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

User's Guide

TEXAS INSTRUMENTS

Contents

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 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

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>` CCStudio project files for cfgdemo example
- `<example/network/client/evmdm642>` CCStudio project files for client example
- `<example/network/helloWorld/evmdm642>` CCStudio project files for helloWorld example
- `<example/serial/client/evmdm642>` CCStudio project files for serial client example
- `<example/serial/router/evmdm642>` CCStudio 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 UART driver
- `<src/hal/evmdm642/userled_dm642>` Source code for EVMDM642 LED driver

1.3   Rebuilding HAL Libraries

Included with the Support Package is a new batch file (MAKEHAL_EVMDM642.BAT) that resides in the `<NDK_INSTALL_DIR>/packages/ti/ndk` directory.

Before using MAKEHAL_EVMDM642 from a command prompt, the NDK batch file DOSRUN_BIOS.BAT must be run from the `<NDK_INSTALL_DIR>/packages/ti/ndk` directory to set up the correct environment for running the TI code generation tools from a command prompt. Make sure that the TI_DIR environment variable is set to point to your Code Composer Studio Development Tools installation. The form of the MAKEHAL_EVMDM642 command is: `makehal_evmdm642 [library] (noclean)`

The library name is required in this command. The value of library determines what device library (or libraries) to build. The value of library can be any of the following:
- ETHERNET: Texas Instruments DM642 Ethernet MAC Driver
- SERIAL: Texas Instruments TL16C752 Dual Serial UART Driver

- USERLED: EVMDM642 User LED Driver

The final parameter *noclean* can be added to the command line to suppress cleaning old object files from the target directory. This is only useful when rebuilding the same driver for the same platform baseboard.

The batch file does not perform stringent argument checking, so incorrect calling arguments may result in a incorrect build.

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 Appendix D of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524), and the interface to the HDLC framer is documented in Appendix C. 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 Appendix D 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 Appendix D 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 *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524), Appendix D. 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 n-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.

www.ti.com

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 nomore 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 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 at the end of this section. The full NDK Ethernet packet driver API is documented in the Appendix D of *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524).

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
	LLPACKET.C	Hardware independent portion of the Low-Level Ethernet Packet Driver
	LLPACKET.H	Private include file for LLPACKET drivers
	DM642.C	Packet mini-driver for DM642

5.2   Ethernet Driver

The NDK packet driver API is discussed in Appendix D of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524), 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: LLPACKET.C

The low-level Ethernet packet driver is very 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 whether to use the device independent LLPACKET.C module.

The standard API to access the packet device as defined in Appendix D of the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524) is implemented by the LLPACKET.C module. This module also handles multiple device instances, and handles the queuing for all received packet data.

To implement the low-level packet API in a device independent manner, the LLPACKET.C module calls 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 to 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) that implements the llPacket API described in the *TMS320C6000 Network Developer's Kit (NDK) Software Programmer's Reference Guide* (SPRU524), Appendix D, 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 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 llPacket 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 n-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)
} PDINFO;
```

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

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, 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 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 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.C module. There is one RX queue for all Ethernet devices. 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.

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 will place 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 will call 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 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.

_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 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 kernel mode (done to support polling drivers).

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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