TMS320C6000
Network Developer’s Kit (NDK) Support Package for EVMDM642

User's Guide

Literature Number: SPRUES5A
January 2007–Revised June 2008

Contents

List of Figures

List of Tables

Read This First

About This Manual

This document contains information about the Network Developer's Kit (NDK) Support Package for EVMDM642. The package includes source code for HAL drivers, and examples to reuse or modify them for customer designed platforms. Pre-built HAL libraries are also delivered with the package.

How to Use This Manual

This document is divided into the following sections:

- **Section 1 – Getting Started:** Introduces the NDK Support Package, which is designed to run the NDK on EVMDM642 platform.
- **Section 2 – User LED Driver:** Describes the user LED driver for EVMDM642.
- **Section 3 – Timer Driver:** Describes the timer driver for EVMDM642.
- **Section 4 – Serial Driver:** Describes the serial driver for TL 16C752.
- **Section 5 – Ethernet Driver:** Describes the EMAC driver for DM642.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface.
- In syntax descriptions, the function or macro appears in a bold typeface and the parameters appear in plainface within parentheses. Portions of a syntax that are in bold should be entered as shown; portions of syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.

Related Documentation from Texas Instruments

The following books describe the TMS320C6x™ devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

SPRU189 — ***TMS320C6000 DSP CPU and Instruction Set Reference Guide.*** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000™ digital signal processors (DSPs).

SPRU190 — ***TMS320C6000DSP Peripherals Overview Reference Guide.*** Provides an overview and briefly describes the peripherals available on the TMS320C6000™ family of digital signal processors (DSPs).

SPRU197 — ***TMS320C6000 Technical Brief.*** Provides an introduction to the TMS320C62x™ and TMS320C67x™ digital signal processors (DSPs) of the TMS320C6000™ DSP family. Describes the CPU architecture, peripherals, development tools and third-party support for the C62x™ and C67x™ DSPs.

SPRU198 — ***TMS320C6000 Programmer's Guide.*** Reference for programming the TMS320C6000™ digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x™ DSP.

SPRU509 — *TMS320C6000 Code Composer Studio Development Tools v3.3 Getting Started Guide* introduces some of the basic features and functionalities in Code Composer Studio™ to enable you to create and build simple projects.

SPRU523 — *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide.* Describes how to use the NDK libraries, how to develop networking applications on TMS320C6000™ platforms, and ways to tune the NDK to fit a particular software environment.

SPRU524 — *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide.* Describes the various API functions provided by the stack libraries, including the low level hardware APIs.

Trademarks

TMS320C6x, TMS320C6000, TMS320C62x, TMS320C67x, C62x, C67x, C64x, Code Composer Studio, DSP/BIOS are trademarks of Texas Instruments.

TMS320C6000 Network Developer's Kit (NDK) Support Package for EVMDM642

1 Getting Started

This section introduces the NDK Support Package for EVMDM642.

1.1 Introduction

The TMS320C6000 NDK Support Package for EVMDM642 includes:

- Source codes and pre-built libraries for the NDK Hardware Adaptation Layer (HAL) drivers
- NDK examples

There are four basic HAL drivers required to operate the NDK: timer, user LED, serial port, and Ethernet. The Support Package provides Ethernet, serial, and user LED drivers specific to EVMDM642 platform. The timer driver is implemented by using DSP/BIOS™ PRD module from the NDK.

1.2 Installing the Support Package

1. Install the Code Composer Studio using the Code Composer Studio CD, if not installed yet. It must be installed before the NDK Support Package software. Be sure that you install a version of Code Composer Studio 3.3 or higher.
2. Install the NDK software, if not installed yet.
3. Copy the *ti.ndk.platforms.evmdm642.tar* under <NDK_INSTALL_DIR>\packages directory, and untar it. The support package installs over the NDK installation. Once installed, the following directories are created under the <NDK_INSTALL_DIR>/packages/ti/ndk directory:
   - <docs/evmdm642> Documentation files for Support Package
   - <example/network/cfgdemo/evmdm642> Code Composer Studio project files for cfgdemo example
   - <example/network/client/evmdm642> Code Composer Studio project files for client example
   - <example/network/helloWorld/evmdm642> Code Composer Studio project files for helloWorld example
   - <example/serial/client/evmdm642> Code Composer Studio project files for serial client example
   - <example/serial/router/evmdm642> Code Composer Studio project files for serial router example
   - <example/tools/evmdm642> Common tools used by Support Package
   - <lib/hal/evmdm642> Pre-built HAL libraries for EVMDM642
   - <src/hal/evmdm642/eth_dm642> Source code for DM642 EMAC driver
   - <src/hal/evmdm642/ser_ti752> Source code of TI 16C752 Dual serial universal asynchronous receiver/transmitter (UART) driver
   - <src/hal/evmdm642/userled_dm642> Source code for EVMDM642 LED driver
4. The network management interface unit (NIMU) is introduced in NDK 1.94.00.001 as an alternative to LL packet architecture to add support for multiple instances of drivers. Several builds of the Ethernet driver libraries are provided with NIMU and LL architectures for Big and Little Endian architectures under <lib/hal/evmdm642>. The libraries are named as <lib_name>.lib for little endian and <lib_name>e.lib for big endian. The libraries built using NIMU are named as <lib_name>_nimu.lib and the ones built using LL architecture are named as <lib_name>_ll.lib.
   Please choose the most appropriate version of the library based on the endianness and whether NIMU/LL is required and rename the library to just <lib_name>.lib for using it with any example projects.
   For example, If an application requires NIMU little endian HAL ethernet driver, rename <lib/hal/evmdm642/hal_eth_dm642_nimu.lib> to <lib/hal/evmdm642/hal_eth_dm642.lib>.

5. Also, choose the appropriate version of the user LED and serial drivers (if required) and rename it, based on endianness. The little endian version of the library is named as <lib_name>.lib and the big endian counterpart is named as <lib_name>e.lib.
   For example, if the target board is big-endian platform then:
   copy <lib/hal/evmdm642/hal_userled_dm642e.lib> to
   <lib/hal/evmdm642/hal_userled_dm642.lib>
   And if a serial port is required: copy <lib/hal/evmdm642/hal_ser_ti752e.lib> to
   <lib/hal/evmdm642/hal_ser_ti752.lib>.
6. Copy the correct NDK libraries for testing based on endianness and NIMU/LL architectures.
   For example, to compile your application with NIMU for a little endian target, change directory to <NDK_INSTALL_DIR>/packages/ti/ndk/lib/C64plus> and execute the following commands at the command prompt:
   copy netctrl_nimu.lib netctrl.lib
   copy nettool_nimu.lib nettool.lib
   copy all_stk\stk_nimu.lib stack.lib
   For more details on choosing and building in appropriate NDK libraries, see the *NDK Libraries* and *Building in NIMU* sections in *TMS320C6000 Network Developer's Kit (NDK) User's Reference Guide* (SPRU523).
7. Configure the environment variables; for DM642, the following variables should be configured:
   NDK_INSTALL_DIR = <ndk_1_94 installation directory>
8. Make sure the PC is connected through USB/JTAG to the DM642 correctly; in Figure 1, the BlackHawk USB560M debugger is used. Install any JTAG drivers required and be sure that PIN1 of the JTAG on DM642 board matches PIN1 of the JTAG connector. Click on the Code Composer Studio Icon, typically placed on the Desktop of the PC, to open Code Composer Studio as shown below.

9. The following screen pops up.

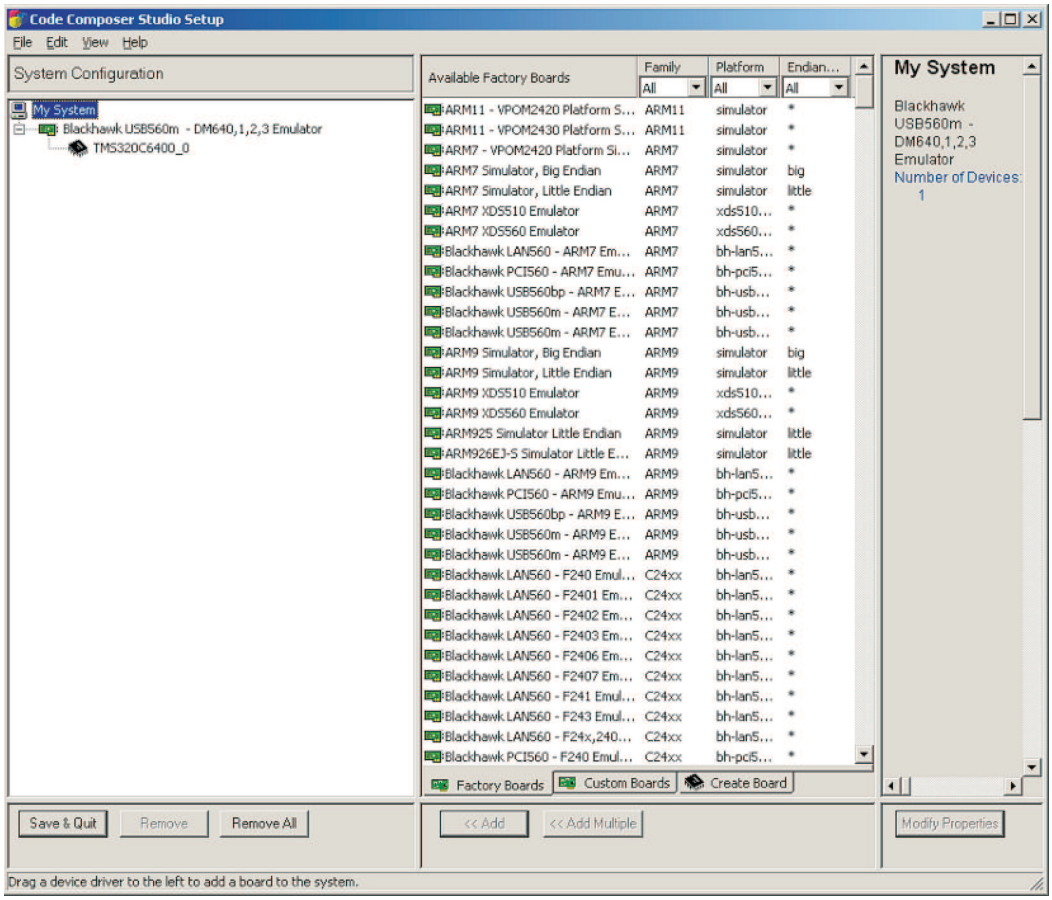

Figure 1. BlackHawk560M Configuration to Connect to DM642

10. Make sure that the Blackhawk USB560M – DM640,1,2,3 emulator is selected. Remove any other existing platforms, if any, already configured. Click on Save & Quit to start Code Composer Studio.

11. Ensure the DM642 GEL file is loaded, once in Code Composer Studio. Click on File → Load Gel. Select the GEL file from the location where the DM642 BSL package was installed (see Figure 2). (This step is done while installing the DM642 support; there is typically a CD provided along with each platform).

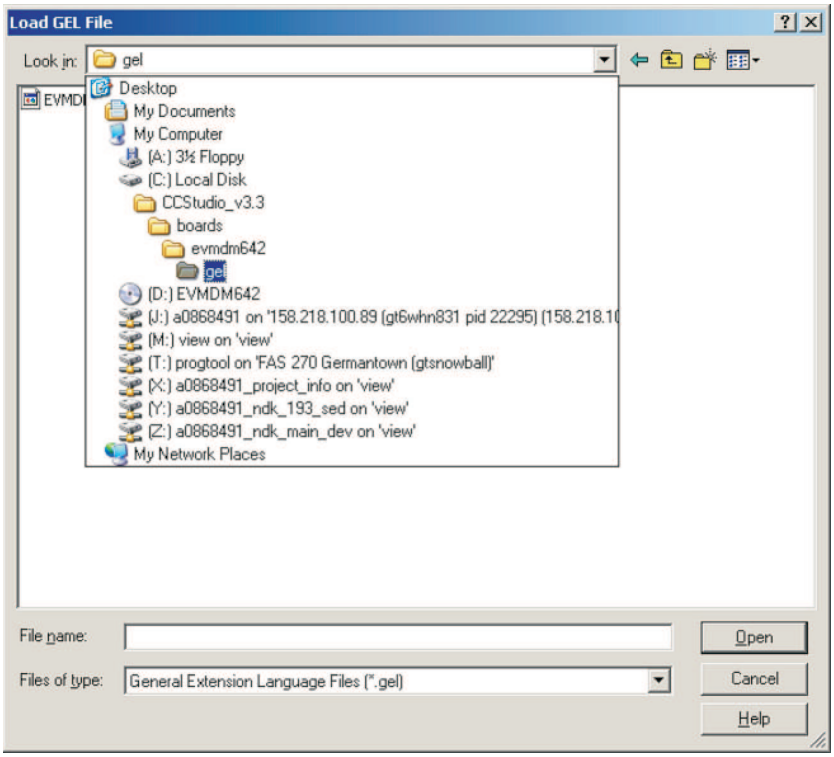

Figure 2. DM642 Gel Load

12. Select the EVMDM642.gel and click Open. The GEL file is downloaded and the board is ready to use. Steps 11 and 12 need to be done every time the DM642 board is powered off and turned back on.

13. Click on Project → Open, once Code Composer Studio is open and ready to use. Browse to <NDK_INSTALL_DIR>/packages/ti/ndk/example/network/helloWorld/evmdm642> and choose the helloWorld.pjt. It should open up the project as shown in Figure 3:

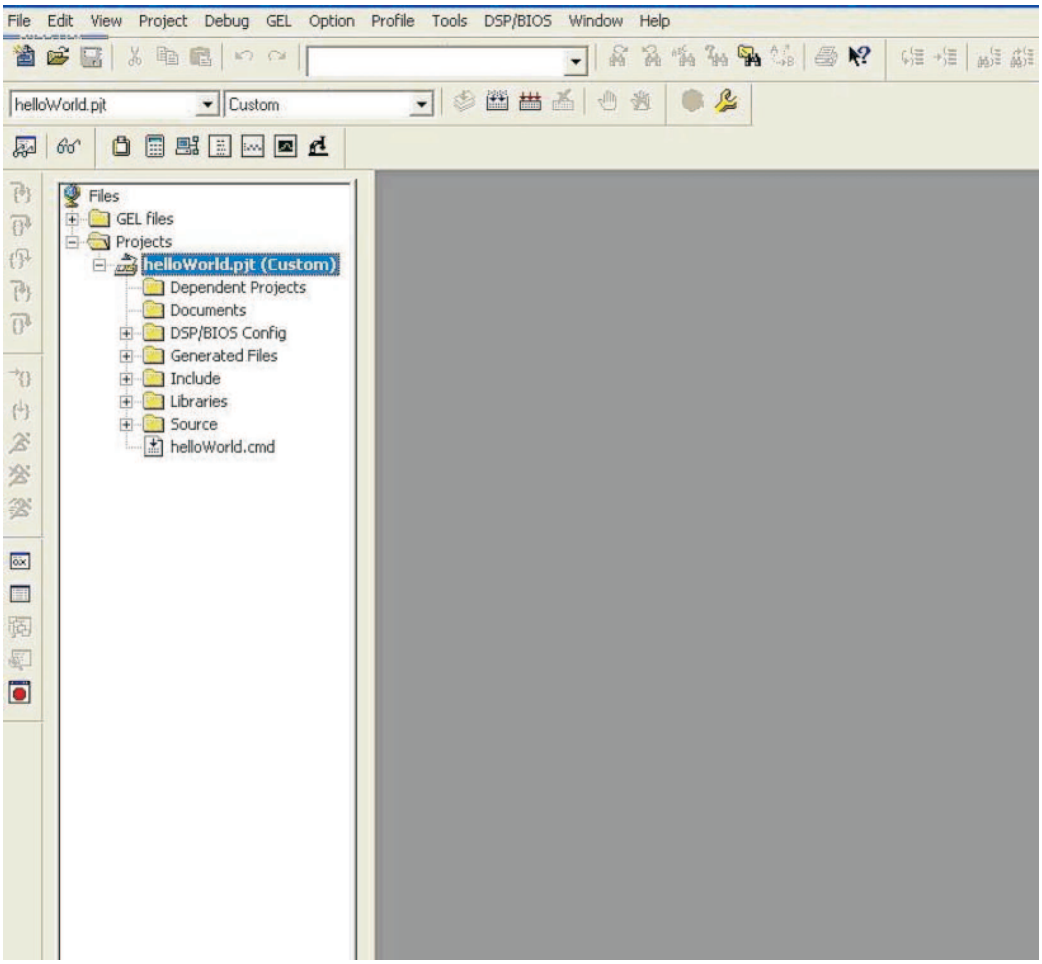

Figure 3. HelloWorld Project Open in Code Composer Studio Window

14. Follow these steps to build the example project with NIMU enabled using Code Composer Studio IDE :
    a. Open the application project in Code Composer Studio IDE. Go to Project → Build Options and click on it as shown in Figure 4:

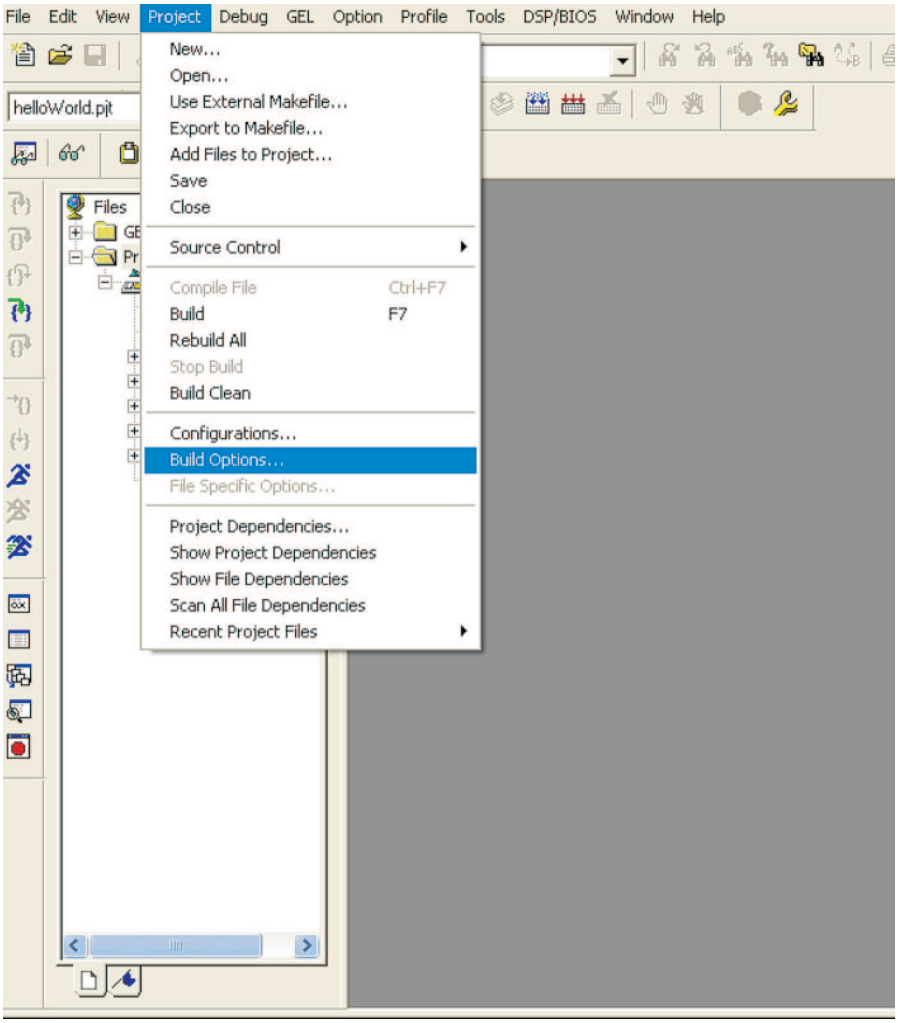

Figure 4. Configuring Build Options Through Code Composer Studio

2. Click on *Preprocessor* in the *Category* box of the *Compiler* tab on the Build Options dialog box that appears as shown in Figure 5:

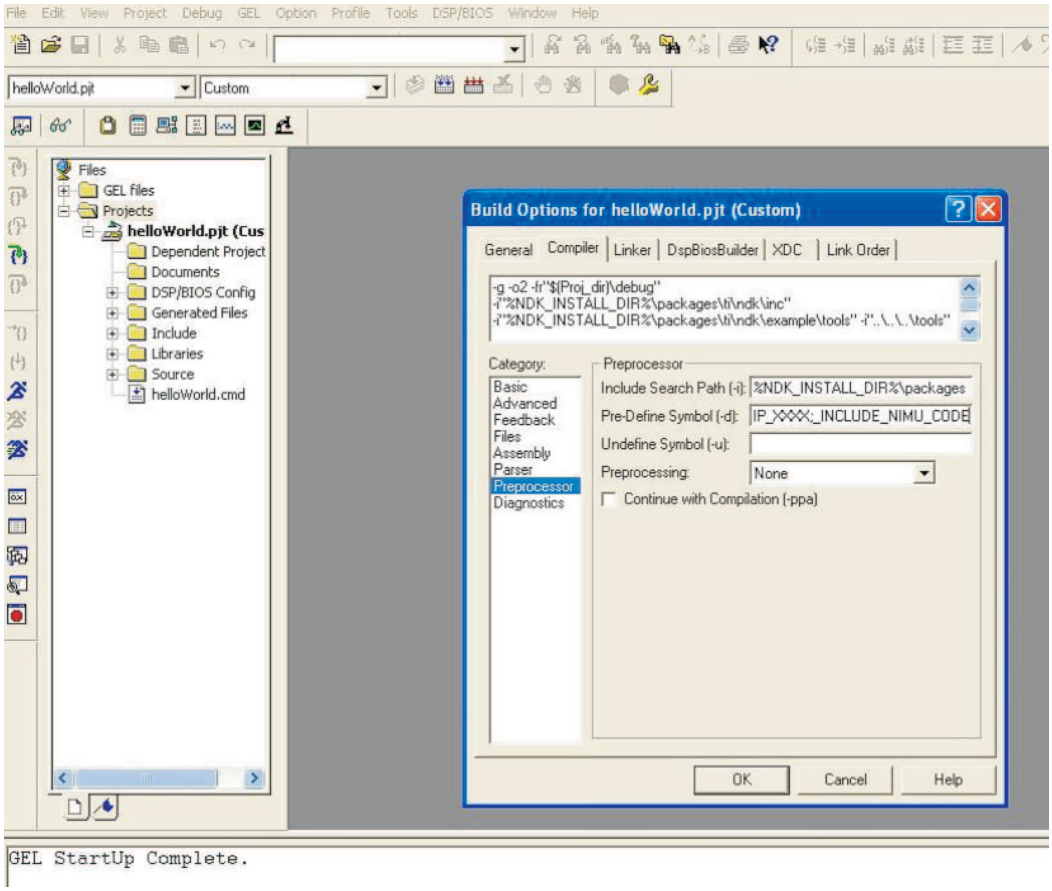

Figure 5. Enabling NIMU for Build Through Code Composer Studio

c. Add the following constant at the end and click on *OK* button to apply the settings in *Pre-Define Symbol (-d)* box as shown in Figure 5.
_INCLUDE_NIMU_CODE;

d. Build the application project according to the following the steps to make sure the NIMU is built into the final executable.

15. Connect to DM642 by entering *Alt + C* or by clicking on Debug → Connect if not already connected to the debugger on the DM642, as shown in Figure 6.

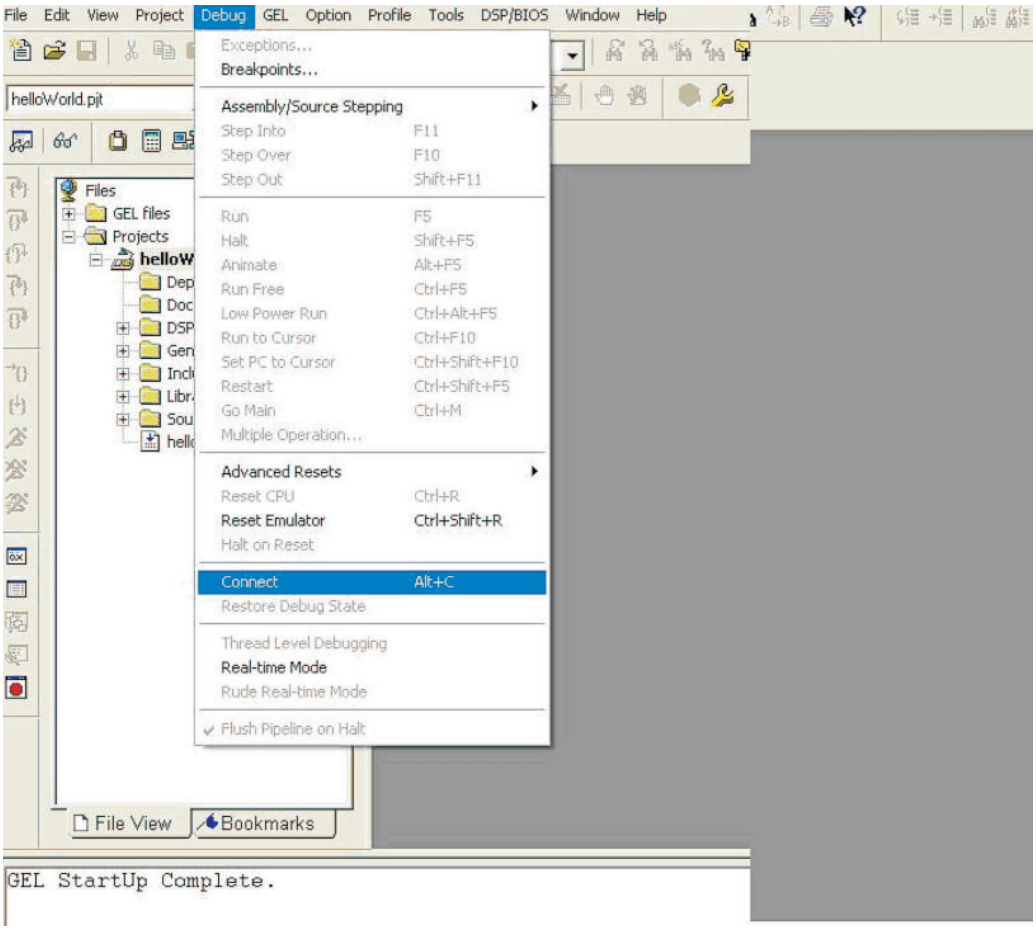

Figure 6. DM642 Connect Procedure

16. Once connected, Figure 7 shows up displaying that the Code Composer Studio debugger is now connected to the target.

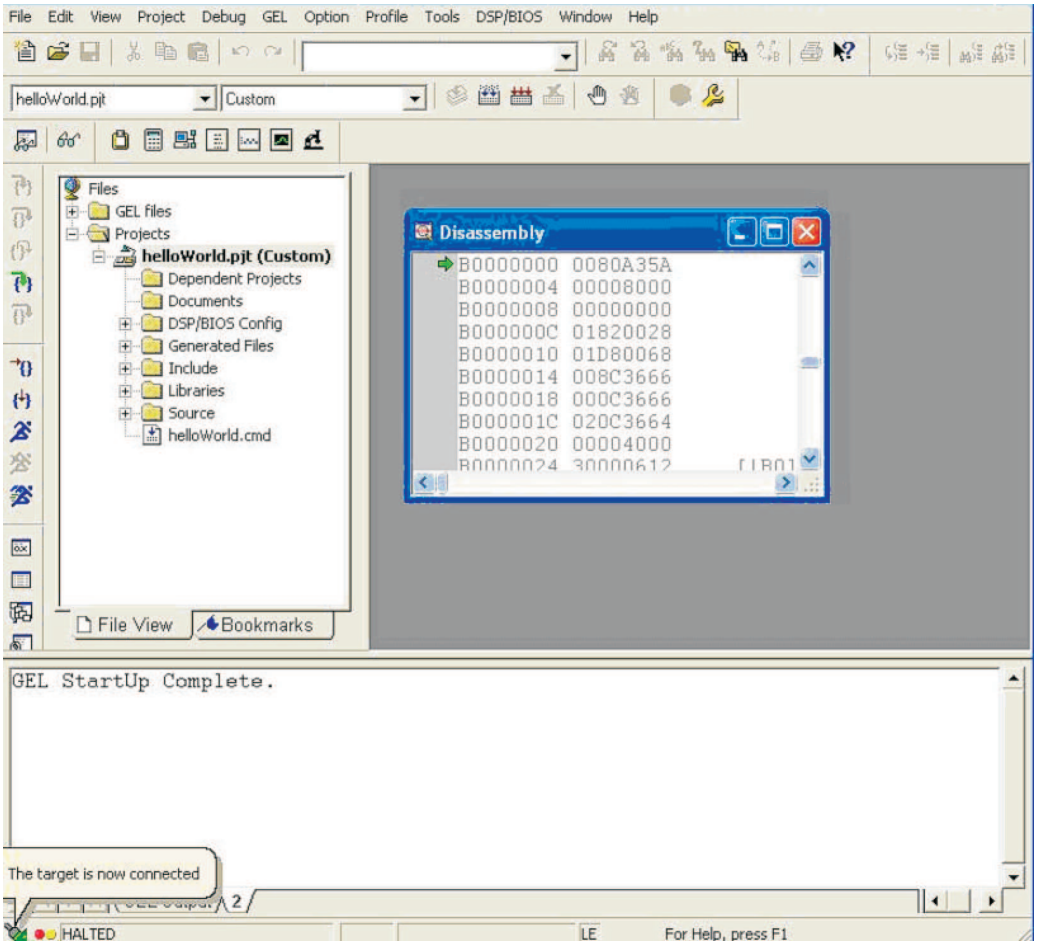

Figure 7. DM642 Connect

17. Build the project by selecting *F7* or go to Project → Build and click on it. Once the project is built without any errors, load the project onto the DM642 platform by clicking on File → Load Program and browse to the bin folder of the helloWorld project; select helloWorld.out. Enter *F5* or click on Debug → Run to start the helloWorld application on the target as shown in Figure 8

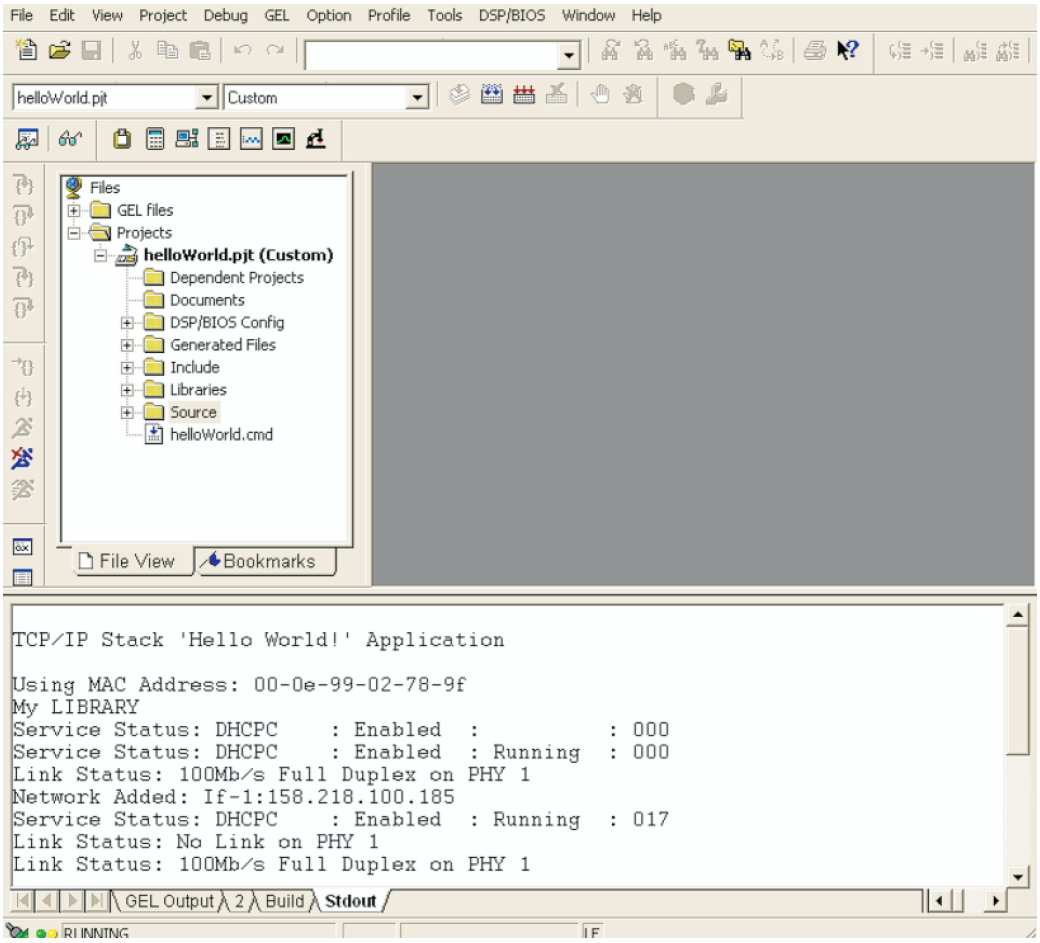

Figure 8. helloWorld Project Running Successfully on DM642

1.3 Rebuilding HAL Libraries

Code Composer Studio is the recommended tool to rebuild the HAL libraries, if needed. To build any specific HAL library from Code Composer Studio, add the relevant HAL library sources to the Code Composer Studio application project and recompile as illustrated in the steps below:

1. Right click on the project file in the Project window, as shown in Figure 9, and select *Add Files to Project* to add any HAL library sources to Code Composer Studio project. This opens up a browse window; you can browse to the required HAL sources directory and select the appropriate source code files to add them.

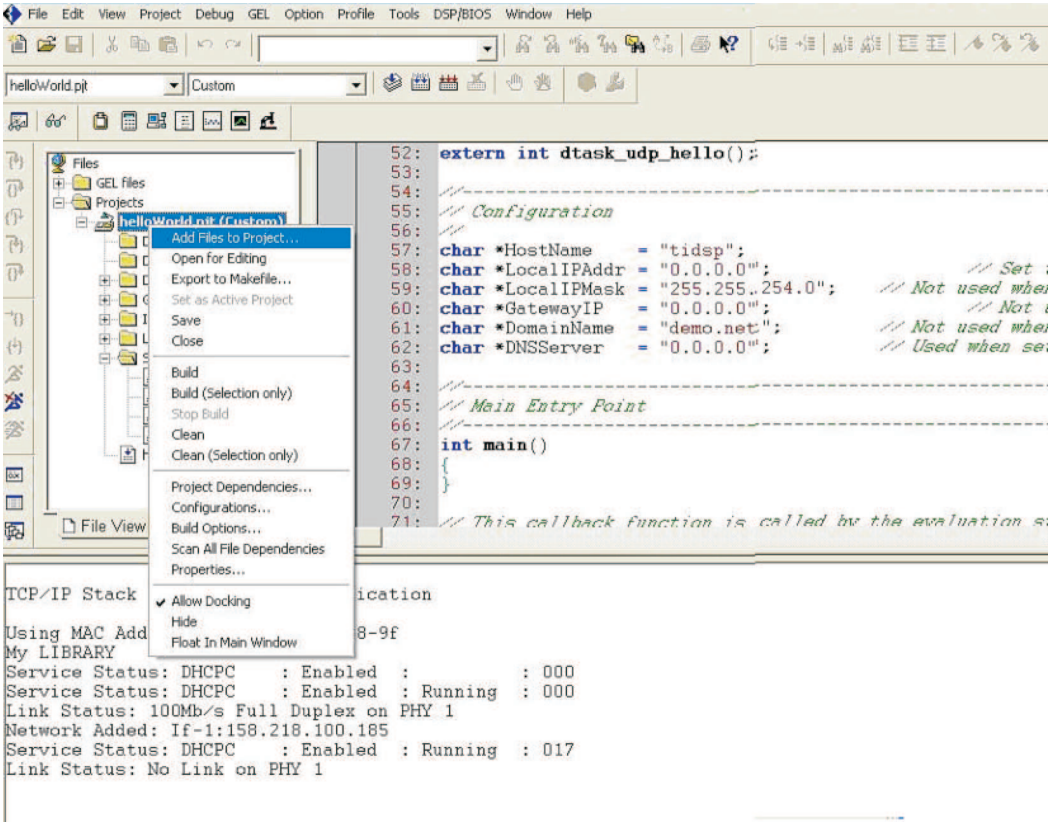

Figure 9. Source File Addition to Existing Project

For example, to add the Ethernet driver source files, browse to
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/eth_dm642>
and select the required .c files as shown in Figure 10.

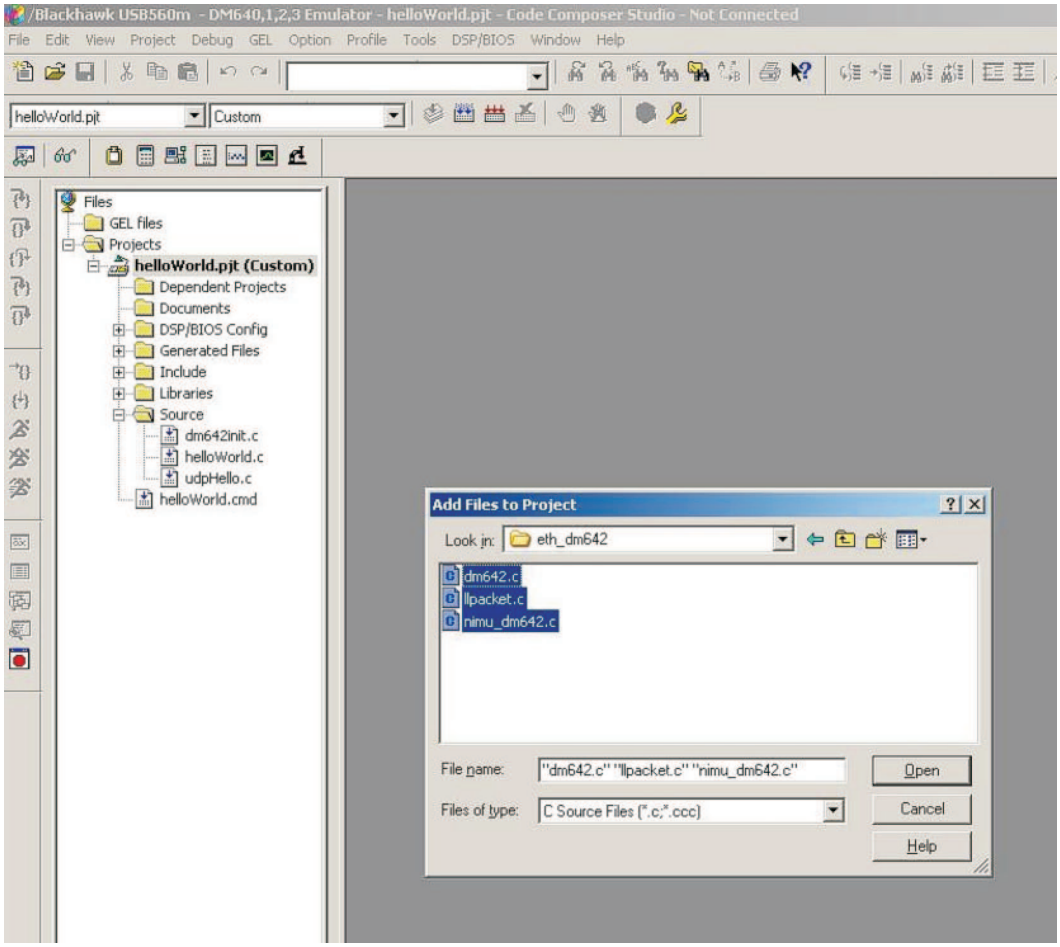

Figure 10. Ethernet HAL Build From Sources

To rebuild the DM642 Ethernet driver from sources ensure the following files are added to project:
With NIMU add:
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/eth_dm642/dm642.c>
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/eth_dm642/nimu_dm642.c>
With LL add:
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/eth_dm642/dm642.c>
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/eth_dm642/llpacket.c>
To rebuild the DM642 user LED driver from sources ensure the following file is added to project:
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/userled_dm642/llled.c>
To rebuild DM642 serial URT driver from sources ensure the following files are added to project:
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/ser_ti752/ti752.c>
<NDK_INSTALL_DIR>/packages/ti/ndk/src/hal/evmdm642/ser_ti752/lllserial.c>

For further explanation on the user LED, serial, and Ethernet driver files and their contents, please refer to Section 2, Section 4 and Section 5 of this document, respectively.

2. Rebuild the project once the necessary HAL library source files have been added to the project.

1.4 Required Terms and Concepts

To port the NDK Support Package device drivers, you should be familiar with the following concepts.

1.4.1 HAL Driver Source Files

Section 1.3 described how to build different HAL drivers for EVMDM642.

1.4.2 Network Control Module (NETCTRL)

The network control module (NETCTRL) is at the center of the NDK and controls the interface of the HAL device drivers to the internal stack functions.

The NETCTRL module and its related APIs are described in both the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524) and the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523). To write device drivers, you must be familiar with NETCTRL. The description given in the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523) is more appropriate for device driver work.

1.4.3 Stack Event (STKEVENT) Object

The STKEVENT event object is a central component to the low-level architecture. It ties the HAL layer to the scheduler thread in the network control module (NETCTRL). The network scheduler thread waits on events from various device drivers in the system, including the Ethernet, serial, and timer drivers.

The STKEVENT object is used by the device drivers to inform the scheduler that an event has occurred. The STKEVENT object and its related API are described in the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). To write device drivers, you must be familiar with STKEVENT.

1.4.4 Packet Buffer (PBM) Object

The PBM object is a packet buffer that is sourced and managed by the Packet Buffer Manager (PBM). The PBM is part of the OS adaptation layer. It provides packet buffers for all packet based devices in the system. Therefore, the serial port and Ethernet drivers both make use of this module.

The PBM object and its related API are described in the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). The *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523) also includes a section on adapting the PBM to a particular included software.

2 User LED Driver

This section describes the user LED software. The user LED driver is a collection of functions that turn on and off LED lights on the EVMDM642 platform. There is only one C file for the user LED: LLLED.C LED driver, located in the subdirectory SRC\HAL\EVMDM642\USERLED_DM642.

3 Timer Driver

This section discusses the software that drives event timing. The timer driver determines the timing for all time driven events in the NDK. The EVMDM642 platform uses the NDK provided *timer_bios* driver, which is implemented using a DSP/BIOS PRD object.

4 Serial Driver

This section describes the operational theory of the HDLC framing layer and low-level serial driver, including instructions on the use and porting of the device driver source code.

4.1 Introduction

The serial driver can be used in one of two ways in the NDK. First, by connecting the serial port to a pipe, the serial interface can drive a TTY command line tool for device configuration purposes. The TTY interface can look like any other socket to a socket based console program. Thus, the same console program can support both Telnet and direct serial link. More commonly, the serial driver implements a PPP device interface to a modem or a peer on the other side of the serial link.

The serial driver provided in the NDK is broken down into two parts: a device independent upper layer, and a device dependent layer. The device dependent layer is called a mini-driver because it only implements a subset of the full driver functions. The mini-driver API is documented in Section 4.3.4. The full NDK serial port driver API is documented in the *Hardware Adaptation Layer (HAL)* section of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524), and the interface to the HDLC framer is documented in the Point-to-Point Protocol. The example applications provide the source code to the HDLC framer interface.

The EXAMPLE\SERIAL directory includes example applications using the serial device.

4.1.1 Source Code

There are two types of serial modules included in the NDK: a stub driver used in a system where a serial port is not required, and a driver for the Texas Instruments TL16C750 and TL16C752 UART. The directories for the two types of device drivers are as follows:

Table 1. Serial Module Device Driver Source Files

Directory	File	Description
<SRC\HAL\EVMDM642\SER_TI752>		Source code for the Texas Instruments TL16C752 dual serial UART driver
	LLSERIAL.C	Hardware independent portion of the low-level serial port driver
	LLSERIAL.H	Private include file for the LLSERIAL drivers
	TI752.C	Serial mini-driver for TL16C752
	TI752.H	Private include file for the serial mini-driver

4.1.2 Theory of Operation

The serial port driver was designed to operate both an AT command set modem (or any serial TTY type application), and also support PPP HDLC-like framing, without the intervention of the TTY (or *character mode*) code. The driver accomplishes this dual role through the ability to open in two different modes.

On initialization, the serial driver is first opened in character mode using the *llSerialOpen()* function. This provides a channel for receiving normal TTY data. With a modem, this channel sends AT commands and gets replies.

When the modem has connected, or the TTY state machine provided by the programmer has detected the presence of HDLC, the HDLC-like framing module is opened using the *llSerialOpenHDLC()* call on the serial port.

Once open in HDLC mode, the hardware specific portion of the low-level serial driver tracks the HDLC frame delimiters and receives HDLC frames, including converted escape sequences, and validating the HDLC checksum. When data is sent in HDLC mode, the low-level serial driver must add the HDLC frame delimiter characters, use escape sequences when necessary, and calculate the outgoing HDLC checksum. While in HDLC mode, the serial device can still indicate character mode data if it is possible to detect the difference, but due to the relaxed standard in HDLC frame delimiting, this may not be practical.

4.2 Serial Port Driver

This section discusses the serial support source files, and the amount of required porting for each.

4.2.1 Important Note on Data Alignment

The NDK libraries have been built with the assumption that the IP header in a data packet is 16-bit aligned. In other words, the first byte of the IP packet (the version/length field) must start on an even 16-bit boundary. In any fixed length header protocol, this requirement can be met by backing off any odd byte header size, and adding it to the header padding specified to the stack. For Ethernet and peer to peer protocol (PPP), the only requirement is that the Ethernet or PPP packet not start on an odd byte boundary.

In addition, all drivers in the NDK are set up to have a 22 byte header. This is the header size of a PPPoE packet when sent using a 14 byte Ethernet header. When all arriving packets use the 22 byte header, it guarantees that they can be routed to any egress device with a header requirement up to that size. For serial operation, this requires that an HDLC packet has 18 bytes of pre-pad to make its total header size 22 bytes.

The value of this pre-pad is #defined as PKT_PREPAD in the file LLSERIAL.H.

4.2.2 Hardware Independent Low-Level Serial Driver: LLSERIAL.c

The low-level serial port driver API is discussed in the *Hardware Adaptation Layer (HAL)* section of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). It is very similar to the low-level Ethernet driver and, like the Ethernet driver, it consists of two parts: a hardware independent module and a hardware specific module. This makes the hardware specific portion of the driver easier to port.

The standard API to access a serial port as defined in the *Hardware Adaptation Layer (HAL)* section of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524) is implemented by the LLSERIAL.C module. This module can also handle multiple device instances.

To implement the low-level serial API in a device independent manner, the LLSERIAL module calls down to a hardware specific module. The interface functions to this module are defined in the LLSERIAL.H include file. The API description of this hardware's specific mini-driver is documented in Section 4.3.4.

4.2.3 Hardware Specific Low-Level Serial Driver

The TI752.C modules include a device driver specific to the TL16C752. These modules communicate with the serial hardware.

In HDLC mode, they also must check the HDLC frame delimiters, add or remove escape sequences, compute or validate the HDLC CRC, and indicate data to the upper layers as frames.

The calling interface to this mini-driver is described in the following section.

4.3 Serial Port Mini-Driver

4.3.1 Overview

As mentioned in the previous section, the low-level serial port driver is broken down into two distinct parts: a hardware independent module (LLSERIAL.C) that implements the llSerial API, and a hardware specific module that interfaces to the hardware independent module. The llSerial API is described in the *Hardware Adaptation Layer (HAL)* section of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). This section describes this small hardware specific module, or mini-driver.

Note that this module is purely optional. A valid serial port driver can be developed by directly implementing the llSerial API described in the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). Even if the mini-driver is used, you may change any of the internal data structures as long as the llSerial interface remains unchanged.

4.3.2 Global Instance Structure

Nearly all the functions in the mini-driver API take a pointer to a serial driver instance structure called SDINFO. This structure is defined in LLSERIAL.H. The following are the base members. The structure can be extended by the mini-driver.

```
//
// Serial device information
//
typedef struct _sdinfo {
    uint            PhysIdx;          // Physical index of device (0 to –1)
    uint            Open;             // Open counter used by llSerial
    HANDLE          hHDLC;            // Handle to HDLC driver (NULL=closed)
    STKEVENT_Handle hEvent;           // Handle to scheduler event object
    UINT32          PeerMap;          // 32 bit char escape map (for HDLC)
    uint            Ticks;            // Track timer ticks
    uint            Baud;             // Baud rate
    uint            Mode;             // Data bits, stop bits, parity
    uint            FlowCtrl;         // Flow Control Mode
    uint            TxFree;           // Transmitter "free" flag
    PBMQ            PBMQ_tx;          // Tx queue (one for each SER device)
    PBMQ            PBMQ_rx;          // Rx queue (one for each SER device)

    PBM_Handle      hRxPend;          // Packet being rx'd
    UINT8           *pRxBuf;          // Pointer to write next char
    uint            RxCount;          // Number of bytes received
    UINT16          RxCRC;            // Receive CRC
    UINT8           RxFlag;           // Flag to "un-escape" character

    PBM_Handle      hTxPend;          // Packet being tx'd
    UINT8           *pTxBuf;          // Pointer to next char to send
    uint            TxCount;          // Number of bytes left to send
    UINT16          TxCRC;            // Transmit CRC
    UINT8           TxFlag;           // Flag to insert character
    UINT8           TxChar;           // Insert character

    void (*cbRx)(char);               // Charmode callback (when open)
    void (*cbTimer)(HANDLE h);        // HDLC Timer callback (when open)
    void (*cbInput)(PBM_Handle hPkt); // HDLC Input callback (when open)
    uint            CharReadIdx;      // Charmode read index
    uint            CharWriteIdx;     // Charmode write index
    uint            CharCount;        // Number of charmode bytes waiting
    UINT8           CharBuf[CHAR_MAX];// Character mode recv data buffer
} SDINFO;
```

Only some of these fields are used in a mini-driver. The structure entries as defined as follows:

Table 2. Structure Entries

Field	Description
PhysIdx	Physical Index of this Device (0 to –1). The physical index of the device determines how the device instance is represented to the outside world. The mini-driver is not concerned about the physical index.
Open	Open Flag. This flag is used by LLSERIAL.C to track whether the mini-driver has been opened. It should not be modified by the mini-driver code.
hHDLC	Handle to HDLC Driver. The handle to the HDLC device is how the system tracks where HDLC data should be sent. When this field is NULL, the driver is not open for HDLC mode, and all data should be treated as character mode. When the field is not NULL, any incoming serial data should be treated as potential HDLC data, and any output packet is treated as an egress HDLC frame. HDLC packets received in HDLC mode are tagged with this handle so that the upper layers can identify the packet's source.
hEvent	Handle to Scheduler Event Object. The handle hEvent is used with the STKEVENT function *STKEVENT_signal()* to signal the system whenever new data is received. In character mode, this event is fired for each character. In HDLC mode, the event is fired when a good HDLC packet is received.

Table 2. Structure Entries (continued)

Field	Description
PeerMap	32-Bit Char Escape Map (for HDLC). The peer map is a 32-bit bitmap coded as (1<<char) where char is an ASCII character 0 through 31. When the bit is set, an outgoing HDLC frame must have the corresponding character escaped in a HDLC frame transmission.
Ticks	Track Timer Ticks. This field converts 100 ms timer ticks to 1 second timer ticks. It is not used by mini-drivers.
Baud	Serial Device Baud Rate. This field holds the current physical baud rate of the serial port in bps (for example, 9600, 19200, 153600, etc.).
Mode	Device Mode. The mode field holds the mode of the serial port in terms of data bits, stop bits, and parity. These values appear in HAL.H. Currently defined values are as follows: • #define HAL_SERIAL_MODE_8N1 0 • #define HAL_SERIAL_MODE_7E1 1
FlowCtrl	Flow Control Mode. The FlowCtrl field determines the flow control mode. These values appear in HAL.H. Currently defined values are as follows: • #define HAL_SERIAL_FLOWCTRL_NONE 0 • #define HAL_SERIAL_FLOWCTRL_HARDWARE 1
TxFree	Transmitter Free Flag. The TxFree flag is used by LLSERIAL.C to determine if new data should be sent immediately by the mini-driver, or placed on the transmit pending queue for later. If the flag is not zero, the mini-driver function *HwSerTxNext()* is called when any new data is queued for transmission. This flag is maintained by the mini-driver.
PBMQ_tx	Tx Queue. The PBMQ_tx queue is a queue of packets waiting to be transmitted. When the transmitter is free and the *HwSerTxNext()* function is called, the mini-driver removes the next packet off this queue and starts transmission. The PBMQ object is a queue of PBM packet buffers and it is operated on by the PBMQ functions defined in the*TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524).
PBMQ_rx	Rx Queue. The PBMQ_rx queue is a queue of packets that have been received on the interface. When a new packet is received, the mini-driver adds it to this queue, and fires a serial event to the STKEVENT handle. PBMQ object is a queue of PBM packet buffers and it is operated on by the PBMQ functions defined in the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524).
hRxPend	PBM_Handle to Packet Being Received. When in HDLC mode, this value holds a handle to the packet that is currently being received by the mini-driver. When the packet is complete, the mini-driver places this packet in the PBMQ_rx queue, and allocates another free packet by calling *PBM_alloc()*.
pRxBuf	Pointer to Next Character in Packet to Receive. When in HDLC mode, this points to where to write the next character of received data. The pointer points somewhere in the current packet buffer whose handle is stored in hRxPend.
RxCount	Number of Bytes Written to RX Packet Buffer so far. When in HDLC mode, this value is the number of characters that have been written to the current packet being received.
RxCRC	RX CRC Running Total. When in HDLC mode, this value is a running total of the current CRC value of the packet being received. It is used as a temporary CRC holding value while packet data is still being received. It is then compared to the CRC contained in the packet to validate the incoming CRC.
RxFlag	Flag Indicating That Next Byte is the Second Half of an Escape Sequence. When in HDLC mode, this flag is set when an escape character is seen. It prompts the RX state machine in the mini-driver to un-escape the next character received.
hTxPend	PBM_Handle to Packet Being Transmitted. This value holds a handle to the packet that is currently being transmitted by mini-driver. When the packet is completely transmitted, the mini-driver frees this packet by calling *PBM_free()*.
pTxBuf	Pointer to Next Character in Packet to Transmit. This is a pointer where to read the next character of transmit data. The pointer points somewhere in the current packet buffer whose handle is stored in hTxPend.
TxCount	Number of Bytes Yet to Send From to TX Packet. This value is the number of characters that have yet to be read and transmitted from the current packet being sent.

Table 2. Structure Entries (continued)

Field	Description
TxCRC	TX CRC Running Total. When in HDLC mode, this value is a running total of the current CRC value of the packet being transmitted. It is used as a temporary CRC holding value while packet data is still being sent. It patches in the correct CRC value as the last two bytes of the packet data.
TxFlag	Flag Indicating That Next Byte is the Second Half of an Escape Sequence. When in HDLC mode, this flag is set when an escape character has to be generated. It prompts the TX state machine in the mini-driver to write the second half of the escape sequence next. This value is stored in TxChar.
cbRx	Pointer to Character Mode Callback Function. This character mode callback function is called by LLSERIAL.C whenever there is character mode data queued up by the serial driver. This is not used by the mini-driver.
cbTimer	Pointer to HDLC Timer Callback Function. The serial driver (LLSERIAL.C) calls this function once every second. The callback function is not used by the mini-driver.
cbInput	Pointer to HDLC Input Callback Function. The serial driver (LLSERIAL.C) calls this function with new HDLC packets. The callback function is not used by the mini-driver.
TxChar	Second Half of Escape Sequence. When in HDLC mode and TxFlag is set, this variable holds the next value to send out the serial port.
CharReadIdx	Character Buffer Read Index. This index is used by LLSERIAL.C to read character data out of the circular character buffer. It is not used by a mini-driver.
CharWriteIdx	Character Buffer Write Index. This index is used by a mini-driver in character mode to write newly received character data to circular character buffer array contained in this structure. As data is written, this index is increased and the CharBufUsed value is increased. Once it reaches the value CHAR_MAX, it is reset to zero.
CharCount	Characters Stored in Character Buffer. Data received in character mode are not placed in a serial frame buffer, but are stored in a circular buffer contained in this instance structure. The maximum number of characters that can be stored is determined by CHAR_MAX. The number of characters currently stored is determined by this value. The value is increased as characters are written to the buffer. The LLSERIAL.C module will decrement this value as characters are read out, so it should only be altered in a critical section.
CharBuf	Character Mode Input Data Buffer. This array acts as the input buffer for character mode data. Unlike HDLC data, individual characters are not built into serial packet buffers. Instead, they are queued for immediate consumption by the character mode user - most likely an AT command set modem state machine, but it could also be a serial console program. The size of this buffer is set by CHAR_MAX.

4.3.3 Mini-Driver Operation

The serial mini-driver is charged with maintaining the serial device hardware, and servicing any required communications interrupts. It is built around a simple open/close concept. When open, the driver is active, and when closed is it not. In general, it must implement the mini-driver API described in the following section. Here are some additional notes on its internal operation.

4.3.3.1 Receive Operation

The mini-driver receives serial data and must classify it as HDLC data or character mode data. It is sufficient to use the current *mode* of the driver to determine how to classify data. For example:

```
// If HDLC handle valid, driver is open on HDLC mode
// Else use charmode
if( MyInstancePtr->hHDLC )
   Treat_Data_as_HDLC();
else
   Treat_Data_as_CharacterMode();
```

More advanced classification heuristics can be attempted (auto recognition of HDLC frames). Once the data is classified, it is placed either in a PBM packet buffer (if HDLC), or the circular character buffer (if character mode data). Empty packet buffers are acquired by calling the *PBM_alloc()* function. The character mode buffer array for non-HDLC data is located in the mini-driver device instance, using the structure fields: CharBuf, CharCount, and CharWriteIdx. When CharCount equals CHAR_MAX, and no more data can be written to the buffer, any new data is discarded.

When the driver is in HDLC mode, the driver receives serial data as HDLC packets, and creates a PBM packet buffer object to hold each HDLC frame. Note that the HDLC flag character (0x7E) is always removed from the HDLC packets. The completed HDLC packet written to the PBM packet buffer has the following format:

Addr (FF)	Control (03)	Protocol	Payload	CRC
1	1	2	1500	2

When a HDLC packet is ready, the mini-driver adds it to the PBMQ_rx queue and signals an event to the STKEVENT object.

On receive, the mini-driver must remove all HDLC escape sequences, and validate the HDLC CRC. Packets with an invalid CRC are discarded. CRC calculation for both receive and transmit is done in-line as the packet is being received. Also, the CRC code in the example driver is based on a 4 bit algorithm. This allows for the use of a 16 entry lookup table instead of a 256 entry table.

4.3.3.2 Transmit Operation

Unlike receive, transmit uses PBM packet buffers to send regardless of whether it is in character mode or HDLC mode. The only difference is that in HDLC mode, the data must be formatted. The mini-driver retrieves the next packet to send off the PBMQ_tx queue when the *HwSerTxNext()* function is called. When all the characters from the packet have been read and transmitted, the PBM packet buffer is freed by calling *PBM_free()*.

On transmit, the mini-driver must use escape sequences when necessary, and compute the HDLC CRC. Note that on a transmitted packet, the 2 byte HDCL CRC is present, just not valid. The mini-driver must validate the CRC when it sends the packet. CRC calculation for both receive and transmit is done in-line as the packet is being received. Also, the CRC code in the example driver is based on a 4 bit algorithm. This allows for the use of a 16 entry lookup table instead of a 256 entry table.

4.3.4 Serial Mini-Driver API

The following API functions must be provided by a mini-driver.

HwSerInit — *Initialize Serial Port Environment*

Syntax uint HwSerInit();

Parameters None

Return Value The number of serial devices in the system.

Description Called to initialize the serial port mini-driver environment, and enumerate the number of devices in the system. A device instance may be opened for each device represented in the return count. If the function returns zero, no serial devices are supported.

HwSerShutdown — *Shutdown Serial Port Environment*

Syntax void HwSerShutdown();

Parameters None

Return Value None

Description Called to indicate that the serial port environment should be completely shut down.

HwSerOpen — *Open Serial Port Device Instance*

Syntax uint HwSerOpen(SDINFO *pi);

Parameters **pi**- Pointer to serial device instance structure

Return Value Returns 1 if the driver was opened, or 0 on error.

Description Called to open a serial device instance. When called, SDINFO structure is valid.

HwSerClose — *Close Serial Port Device Instance*

Syntax void HwSerClose(SDINFO *pi);

Parameters **pi**- Pointer to serial device instance structure

Return Value None

Description Called to close a serial device instance. When called, any PBM packet buffers held by the driver instance, including hRxPend, hTxPend, and PBMQ_tx, are freed by calling *PBM_free()*. In addition, the character mode buffer is reset (read pointer, write pointer, and character count all set to NULL). Packets that have been placed on the PBMQ_rx queue are flushed by LLSERIAL.C.

HwSerTxNext — *Transmit Next Buffer in Transmit Queue*

Syntax void HwSerTxNext(SDINFO *pi);

Parameters **pi**- Pointer to serial device instance structure

Return Value None

Description Called to indicate that a PBM packet buffer has been queued in the transmit pending queue (PBMQ_tx) contained in the device instance structure, and LLSERIAL.C believes the transmitter to be free (TxFree set to 1). The mini-driver uses this function to start the transmission sequence.

HwSerSetConfig *Set Serial Port Configuration*

Syntax void HwSerSetConfig(SDINFO *pi);

Parameters **pi**- Pointer to serial device instance structure

Return Value None

Description Called when the values contained in the SDINFO instance structure are altered. The structure fields used for configuration are Baud, Mode, and FlowCtrl. The mini-driver should update the serial port configuration with the current SDINFO settings.

HwSerPoll *Serial Polling Function*

Syntax void _HwSerPoll(SDINFO *pi, uint fTimerTick);

Parameters **pi**- Pointer to serial device instance structure

fTimerTick- Flag indicating the 100 ms have elapsed

Return Value None

Description Called by LLSERIAL.C at least every 100 ms, but calls can come faster when there is serial activity. The mini-driver is not required to perform any operation in this function, but it can be used to check for device lockup conditions. When the call is made due to the 100 ms time tick, the fTimerTick calling parameter is set.

Note that this function is not called in kernel mode (hence, the underscore in the name). This is the only mini-driver function called from outside kernel mode (done to support polling drivers).

5 Ethernet Driver

This section describes the operational theory of the low-level Ethernet driver, including instructions on the use and porting of the device driver source code.

5.1 Introduction

The Ethernet packet driver provided in NDK is broken down into two parts: a device independent upper layer and a device dependent layer. The device dependent layer is called a mini-driver because it only implements a subset of the full driver functions. The mini-driver API is documented at the end of this section. The device independent upper layer of the Ethernet packet driver can be implemented in two ways: using the old-style LL packet driver architecture and using the new NIMU architecture. The NIMU architecture enables the NDK stack to control and communicate with multiple instances of a driver. This is more beneficial when compared to the old LL packet driver architecture; originally, the NDK stack could communicate with only one instance of the LL driver at a time. An NDK Ethernet driver developer needs to make a design choice of either going with the LL packet driver or NIMU architectures and develop the driver accordingly. The LL packet driver architecture API is documented in the *Hardware Adaptation Layer (HAL)* section of *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524). The NIMU architecture and API are described in detail in the *Network Interface Management Unit (NIMU)* section of *TMS320C6000 Network Developer's Kit (NDK) Programmer's Reference Guide* (SPRU524). Also, the NIMU NDK core stack changes and build instructions can be found in the *Stack Library Design* and *Building in NIMU* sections of *TMS320C6000 Network Developer's Kit (NDK) User's Guide* (SPRU523), respectively.

5.1.1 Ethernet Driver Source Files

The Ethernet packet driver source files are located in various subdirectories according to their function.

Table 3. Ethernet Packet Driver Source Files

Directory	File	Function
<SRC\HAL\EVMDM642\ETH_DM642>		Source code of the Texas Instruments DM642 Ethernet Driver
	NIMU_DM642.C	Hardware independent portion of Ethernet packet driver using NIMU architecture
	LLPACKET.C	Hardware independent portion of the low-level Ethernet packet driver
	LLPACKET.H	Private include file for LLPACKET and NIMU drivers
	DM642.C	Packet mini-driver for DM642 EMAC

5.2 Ethernet Driver

The NDK LL packet driver API and the NIMU API are discussed in the *Hardware Adaptation Layer (HAL)* and *Network Interface Management Unit (NIMU)* sections of the *TMS320C6000 Network Developer's Kit (NDK) Programmer's Reference Guide* (SPRU524), respectively, which includes how to implement the individual API functions. The sections below discuss the implementation of an Ethernet packet mini-driver.

5.2.1 Important Note on Data Alignment

The NDK libraries have been built with the assumption that the IP header in a data packet is 16-bit aligned. In other words, the first byte of the IP packet (the version/length field) must start on an even 16-bit boundary. In any fixed length header protocol, this requirement can be met by backing off any odd byte header size, and adding it to the header padding specified to the stack. For Ethernet and PPP, the only requirement is that the Ethernet or PPP packet not start on an odd byte boundary.

In addition, all drivers in the NDK are set up to have a 22 byte header. This is the header size of a PPPoE packet when sent using a 14 byte Ethernet header. When all arriving packets use the 22 byte header, it guarantees that they can be routed to any egress device with a header requirement up to that size. For Ethernet operation, this requires that a packet has 8 bytes of pre-pad to make its total header size 22 bytes.

The value of this pre-pad is #defined as PKT_PREPAD in the file LLPACKET.H.

5.2.2 Hardware Independent Low-Level Ethernet Driver

The low-level Ethernet packet driver is similar to the low-level serial port driver. It consists of two parts: a hardware independent module and a hardware specific module, which makes the hardware specific portion of the driver easier to port. When deciding how to port the packet driver, you must choose for the device independent module either the LLPACKET.C or the NIMU enabled counterpart NIMU_DM642.C. Currently, the choice between NIMU and LLPacket modules is done based on whether the pre-processor symbol _INCLUDE_NIMU_CODE is defined or not. If this constant is not defined, LLPACKET.C is compiled in, otherwise NIMU_DM642.C is compiled in. For instructions to build in NIMU using Code Composer Studio, see Step 14 in Section 1.2 of this document and the *Building in NIMU* section of the *TMS320C6000 Network Developer's Kit (NDK) User's Guide* (SPRU523).

The standard API to access the packet device as defined in the *Hardware Adaptation Layer (HAL)* section of the *TMS320C6000 Network Developer's Kit (NDK) Programmer's Reference Guide* (SPRU524) is implemented by the LLPACKET.C module. The NIMU architecture and API described in detail in the *Network Interface Management Unit (NIMU)* section of the *TMS320C6000 Network Developer's Kit (NDK) Programmer's Reference Guide* (SPRU524) is implemented by the NIMU_DM642.C module. The NIMU enabled module can handle multiple device instances, and is charged with handling the queuing for all received packet data.

To implement the low-level packet API in a device independent manner, the LLPACKET.C or NIMU_DM642.C modules call down to a hardware specific module. The interface functions to this module are defined in the LLPACKET.H include file. They are documented to some degree in the example source code of the hardware specific modules. The LLPACKET.H file also contains the specifications for the buffering of packets.

5.2.3 Hardware Specific Low-Level Ethernet (Mini) Driver

The mini-driver module is a device driver specific to its target hardware. Its basic function is to talk to the Ethernet MAC hardware. It also must interface to any other hardware specific to the target platform. For example, it can setup interrupts, cache control, and the EDMA controller.

The interface specification is capable of handling multiple devices, but the example implementations mostly only support a single device instance. Notes are made in the source code as to where alterations can be made to support multiple devices.

5.3 Ethernet Packet Mini-Driver

5.3.1 Overview

As mentioned in the previous section, the low-level Ethernet packet driver is broken down into two distinct parts: a hardware independent module (LLPACKET.C/NIMU_DM642.C) that implements the llPacket/NIMU APIs and a hardware specific module that interfaces to the hardware independent module. This section describes this small hardware specific module, or mini-driver.

Note that this module is purely optional. A valid packet driver can be developed by directly implementing the llPacket/NIMU API described in the *TMS320C6000 Network Developer's Kit (NDK) Programmer's Reference Guide* (SPRU524). Even if the mini-driver is used, you can change any of the internal data structures as long as the llPacket/NIMU interface remains unchanged.

5.3.2 Global Instance Structure

Nearly all the functions in the mini-driver API take a pointer to a packet driver instance structure called PDINFO. This structure is defined in LLPACKET.H:

```
//
// Packet device information
//
typedef struct _pdinfo {
    uint                        PhysIdx;                // Physical index of this device (0 to
-1)
    HANDLE                      hEther;                 // Handle to logical driver
    STKEVENT_Handle   hEvent;
    UINT8                       bMacAddr[6];            // MAC Address
    uint                        Filter;                 // Current RX filter
    uint                        MCastCnt;               // Current MCast Address Countr
    UINT8                       bMCast[6*PKT_MAX_MCAST];
    uint                        TxFree;                 // Transmitter "free" flag
    PBMQ                        PBMQ_tx;                // Tx queue (one for each PKT device)
#ifdef _INCLUDE_NIMU_CODE
    PBMQ                        PBMQ_rx;                // Rx queue (one for all PKT devices)
#endif
} PDINFO;
```

Only some of these fields are used in a mini-driver. The structure entries as defined as shown in Table 4:

Table 4. Structure Entries

Field	Description
PhysIdx	Physical Index of This Device (0 to –1). The physical index of the device determines how the device instance is represented to the outside world. The mini-driver is not concerned about the physical index.
hEther	Handle to Ethernet Driver. This is a handle NDK Ethernet instance that is bound to the physical Ethernet driver. When a packet is received, it is tagged with this Ethernet handle before being placed on the global PBMQ_rx queue. This allows the Ethernet module to identify the ingress device.
hEvent	Handle to Scheduler Event Object. The handle hEvent is used with the STKEVENT function *STKEVENT_signal()* to signal the system whenever a new packet is received.
bMacAddr	Ethernet MAC Address. This is a byte array that holds the Ethernet MAC address. It is set to a default value by LLPACKET.C/NIMU_DM642.C, but can be used or altered by the mini-driver when the device opens. If the MAC contains its own unique MAC address, this value is written to bMacAddr. If the MAC does not have a MAC address, the value bMacAddr programs the MAC device.
Filter	Current Rx Filter. The receive filter determines how the packet device should filter incoming packets. This field is set by LLPACKET.C/ NIMU_DM642.C and used by the mini-driver to program the MAC. Legal values include: • ETH_PKTFLT_NOTHING: No Packets • ETH_PKTFLT_DIRECT: Only directed Ethernet • ETH_PKTFLT_BROADCAST: Directed plus Ethernet Broadcast • ETH_PKTFLT_MULTICAST: Directed, Broadcast, and selected Ethernet Multicast • ETH_PKTFLT_ALLMULTICAST: Directed, Broadcast, and all Multicast • ETH_PKTFLT_ALL: All packets
MCastCnt	Number of Multicast Addresses Installed. The field holds the current number of multicast addresses stored in the multicast address list (also in this structure). The multicast address list determines what multicast addresses (if any) the MAC is allowed to receive.
bMCast	Multicast Address List. This field is a byte array of consecutive 6 byte multicast MAC addresses. The number of valid addresses is stored in the MCastCnt field. The multicast address list determines what multicast addresses (if any) the MAC is allowed to receive.
TxFree	Transmitter Free Flag. The TxFree flag is used by LLPACKET.C/ NIMU_DM642.C to determine if a new packet can be sent immediately by the mini-driver, or if it should be placed on the transmit pending queue for later. If the flag is not zero, the mini-driver function *HwPktTxNext()* is called when a new packet is queued for transmission. This flag is maintained by the mini-driver.
PBMQ_tx	Transmit Pending Queue. The transmit pending queue holds all the packets waiting to be sent on the Ethernet device. The mini-driver pulls PBM packet buffers off this queue in its *HwPktTxNext()* function and posts them to the Ethernet MAC for transmit. Once the packet has been transmitted, the packet buffer is freed by calling *PBM_free()*.

5.3.3 Mini-Driver Operation

The Ethernet packet mini-driver maintains the device hardware, and services any required communications interrupts. It is built around a simple open/close concept. When open, the driver is active, and when closed, it is not. In general, it must implement the mini-driver API described in the following section. The following sections provide additional information on its internal operation.

5.3.3.1 Receive Operation

The mini-driver receives packets when the device is open. When an Ethernet packet is received, it is placed in a PBM packet buffer. Empty packet buffers are allocated by calling *PBM_alloc()*.

Once the packet buffer is filled, it should be placed onto the receive pending queue (*PBMQ_rx*) defined in the LLPACKET.H. For LLPacket style devices, there is only one RX queue for all Ethernet devices, therefore, the mini-driver must set the RX IF device to the value of hEther in the instance structure before placing it on the RX queue. Contrary to this, the NIMU style devices have one RX queue each for each Ethernet device instance; the mini-driver can simply en-queue the packet buffer onto the RX queue of that Ethernet instance.

After the data frame buffer has been pushed onto the Rx queue, the mini-driver signals an Ethernet event to the STKEVENT handle supplied in the driver instance structure.

5.3.3.2 Transmit Operation

When the transmitter is idle, the mini-driver must set the TxFree field of its instance structure to 1. When a new packet is ready for transmission, LLPACKET.C/ NIMU_DM642.C places the PBM packet buffer on the PBMQ_tx queue of the mini-driver's instance structure.

Once a new packet has been written to the transmit pending queue, if TxFree is set, LLPACKET.C/NIMU_DM642.C calls the mini-driver *HwPktSendNext()* function. At this time, the mini-driver should clear the TxFree field, and start transmission of the packet. Once the packet has been sent, the packet buffer is freed by calling *PBM_free()*. This call can be made at interrupt time.

5.3.4 Ethernet Packet Mini-Driver API

The following API functions must be provided by a mini-driver.

HwPktInit	***Initialize Packet Driver Environment***
Syntax	uint HwPktInit();
Parameters	None
Return Value	The number of Ethernet packet devices in the system
Description	Called to initialize the packet mini-driver environment, and enumerate the number of devices in the system. A device instance may be opened for each device represented in the return count. If the function returns zero, no devices are supported.

HwPktShutdown	***Shutdown Packet Driver Environment***
Syntax	void HwPktShutdown();
Parameters	None
Return Value	None
Description	Called to indicate that the packet driver environment should be completely shut down.

HwPktOpen — *Open Ethernet Packet Device Instance*

Syntax uint HwPktOpen(PDINFO *pi);

Parameters **pi**- Pointer to Ethernet packet device instance structure

Return Value Returns 1 if the driver was opened, or 0 on error.

Description Called to open a packet device instance. When called, PDINFO structure is valid. The device should be opened and made ready to receive and transmit Ethernet packets.

HwPktClose — *Close Ethernet Packet Device Instance*

Syntax void HwPktClose(PDINFO *pi);

Parameters **pi**- Pointer to Ethernet packet device instance structure

Return Value None

Description Called to close a packet device instance. When called, any outstanding packet buffers held by the instance should be freed using *PBM_free()*.

HwPktTxNext — *Transmit Next Buffer in Transmit Queue*

Syntax void HwPktTxNext(PDINFO *pi);

Parameters **pi**- Pointer to Ethernet packet device instance structure

Return Value None

Description Called to indicate that a packet buffer has been queued in the transmit pending queue contained in the device instance structure, and LLPACKET.C/NIMU_DM642.C believes the transmitter to be free (TxFree set to 1). The mini-driver uses this function to start the transmission sequence.

HwPktSetRx — *Set Ethernet Rx Filter*

Syntax void HwPktSetRx(PDINFO *pi);

Parameters **pi**- Pointer to Ethernet packet device instance structure

Return Value None

Description Called when the values contained in the PDINFO instance structure for the Rx filter or multicast list are altered. The mini-driver should update its filter settings at this time.

HwPktIoctl — *Execute Driver Specific IOCTL Command*

Syntax

uint HwPktIoctl(PDINFO *pi, uint cmd, void *arg);

Parameters

pi- Pointer to Ethernet packet device instance structure

cmd - Device specific command

arg - Pointer to command specific argument

Return Value

This function returns 1 on success.

Description

Execute driver specific IOCTL command. There are no command defined for existing driver.

_HwPktPoll — *Mini-Driver Polling Function*

Syntax

void _HwPktPoll(SDINFO *pi, uint fTimerTick);

Parameters

pi- Pointer to serial device instance structure

fTimerTick- Flag indicating the 100 ms have elapsed

Return Value

None

Description

Called by LLPACKET.C/NIMU_DM642.C at least every 100 ms, but calls can come faster when there is network activity. The mini-driver is not required to perform any operation in this function, but it can be used to check for device lockup conditions. When the call is made due to the 100 ms time tick, the *fTimerTick* calling parameter is set.

Note that this function is not called in kernel mode, hence, the underscore in the name. This is the only mini-driver function called from outside the kernel mode (done to support polling drivers).

Appendix A Revision History

Table A-1 lists the changes made since the previous version of this document.

Table A-1. Document Revision History

Reference	Additions/Modifications/Deletions
Section 1.2	Replaced contents with new source
Section 1.3	Replaced contents with new source
Section 5.1	Replaced contents with new source
Table 3	Replaced contents of the table
Section 5.2	Replaced contents with new source
Section 5.2.2	Changed title of the section
Section 5.2.2	Replaced contents with new source
Section 5.3.1	Replaced contents with new source
Section 5.3.2	Replaced text and code
Section 5.3.2	Replaced text in table - bMacAddr
Section 5.3.2	Replaced text in table - Filter
Section 5.3.2	Replaced text in table - TxFree
Section 5.3.3.1	Replaced contents with new source
Section 5.3.3.2	Replaced contents with new source
Section 5.3.4	Replaced text in module - HwPktTxNext
Section 5.3.4	Replaced text in module - _HwPktPoll