	61	0x58684C11	62	0xC1611DAB	63	0xB6662D3D
64	0x76DC4190	65	0x01DB7106	66	0x98D220BC	67	0xEFD5102A
68	0x71B18589	69	0x06B6B51F	70	0x9FBFE4A5	71	0xE8B8D433
72	0x7807C9A2	73	0x0F00F934	74	0x9609A88E	75	0xE10E9818
76	0x7F6A0DBB	77	0x086D3D2D	78	0x91646C97	79	0xE6635C01
80	0x6B6B51F4	81	0x1C6C6162	82	0x856530D8	83	0xF262004E
84	0x6C0695ED	85	0x1B01A57B	86	0x8208F4C1	87	0xF50FC457
88	0x65B0D9C6	89	0x12B7E950	90	0x8BBEB8EA	91	0xFCB9887C
92	0x62DD1DDF	93	0x15DA2D49	94	0x8CD37CF3	95	0xFBD44C65
86	0x4DB26158	97	0x3AB551CE	98	0xA3BC0074	99	0xD4BB30E2
100	0x4ADFA541	101	0x3DD895D7	102	0xA4D1C46D	103	0xD3D6F4FB
104	0x4369E96A	105	0x346ED9FC	106	0xAD678846	107	0xDA60B8D0
108	0x44042D73	109	0x33031DE5	110	0xAA0A4C5F	111	0xDD0D7CC9
112	0x5005713C	113	0x270241AA	114	0xBE0B1010	115	0xC90C2086
116	0x5768B525	117	0x206F85B3	118	0xB966D409	119	0xCE61E49F
120	0x5EDEF90E	121	0x29D9C998	122	0xB0D09822	123	0xC7D7A8B4
124	0x59B33D17	125	0x2EB40D81	126	0xB7BD5C3B	127	0xC0BA6CAD
128	0xEDB88320	129	0x9ABFB3B6	130	0x03B6E20C	131	0x74B1D29A
132	0xEAD54739	133	0x9DD277AF	134	0x04DB2615	135	0x73DC1683
136	0xEAD54739	137	0x94643B84	138	0x0D6D6A3E	139	0x7A6A5AA8
140	0xE40ECF0B	141	0x9309FF9D	142	0x0A00AE27	143	0x7D079EB1
144	0xF00F9344	145	0x8708A3D2	146	0x1E01F268	147	0x6906C2FE

Table A-1. Lookup Table for CRC32 (continued)

No.	Value	No.	Value	No.	Value	No.	Value
148	0xF762575D	149	0x806567CB	150	0x196C3671	151	0x6E6B06E7
152	0xFED41B76	153	0x89D32BE0	154	0x10DA7A5A	155	0x67DD4ACC
156	0xF9B9DF6F	157	0x8EBEEFF9	158	0x17B7BE43	159	0x60B08ED5
160	0xD6D6A3E8	161	0xA1D1937E	162	0x38D8C2C4	163	0x4FDFF252
164	0xD1BB67F1	165	0xA6BC5767	166	0x3FB506DD	167	0x48B2364B
168	0xD80D2BDA	169	0xAF0A1B4C	170	0x36034AF6	171	0x41047A60
172	0xDF60EFC3	173	0xA867DF55	174	0x316E8EEF	175	0x4669BE79
176	0xDF60EFC3	177	0xBC66831A	178	0x256FD2A0	179	0x5268E236
180	0xCC0C7795	181	0xBB0B4703	182	0x256FD2A0	183	0x5505262F
184	0xC5BA3BBE	185	0xB2BD0B28	186	0x2BB45A92	187	0x5CB36A04
188	0xC2D7FFA7	189	0xB5D0CF31	190	0x2CD99E8B	191	0x5BDEAE1D
192	0x9B64C2B0	193	0xEC63F226	194	0x756AA39C	195	0x026D930A
196	0x9C0906A9	197	0xEB0E363F	198	0x72076785	199	0x05005713
200	0x95BF4A82	201	0xE2B87A14	202	0x7BB12BAE	203	0x0CB61B38
204	0x92D28E9B	205	0xE5D5BE0D	206	0x7CDCEFB7	207	0x0BDBDF21
208	0x86D3D2D4	209	0xE5D5BE0D	210	0x68DDB3F8	211	0x1FDA836E
212	0x81BE16CD	213	0xF6B9265B	214	0x6FB077E1	215	0x18B74777
216	0x88085AE6	217	0xFF0F6A70	218	0x66063BCA	219	0x11010B5C
220	0x8F659EFF	221	0xF862AE69	222	0x616BFFD3	223	0x166CCF45
224	0xA00AE278	225	0xD70DD2EE	226	0x4E048354	227	0x3903B3C2
228	0xA7672661	229	0xD06016F7	230	0x4969474D	231	0x3E6E77DB
232	0xAED16A4A	233	0xD9D65ADC	234	0x40DF0B66	235	0x37D83BF0
236	0xA9BCAE53	237	0xDEBB9EC5	238	0x47B2CF7F	239	0x30B5FFE9
240	0xBDBDF21C	241	0xCABAC28A	242	0x53B39330	243	0x24B4A3A6
244	0xBAD03605	245	0xCDD70693	246	0x54DE5729	247	0x23D967BF
248	0xB3667A2E	249	0xC4614AB8	250	0x5D681B02	251	0x2A6F2B94
252	0xB40BBE37	253	0xC30C8EA1	254	0x5A05DF1B	255	0x2D02EF8D

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265