DSP/BIOS Hardware and Software UART Device Drivers

Software Development Systems

ABSTRACT

This application note describes the implementation of a DSP/BIOS device driver for both hardware UARTs and software simulated UARTs. The hardware UART is the 16550-based UART on the DSK 5402, while the software UART is one that is simulated on a DSP's McBSP channel. These drivers were written in conformance to the DSP/BIOS IOM device driver model and APIs.

Features:

- Applications can use both the hardware and the software UART drivers seamlessly using the same functions.
- The UARTs can be configured via device-specific configuration structures to suit a wide range of application needs.
- Applications can access the drivers either via the GIO class driver interface or the traditional SIO or PIP interfaces.

Contents

List of Figures

List of Tables

1 Overview

The device driver described here is part of an IOM mini-driver. That is, it is implemented as the lower layer of a 2-layer device driver model. The upper layer is called the class driver and can be either the DSP/BIOS GIO, SIO/DIO, or PIP/PIO modules. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests. Figure 1 shows the overall DSP/BIOS device driver architecture. For more information about the IOM device driver model as well as the GIO, SIO/DIO, and PIP/PIO modules, see the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

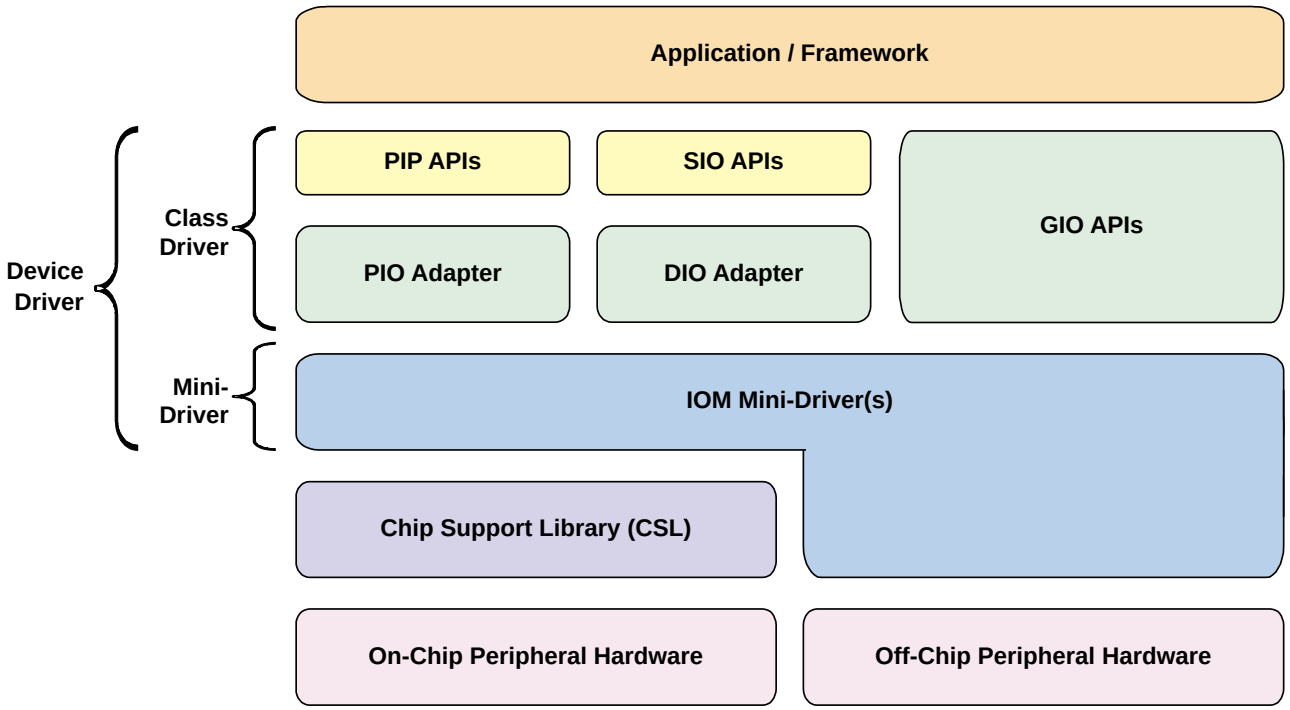

Figure 1. DSP/BIOS IOM Device Driver Model

This document deals with both the hardware and the software simulated UART drivers. As shown in Figure 2, the UART mini-driver itself is split into two parts. The generic functionality of the UART mini-driver is captured in the upper layer, which is named UARTMD. This layer also implements the interfaces necessary to subscribe to the IOM mini driver interface.

The lower layer is UART device specific and is dependent on the nature of the UART peripheral. All software simulated UARTs are implemented as libraries that are named uarthw_xxx_mcbsp.lxx, while hardware UARTs are named uarthw_xxx.lxx. For example, the lower layer of a McBSP-based UART mini-driver on the C5402 DSK would be called uarthw_dsk5402_mcbsp.l54, and the hardware UART-based mini-driver would be called uarthw_dsk5402.l54.

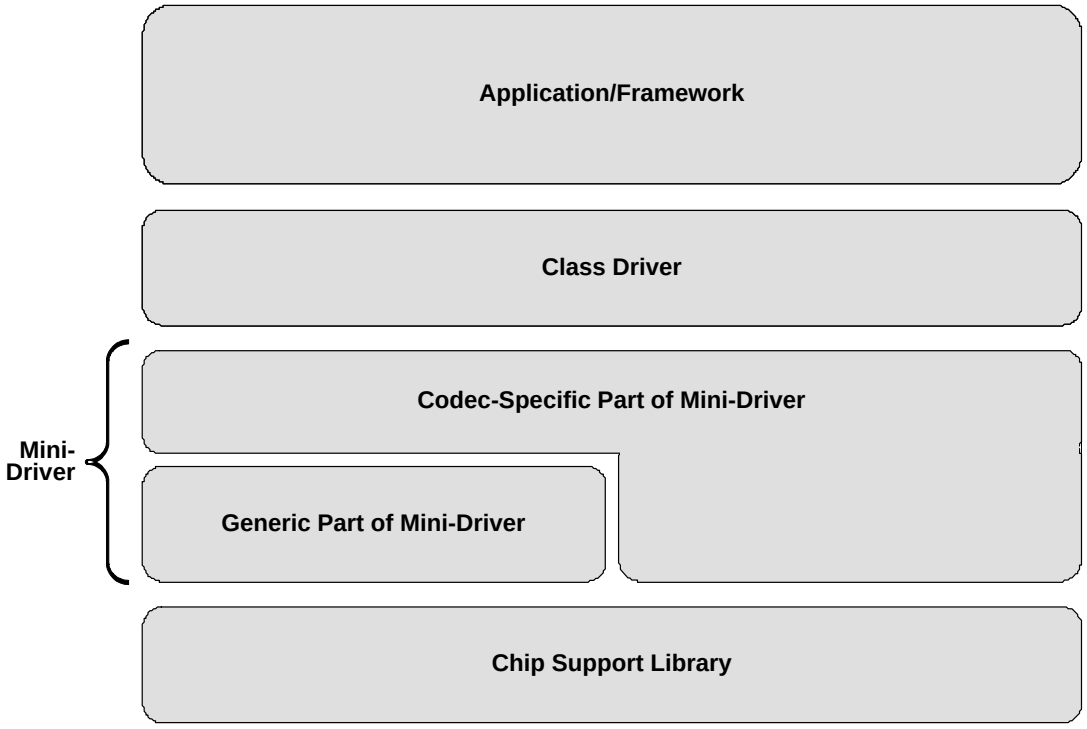

Figure 2. UART Device Driver Partitioning

This approach of splitting the mini-driver into two parts allows us to abstract the generic UART functionality in the common higher-level driver that will not change with hardware and platform. The same code is portable across multiple platforms and can be used in conjunction with any UART implementation. The only requirement is that the lower-level hardware specific implementation should subscribe to and implement the interfaces defined by this generic UART driver. In this way, the hardware specific driver can restrict itself to dealing with the device specific requirements while the generic driver takes care of IOM interfacing, data buffering, queue management, channel management and a host of other bookkeeping activities.

Applications that want to use UART mini drivers should use the generic UART mini-driver in combination with a hardware-specific mini-driver based on their need. We recommend that applications use the generic mini-driver provided with the Device Driver Developer's Kit. For the device specific layer, you can use a TI provided driver if available, or you can create your own drivers based on the defined interfaces. This way, it is easier for application writers to create their own hardware specific drivers based on their requirements without having to worry about the IOM interfacing, data buffering, queue management, channel management and other generic book keeping activities.

2 Usage

This section of the document deals with the configuration of the driver using CDB, the device specific configuration parameters which can be set by the application and the functionality exposed by the driver to the application.

2.1 Configuration

To add this device driver to the DSP/BIOS Configuration tool, open the configuration tool, right-click on the User-Defined Devices icon under the Device Drivers section and select Insert UDEV. From the Objects menu (or by right-clicking on the object), rename the object from UDEV to a unique name for the device driver. Open the Properties dialog for the device you created by right-clicking on the object and modify its properties as follows. This section only gives the configuration for a driver configured to be used via the GIO interface. For details on how to configure the driver for use with SIO or PIP refer to the appropriate documentation.

- **Init function table:** _UARTMD_init
- **Function table ptr:** _UARTMD_Fxns
- **Function table type:** Select IOM_Fxns. For SIO based usage this will be DEV_Fxns.
- **Device ID:** N/A, not used by the driver
- **Device params ptr:** A pointer to an optional device configuration structure. This structure contains fields for the UARTMD code and a pointer to an implementation-specific structure. For the 16550 based UART driver on the DSK 5402, this is a pointer to an object of type UARTHW_DSK5402_Params as defined in the header file uarthw_dsk5402.h. For a McBSP based UART driver, is a pointer to an object of type UARTHW_MCBSP_Params which is defined in the header file uarthw_mcbsp.h. If not specified, a default configuration structure shall be used which is device specific. Refer to the documentation in the device specific sections to find out more about the default configuration. More information regarding device parameter can be found in the readme.txt located in the mini-driver specific directory.
- **Device global data ptr:** N/A Not used by this driver.

2.2 Device and Channel Parameters

The driver parameters structure allows a user to specify in the DSP/BIOS Configuration tool the driver specific parameters when the driver is initialized. This configuration structure is defined as follows:

```
typedef struct UARTMD_DevParams {
    Int versionId;
    Bool            packedChars;    /* only used for c55xx and c54xx */
    Ptr             uarthwParams;   /* pointer to UARTHW–specific params */
} UARTHW_DevParams;
```

- **versionId:** Version number of the driver.

- **packedChars:** This field is only used by the C54x and C55x versions of this driver. If packedChars is TRUE, the driver will output all 16-bits of a DSP word (the low 8 bits are output first, followed by the high eight bits). The minimum addressable unit of memory for the C54x and C55x is 16 bits, but typical UART devices operate on 8-bit bytes. This flag is used specify whether the driver should output only the 8 low bits of a word or all 16 bits of the word. For ASCII or terminal based applications, this flag should be set to FALSE. The default value for this field is FALSE. This field is ignored by C6x versions of this driver.
- **uarthwParams:** This field is a pointer to the UARTHW implementation-specific configuration structure. If this field is NULL, the UARTHW code will use a default values for the UART configuration. The UARTHW parameter structures for the supported UARTHW implementations are described in the next section.

2.2.1 McBSP-Based Configuration Structure

The McBSP based UART driver's configuration structure is defined as follows:

```
/* Interrupt mask definition */
#ifdef _6x_
#include <csl_stdinc.h>         /* for Uint32 */
typedef Uns UARTHW_MCBSP_IntrMask;
#endif
#ifdef _54_
#include <csl_stdinc.h>
typedef struct _UARTHW_MCBSP_IntrMask {
     Uns rxIntrMask;
     Uns txIntrMask;
}UARTHW_MCBSP_IntrMask;
#endif
#ifdef _55_
#include <csl_std.h>
typedef struct _UARTHW_MCBSP_IntrMask {
    Uns rxIerMask[2];
    Uns txIerMask[2];
}UARTHW_MCBSP_IntrMask;
#endif

typedef struct UARTHW_MCBSP_Params {
    Uns             mcbspId;      /* McBSP port id */
    Uns             dmaRxId;      /* DMA channel id (C5000 only) */
    Uns             dmaTxId;      /* DMA channel id (C5000 only) */
    Uint32          mcbspClkIn;   /* McBSP frequency */
    Uint32          baud;         /* baud rate */
    UARTHW_MCBSP_IntrMask intrMask;
} UARTHW_MCBSP_Params;
```

- **mcbspId:** The McBSP port number to use to simulate the UART. This allows the user to specify a port other than the default.
- **dmaRxId:** The DMA channel ID to be used for receiving UART data. This is not used on C6000 DSPs.

- **dmaTxId:** The DMA channel id to be used for transmitting UART data. This is not used on C6000 DSPs.
- **mcbspClkIn:** The input frequency to the McBSP. The McBSP can be driven either by the CPU clock or an external clock.
- **baud:** The desired baud rate for the simulated UART.
- **intrMask:** Interrupt mask, set in the ISR

2.2.2 DSK 5402 Hardware UART-Based Configuration Structure

The DSK 5402 hardware UART driver's configuration structure is defined as follows:

```
typedef struct UARTHW_DSK5402_Params {

    /* Flow Control Parameters */
    UARTHW_DSK5402_FlowControl    flowControl;

    /* Communication Parameters */
    UARTHW_DSK5402_Parity         parity;
    UARTHW_DSK5402_WordLen        wordSize;
    UARTHW_DSK5402_StopBits       stopBits;
    UARTHW_DSK5402_Baud           baud;
    Uns intrMask;
} UARTHW_DSK5402_Params;
```

- **flowControl:** The type of flow control to be used. It can be one of the following values:
  UARTHW_DSK5402_FLOW_NONE (No flow control)
  UARTHW_DSK5402_FLOW_AFE_RTSCTS (Auto flow control with RTS/CTS enabled)
  UARTHW_DSK5402_FLOW_AFE_CTS (Auto flow control with CTS enabled)
- **wordSize:** The word size to be used. It can be one of the following values:
  UARTHW_DSK5402_WORD8 (8 bits per word)
  UARTHW_DSK5402_WORD7 (7 bits per word)
- **parity:** Parity can have the following values:
  UARTHW_DSK5402_DISABLE_PARITY
  UARTHW_DSK5402_EVEN_PARITY
  UARTHW_DSK5402_ODD_PARITY
- **stopBits:** This specifies the number of stop bits to be used for communication. The following values are possible:
  UARTHW_DSK5402_STOP1
  UARTHW_DSK5402_STOP2

- **baud:** This specifies the baud rate to be used for communication. It can have the following values:

  UARTHW_DSK5402_BAUD_19200 (baud of 19200)

  UARTHW_DSK5402_BAUD_38400 (baud of 38400)

  UARTHW_DSK5402_BAUD_57600 (baud of 57600)

  UARTHW_DSK5402_BAUD_115200 (baud of 115200)

If the application does not specify any configuration structure, a default configuration structure is used which specifies:

- baud to 115200
- 8-bit word size
- one stop bit
- no parity
- no flow control

This can be obtained using the macro UARTHW_DSK5402_DEFAULTPARAMS defined in the device specific header file uarthw_dsk5402.h

- **intrMask:** Interrupt mask, set in the ISR.

2.2.3 Event Handling

The application can register an application-supplied callback with the driver along with an event mask. The driver will notify the application through this callback whenever an event of interest occurs. The application can register the callback via a control call as described earlier. The following sections describe the application supplied callback and the notify structure used to supply the callback and the event mask.

2.2.3.1 Application-Supplied Callback

Description The callback function to be supplied by the application has the following prototype:

Function Prototype

```
typedef Void   (*UARTMD_TnotifyHandler)(Uns evtStatus,
Uns val);
```

Arguments

evtStatus	This specifies the event of interest to the application, which triggered the notification.
val	Holds a value that needs to be passed to the callback function based on the type of event that was generated.

Return Value none

2.2.3.2 UARTMD_NotifyStruct

The application can register with the UART driver for certain events of interest. The UART driver will invoke the application-supplied callback when any of the events of interest occur. The structure used is as shown below

```
typedef struct UARTMD_NotifyStruct _{
UARTMD_Tnotifyhandler   notifyFunc;
Uns evtmask;
} UARTMD_NotifyStruct;
```

- **notifyFunc:** is the application supplied callback function.
- **evtmask:** indicates the events in which the application is interested. The application can choose from the following events.

  UARTMD_EVT_CTSCHANGE (CTS line status change)

  UARTMD_EVT_DSRCHANGE (DSR line status change)

  UARTMD_EVT_BREAK (Detect a Break)

  UARTMD_EVT_PERR (Parity Error)

  UARTMD_EVT_FERR (Framing Error)

  UARTMD_EVT_OERR (Overrun Error)

  UARTMD_EVT_BERR (Buffer Overflow Error)

2.3 Control Commands

Table 1. Control Commands

Command	Argument	Description	Supported on H/W UARTs?	Supported on S/W UARTs?
IOM_CHAN_TIMEDOUT	Ignored	Typically issued by IOM layer when a timeout is encountered on a synchronous call. This call cleans up request queue at the UART driver. Not usually issued by application.	Yes	Yes
UARTMD_REGISTER_NOTIFY	Pointer to UARTMD_NotifyStruct which contains the event mask and callback function to be invoked when the said event occurs	Registers an application-supplied callback with the driver along with an event mask. The driver will notify the application through this callback whenever an event of interest occurs. The event mask can be a logical OR of the event values as defined by UARTMD_EventMask in the UART-specific header file uartmd.h	Yes	Yes
UARTMD_SETBREAK	The break value either 0 or 1	If arg value is 1, then break is turned on. If arg value is 0, then break is turned off.	Yes	Yes
UARTMD_GETMODEMSTATUS	Pointer to a Uns which stores the retrieved modem status	Retrieves modem status for UART device.	Yes	No
UARTMD_SETRTS	The RTS value to set either a 0 or a 1.	This call turns on or off the RTS line.	Yes	No
UARTMD_SETDTR	The DTR value to set either a 0 or a 1.	This call turns on or off the DTR line.	Yes	No

3 UART Architecture

The following sections describe the generic UART layer implementation, data structures used internally by the generic higher-level layer and the interfaces defined for the lower level drivers. Finally, we describe the implementation details for the hardware based and software based UART drivers.

3.1 Generic UART Implementation

The generic UART layer is the higher-level layer that exposes the UART interface to the application world. It implements the functions necessary to support the IOM interface, handles events coming from the low level drivers in a generic way and interfaces with the low level drivers to transmit and receive data.

3.1.1 Data Structures

The UART driver internally maintains the device context and state information using the UartPortObj structure. The structure is as shown

3.1.1.1 UART Port Object

```
typedef struct UartPortObj {
    UARTMD_TnotifyHandler        notifyFunc;
    Uns                          evtMask;
    UartChanObj                  chans[NUMCHANS];
} UartPortObj, *UartPortHandle;
```

- **notifyFunc:** the callback supplied by the application to the mini-driver to be notified of certain events of interest to the application. This field and the evtMask field are initialized when the application calls mdControlChan (via SIO_ctrl, GIO_control or PIO_ctrl).
- **evtMask:** The application specifies which events it would like to be notified about via the mdControlChan function. evtMask is a bit mask that holds the application specified value.
- **chans[NUMCHANS]:** Channel structures for the input and output channel associated with the device. This mini driver supports one input channel and one output channel, so NUMCHANS is set to 2.

3.1.1.2 UART Channel Object

The UART driver internally maintains the channel context and state information using the UartChanObj structure. The pointer to this structure is returned as the handle to the IO manager upon a device open. The structure is as shown below.

```
typedef struct UartChanObj
{
       Uns                       inUse;       /* TRUE if channel is in use */
       Int                       mode;        /* INPUT or OUTPUT */
       IOM_Packet                dataPacket;  /* current active I/O Packet
       Char                      *bufptr;     /* pointer within current buffer */
       Uns                       bufcnt;      /* size of remaining I/O job */
       QUE_Obj                   pendlist;  /* IOM_Packets pending I/O go here */
       CIRC_Obj                  circ;                /* Circular Buffer */
       IOM_TiomCallback          cbFxn;  /* to notify client when I/O complete */
       Ptr                       cbArg;       /* argument for cbFxn() */
#if SUPPORTPACKEDCHARS
       Bool                      packedChars; /* TRUE => output all 16 bits */
       Bool                      halfWay;   /* TRUE if we're between 1/2 words */
       Char                      halfWord;    /* holds 1/2 word */
#endif
} UartChanObj, *UartChanHandle;
```

- **inUse:** indicates whether a channel is currently being used.
- **mode:** indicates the mode in which the device has been opened: input or output.
- **dataPacket:** indicates the current active packet.
- **bufptr:** address for the current packet
- **bufcnt:** size of the current packet
- **circ:** the internal ring buffer being maintained by the UART driver to buffer the data received when an application read is not pending.
- **cbFxn:** the class-driver supplied callback routine for the mini-driver.
- **cbArg:** the class-driver supplied callback argument for the mini-driver.
- **packedChars:** indicates whether driver should output all 16 bits of a word or only the low 8 bits. PackedChars is only used for 54x and 55x. packedChars is set to the value specified by the UARTMD_DevParams parameter in the mini-driver's mdBindDev function.
- **halfWay:** used only when packedChars is TRUE. Halfway is set to TRUE after driver outputs or inputs the first 8 bits of a 16-bit word.
- **halfWord:** halfWord is only used when packedChars is TRUE. HalfWord is valid when halfway is TRUE. HalfWord holds the high 8 bits of the 16-bit word for output or the low 8 bits of an input word.

3.2 Packet Processing

The mini-driver interface function mdSubmitchan will be called by the class driver with a command embedded in the request packet to perform a read, a write, channel flush, or channel abort. Hardware interrupts will then be disabled for the duration of the processing of the packet. For the case of a request to read or write, the request packet will be put on the queue that was attached to the channel on which the request was made. If there are no other packets in process, the new request will be processed immediately. Otherwise, the request will remain on the queue until the request packets ahead of it on the queue are processed. Packets are processed in sequence by simply taking them off the queue one by one and setting up a transfer.

When packets are submitted to the driver with a command of Flush, all pending input jobs are to be completed in the order they were submitted with a status of IOM_ABORT and all output jobs are to be completed routinely.

When packets are submitted to the driver with a command of Abort, all pending calls are completed in the order they were submit with a status of IOM_ABORTED.

When a read request is made to the UART mini-driver and no requests are pending, the UART driver sets up the packet buffer as the source for further data transmission. If the packet buffer size is 0 then function returns IOM_COMPLETED. If the internal circular buffer is empty, then function returns with IOM_PENDING. Otherwise, a data character will be read from the internal circular buffer until the packet has no more characters or the lower-lever circular buffer is empty.

When a write request is made to the UART mini driver and no requests are pending, the UART driver sets up the packet buffer as the source for further data transmission. If the packet buffer size is 0 then function returns IOM_COMPLETED. If the internal circular buffer cannot accept any more data then function returns with IOM_PENDING. Otherwise, a data character will be written into the internal circular buffer until the packet has no more characters or the lower-lever circular buffer cannot accept any more characters.

3.3 Event Handlers

During the mdBindDev call, the generic UART layer registers a set of functions with the lower-level hardware-specific drivers to notify it of events related to the UART. These four event handlers are invoked due to line status changes, modem status changes, receipt of a new character and when the transmit buffer is empty. The four event handlers are implemented in the generic UART layer. This section describes these handler functions further.

3.3.1 cbLineStatus

Description The lower layer invokes this function to notify the generic UART layer of a change in line status. This is typically used by the UART driver to notify the application layer when it registers for notifications on line status changes.

Function Prototype `Void cbLineStatus (UartPortHandle port, Int lsrVal)`

Arguments

port	This is a mini driver device object, which identifies the UART device.
lsrVal	This indicates the line status register value at the time of the interrupt.

3.3.2 cbModemStatus

Description The lower layer invokes this function to notify the generic UART layer of a change in modem status. This is typically used by the UART driver to notify the application layer when it registers for notifications on modem status changes.

Function Prototype `Void cbModemStatus (UartPortHandle port, Int msrVal)`

Arguments

port	This is a mini driver device object, which identifies the UART device.
msrVal	This indicates the modem status register value at the time of the interrupt.

3.3.3 cbRxHandler

Description The lower layer invokes this function to notify the generic UART layer of a new character that has been received. This is the main receive data processing routine where the received character is stored in the circular buffer if no request is pending, or transferred to an active request's buffer directly if a read request is pending.

Function Prototype `Void cbRxHandler (UartPortHandle port, Int c)`

Arguments

port	This is a mini driver device object, which identifies the UART device.
c	This indicates the new character that has been received.

3.3.4 cbTxHandler

Description The lower layer invokes this function to notify the generic UART layer of buffer empty condition. This signals to the upper layer that it can transfer the next character if it has one to be transferred. This is the main transmit data processing routine where the next character to transmit is retrieved and sent. This routine can also be called from the submit routine when a write request is issued if no other write request is pending.

Function Prototype

```
Void  cbTxHandler (UartPortHandle port)
```

Arguments

3.4 Lower-Level UART Layer Interfaces

The generic UART layer communicates with the lower level UART driver through a set of well-defined generic functions. All UART device driver writers who wish to write only the low level UART layer should subscribe to this interface. These functions are described in the sections that follow.

3.4.1 UARTHW_attach

Description The generic UART layer invokes this function when the driver's bind call is called. UART device driver writers who write only the low level drivers as per this interface are expected to plug in the device-specific interrupt handler, enable the interrupt and configure the device to the settings given in the call. This function is called during initialization of the DSP/BIOS RTOS and all restrictions that apply within DSP/BIOS initialization apply here. The function prototype is shown below:

Function Prototype

```
Int  UARTHW_attach (Ptr attrs, Ptr cbArg,
UARTHW_Tcallback *cbFxns)
```

Arguments

attrs	This is the hardware driver specific configuration parameter structure. This is either UARTHW_MCBSP_Params structure or the UARTHW_DSK5402_Params structure based on the hardware it supports.
cbArg	This is a callback argument that is passed back when the low level driver invokes the event handlers of the generic UART layer. This is the device object pointer of the generic UART layer that is used by the generic UART layer in the context of the event handlers.
cbFxns	This is a table of function pointers corresponding to handlers for line status change, modem status change, new character received and transmit buffer empty condition. These handlers were discussed previously.

Return Value This function returns IOM_COMPLETE on success or an error value if some of the settings are not valid.

3.4.2 UARTHW_resetDevice

Description The generic UART layer invokes this function when the UART driver wants to reset the device. This may be in response to a device timeout condition that resulted in the class driver issuing an IOM_CHAN_TIMEDOUT call to the UART mini driver. The function prototype is shown below:

Function Prototype
```
Void  UARTHW_resetDevice ( )
```

Arguments none

Return Value none

3.4.3 UARTHW_getModemStatus

Description This function is invoked by the generic UART layer when the UART driver wants to obtain the modem status either in response to an application request or internally. The function prototype is shown below:

Function Prototype
```
Int  UARTHW_getModemStatus (Char * pmodemStatus )
```

Arguments

pmodemStatus	This holds the address of the location where the modem status value will be placed.

Return Value IOM_COMPLETE on success or an error value if not supported.

3.4.4 UARTHW_setRTS

Description This function is invoked by the generic UART layer when the UART driver wants to set or clear the RTS line. This will be in response to an application request. The function prototype is shown below:

Function Prototype
```
Int  UARTHW_setRTS ( Int rtsVal)
```

Arguments

rtsVal	This indicates whether the RTS line has to be set or cleared based on whether it is one or zero.

Return Value IOM_COMPLETE on success or an error value if not supported.

3.4.5 UARTHW_setDTR

Description This function is invoked by the generic UART layer when the UART driver wants to set or clear the DTR line. This will be in response to an application request. The function prototype is shown below:

Function Prototype
```
Int  UARTHW_setDTR ( Int  dtrVal)
```

Arguments

dtrVal	This indicates whether the DTR line has to be set or cleared based on whether it is one or zero.

Return Value IOM_COMPLETE on success or an error value if not supported.

3.4.6 UARTHW_setBreak

Description This function is invoked by the generic UART layer when the UART driver wants to set or clear the break condition. This will be in response to an application request. The function prototype is shown below:

Function Prototype
```
Int  UARTHW_setBreak ( Int breakVal)
```

Arguments

breakVal	This indicates whether the break condition has to be set or cleared based on whether it is one or zero.

Return Value IOM_COMPLETE on success or an error value if not supported.

3.4.7 UARTHW_txEmpty

Description This function is invoked by the generic UART layer when the UART driver wants to know if the transmit buffer is empty. The function prototype is shown below:

Function Prototype
```
Int  UARTHW_txEmpty ( )
```

Return Value A non-zero return value indicates that transmit buffer is empty and a zero return value indicates it is full.

3.4.8 UARTHW_writeChar

Description This function is invoked by the generic UART layer when the UART driver wants to write a character to the UART device. The function prototype is shown below:

Function Prototype
```
Void  UARTHW_writeChar ( Char c)
```

Arguments

c	The character to write.

Return Value none

3.5 Hardware UART Implementation

This section discusses the functions implemented by the hardware UART. In our case this is derived from the UART implementation on the DSK 5402 for a 16550 based UART. This section also describes the interrupt handling in the lower layer for this implementation.

3.5.1 UARTHW_attach

The DSK 5402 based 16550 UART driver plugs in the interrupt handler for the interrupt vector 17. It then sets up the baud rate, configures the communication parameters and sets up the FIFO for the UART. In our driver, the FIFO is disabled. Interrupts are then enabled for all events of interest from the UART. The flow control settings are also set up. The communication and flow control settings should be provided by the application through the configuration parameters structure. If not provided, a default setting of 115200 baud, 8-bit word size , no parity, 1 stop bit and no flow control is assumed. This driver always returns success from this routine.

3.5.2 UARTHW_resetDevice

The DSK 5402 based 16550 UART driver clears the FIFO through the FIFO control register.

3.5.3 UARTHW_getModemStatus

The DSK 5402 based 16550 UART driver reads the modem status register and returns this value.

3.5.4 UARTHW_setRTS

The DSK 5402 based 16550 UART driver sets the RTS line using the modem control register.

3.5.5 UARTHW_setDTR

The DSK 5402 based 16550 UART driver sets the DTR line using the modem control register.

3.5.6 UARTHW_setBreak

The DSK 5402 based 16550 UART driver sets the break condition using the line control register.

3.5.7 UARTHW_txEmpty

The DSK 5402 based 16550 UART driver detects the condition using the line status register.

3.5.8 UARTHW_writeChar

The DSK 5402 based 16550 UART driver transmits the character through the transmit register.

3.5.9 Hardware UART Interrupt Handling

The DSK 5402 based 16550 UART generates an interrupt for the following conditions:

1. New character received
2. Transmit Buffer Empty
3. Line status changed due to errors in framing or buffer over runs
4. Modem status change.

For all of these conditions, the low-level device specific UART driver handles the generated interrupt and invokes the handler in the generic UART layer with the relevant information. The handlers in the generic UART layer for these events were discussed above.

3.6 Software UART Implementation

This section discusses the functions implemented by the software McBSP based UART. In our case, we will describe a C5402, C5510, C6711, and C6412 implementation of this driver. This section also describes the interrupt handling in the lower layer for this implementation.

3.6.1 Background Information

The biggest challenge in interfacing a synchronous device to an asynchronous signal is not in the transmission but rather in the reception. Transmission is a simple process in terms of the timings of the signals; the serial port can transmit according to its clock and the receiver will correctly decode the signal, as long as the start and stop bits are appropriately placed and the sampling rate is appropriate. The receiver timings are more complicated. The asynchronous signal, by nature, can be received at any time, and most likely will not be aligned with the serial port clock. Also, there can be slight differences in the baud rate compared to the sample rate of the serial port, causing the received data to “slide”. Because of these issues, the serial port receive channel and software must be setup appropriately to recognize these constraints and to work around them.

The length of the packets are start bits + data bits + stop bits. There is 1 start bit and 1-stop bits.

For the McBSP to be used for the purposes of a software UART, it must be set up for dual phase frames, with the first phase having 16-bit words and the second having 8 bit words. For the purposes of calculating the frame lengths, the driver is always set up to have 8 bit word size, no parity and two half stop bits while transmitting and one half stop bit while receiving.

For the McBSP to be used for the purposes of a software UART for transmission, the UART must be able to send half stop bits. Therefore, the McBSP transmit port is set for dual phase frames, with the first phase having 16 bit words and the second phase having 8-bit words. For the purposes of calculating the frame lengths, the driver is always set up to have 8-bit word size, no parity and two half stop bits. The data transmit (DX) pin of the DSP is tied to the transmit data line of the interface.

From Figure 3 we can see that the asynchronous signal line is always high unless a data packet has been sent across. When a packet is sent, the start bit is sent first, so the signal will go low. By tying the receive data line to the data receive (DR) and frame sync (FSR) pins of the McBSP receive channel, we can trigger the McBSP to start receiving the packet whenever the line goes low. To prevent the McBSP from re-triggering, it is set to ignore all frame syncs during the receive packet.

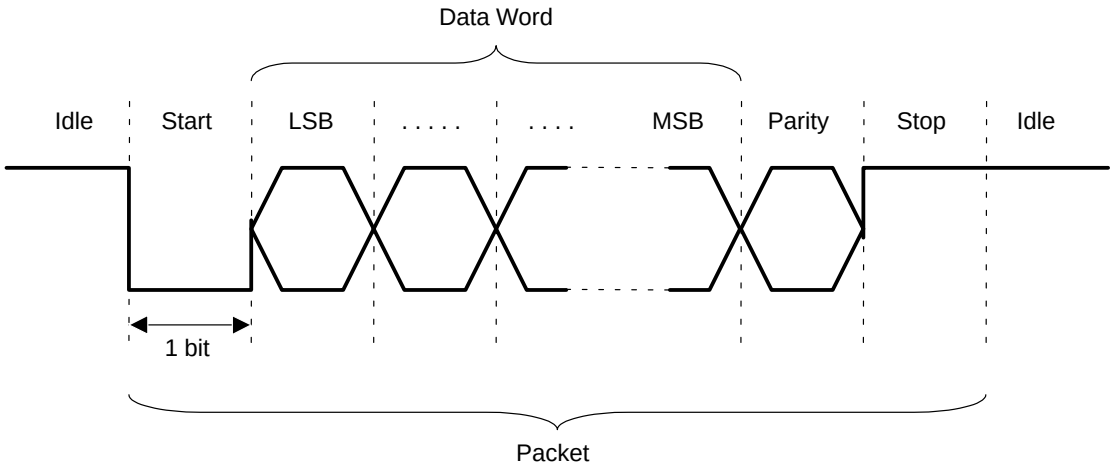

Figure 3. UART Data Packet

During decoding, the center of each over-sampled bit is checked. Only the first half of the stop bit is received and checked, which gives more flexibility in the sampling rate, as will be seen below. Therefore, the McBSP receive port is set to have dual phase frames. The word size of the first phase is 16 bit words and the second phase is one 8 bit word. See Figure 4 for an example of how the McBSP receive frame aligns with the data packet.

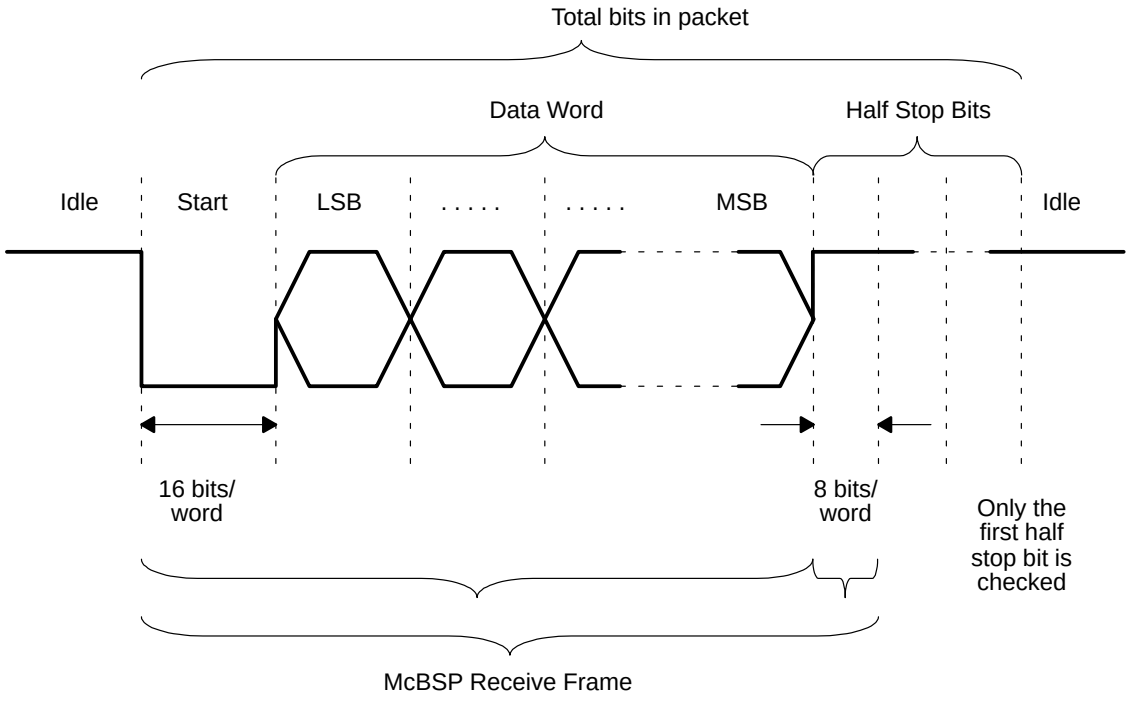

Figure 4. McBSP Receive Frame Structure

The sampling rate of the McBSP is critical to the correct operation of the software UART. The McBSP will ignore all subsequent frame syncs during the reception of the frame we have defined above. To get the maximum data rate, it must be able to detect the next start bit, which could immediately follow the stop bit. The frames syncs and receive data are latched on the falling edges of the serial port clock. For a frame sync to be detected, the signal must be high for at least one clock cycle before it goes low again. This resets the frame sync logic. Therefore, the McBSP must be finished reading in the first data packet before the transition from the stop bit to the next start bit occurs.

In an ideal case, the clock edges of the serial port line up with the bit edges of the data packet, there are exactly 16 clock periods for each bit in the packet, and the offset between the beginning of the start bit and the falling edge of the serial port clock is minimal. See Figure 5 for an example of this timing.

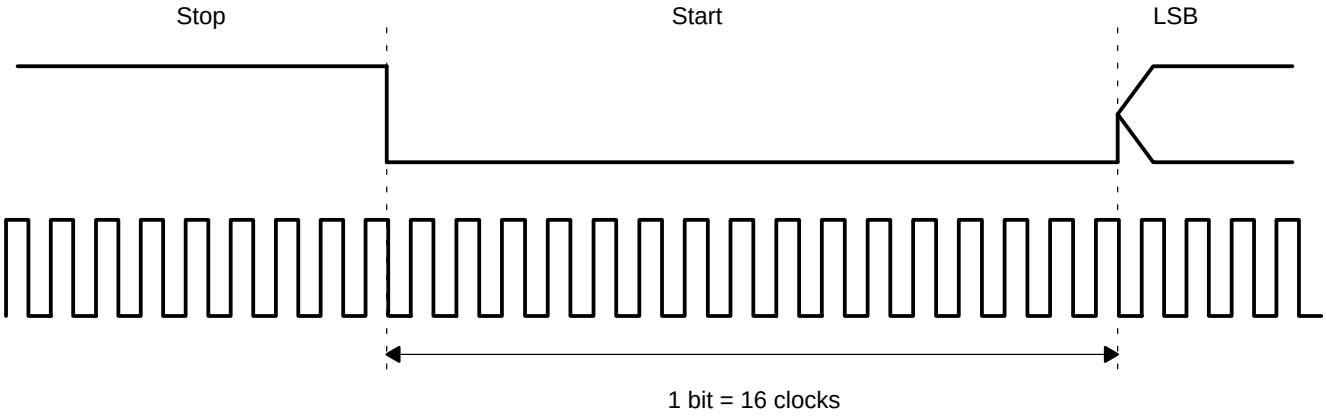

Figure 5. Timing of a Signal-to-Serial Port Clock

The DMA setup involves opening the transmit and receive channels. In the case of the 5402, we use the Auto Buffering Mode. In the case of our driver, we use post increment mode and the driver writer may choose to change that if necessary.

The receive DMA channel is set up to transfer data from McBSP receive register to CPU memory. The transmit DMA channel is set up to transfer data from CPU memory to McBSP transmit register. The transfers are synchronized with McBSP receive and transmit events, respectively. The driver plugs the interrupt handlers for the interrupt vectors corresponding to the DMA channels to handle the DMA completion and error events. This driver validates configuration parameters given by the application and returns success from this routine upon successfully setting up and starting the DMA channels.

On the receive side, the McBSP is constantly running and always has the DMA enabled. The DMA interrupts the DSP when a packet is read in and the DSP then decodes the over-sampled packet and performs error checking. Note that the receive DMA channel only moves data into the buffer when the McBSP gets new frame syncs, which only occurs when a new packet is received. Because the DMA buffer has 2 halves, the receive data is double buffered.

The DMA transmit buffer has 2 halves, allowing double buffering of the transmit data. As long as there is an empty half, more transmit data can be written into the buffer. If the DMA transmit channel generates an interrupt but there is still another word in the buffer to transmit, the DMA need not be disabled. However, if the DMA has just transmitted the last valid word in the buffer, the DMA must be halted and then restarted when a new transmission is desired.

3.7 Software Line Driver

The software UART I/O mini-driver was tested using a line driver daughter card connected to the 80-pin peripheral interface on the board. The Peripheral interface has multiple singles brought out. The signals of interest for the software UART are the McBSP data transmit (DX) pin, the McBSP data receive (DR) pin, and the McBSP frame sync receive (FSR) pin. These are the pins from which asynchronous data will be transmitted and received. Figure 6 illustrates the line driver card used to validate the driver. Note from the schematic the both the McBSP data receive data pin and the McBSP Receive Frame Sync are tied together. By tying the receive data line to the data receive (DR) and frame sync (FSR) pins of the McBSP receive channel, we can trigger the McBSP to start receiving the packet whenever the line goes low as described in the previous section.

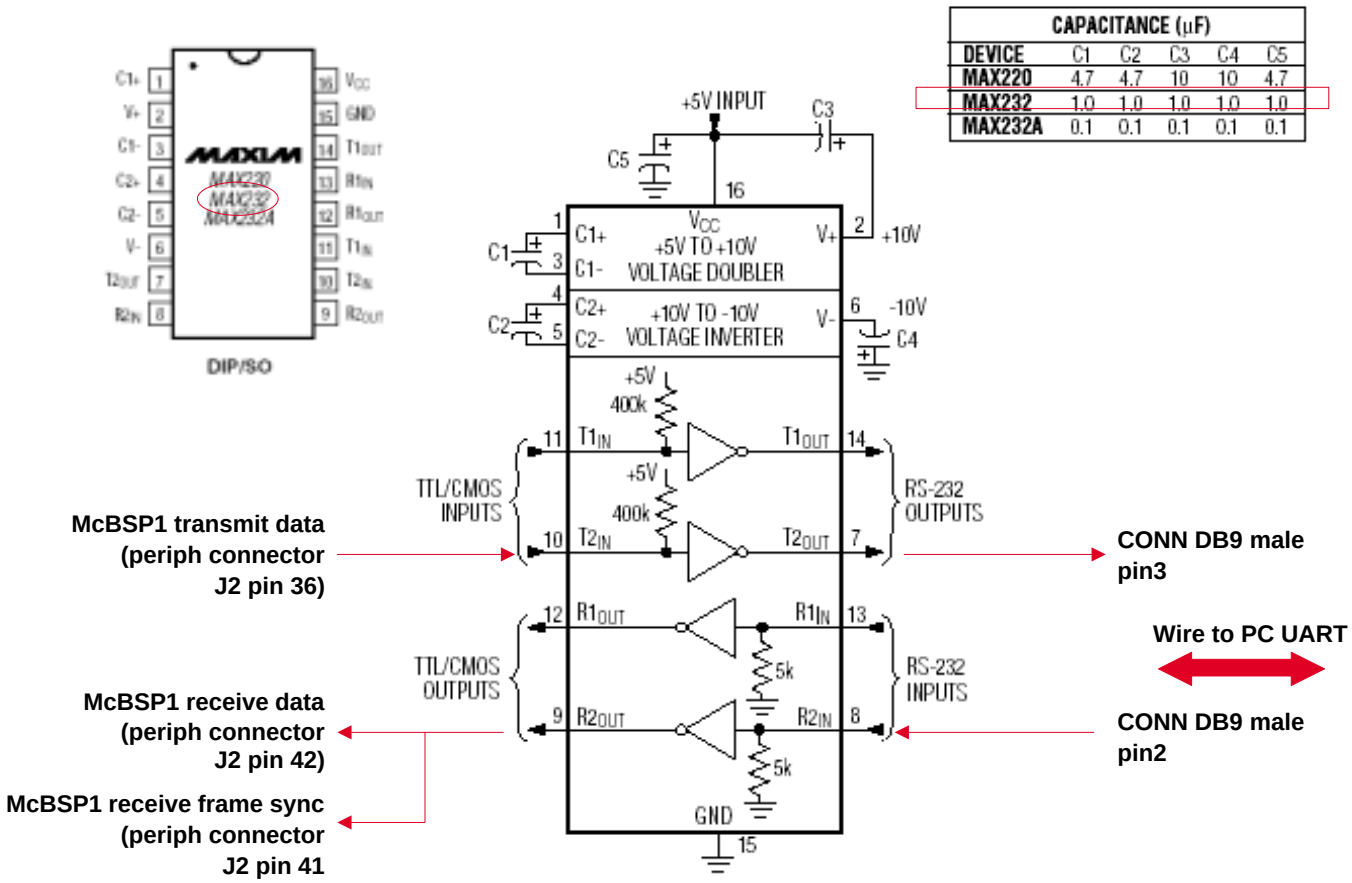

Figure 6. Line Driver Daughtercard Schematic

3.7.1 UARTHW_attach

The McBSP based software UART implementation must setup the McBSP. The application should supply the configuration required through the configuration structure. If none is provided, a default structure is assumed. The structure will vary depending on the ISA/CPU supported. Tan example of the default structure can be found in one of the following files:

- Uarthw_c55xx_mcbsp.c
- Uarthw_C54xx_mcbsp.c
- Uarthw_c6x1x_mcbsp.c

The UARTHW_attatch function makes a call to UARTHW_MCBSP_start which starts the McBSP and the DMA, sets up the transmit and receive buffers, and plugs-in the appropriate interrupt service routine (ISR). The UARTHW_MCBSP_setup function will vary depending on ISA.

3.7.2 UARTHW_resetDevice

The McBSP based software UART implementation has a dummy implementation for this function.

3.7.3 UARTHW_getModemStatus

The McBSP based software UART implementation returns an error value, as the software UART does not support this function.

3.7.4 UARTHW_setRTS

The McBSP based software UART implementation does not support this function and returns an error.

3.7.5 UARTHW_setDTR

The McBSP based software UART implementation does not support this function and returns an error.

3.7.6 UARTHW_setBreak

The McBSP based software UART implementation records the break condition status and transmits all zeroes on the McBSP line if a break condition is desired and transmits all ones on the McBSP line if the break condition is turned off.

3.7.7 UARTHW_txEmpty

The McBSP based software UART implementation bases this decision on whether it has a character to be transferred out by the DMA. If so, it returns 0. Otherwise, it returns a non-zero value.

3.7.8 UARTHW_writeChar

The McBSP based software UART implementation stores the character to be transmitted in an internal buffer. When the next transmit DMA interrupt occurs, the character is encoded and transmitted over the McBSP channel.

3.8 Software UART Interrupt Handling

The McBSP based software UART implementation handles two interrupts: one for the DMA transmit channel and one for the DMA receive channel. The transmit channel interrupt is triggered after each half the buffer has been transmitted. When no characters are being sent, the buffer contains all ones, which maintains the line high. When a character needs to be sent, it is encoded and the corresponding data is put in the transmit buffer which is transmitted over the McBSP line. In any case, the transmit DMA channel is always active and transmitting the data it has in its buffer.

The receive DMA channel interrupt is triggered whenever a new character is received, which in turn triggers a receive frame sync to be generated. Upon receipt of the frame sync, the McBSP channel samples the receive line at the set frequency and data collected is transferred through the DMA into the internal receive buffer. When the buffer is filled up (either half can be filled up), a receive channel interrupt is triggered. Upon receipt of this interrupt, the data in the receive buffer is decoded and a character is constructed out of the data.

From the respective interrupt handlers, the high level UART driver's transmit empty or receive buffer full handlers are invoked and data is transmitted or buffered as in the case of a normal hardware based UART.

3.8.1 Constraints

The DMA is used in the implementation for each of the MCBSP based software UARTs. This introduces the following constraints:

- For C6x11, the .bss section must reside in a non-cached region of memory. This is needed to maintain coherency between the CPU and the EDMA that is interfacing with the McBSP. If the application requires the .bss memory section to be in cached memory, the uarthw_mcbsp.c code can be modified to include CSL cache calls to keep the cache coherent. The code can also be modified to specify an alternate section for the rxBuffer and txBuffer using '#pragma DATA_SECTION'. You will need to modify the linker .cmd file to include this new section.
- For C5402, the .bss section must reside at an address less than 0x4000. This is required since the DMAs on the C5402 can only address data memory in this area. If placing .bss at this address is a problem, the code in uarthw_mcbsp.c can be modified to specify an alternate section for rxBuffer and txBuffer using '#pragma DATA_SECTION'. You will need to modify the linker .cmd file to include this new section.
- For C55x, the .bss section must reside in on-chip DARAM. This is required since the DMA is hard-coded to use the DARAM port. You can modify the code in uart_mcbsp.c to change the DMA port type. The code can also be modified to specify an alternative section for rxBuffer and txBuffer using '#pragma DATA_SECTION'. You will need to modify the linker .cmd file to include this new section.

4 References

1. *TMS320C6000 McBSP: UART* (SPRA633A)
2. *Implementing a Software UART on the TMS320C54x with the McBSP and DMA* (SPRA661A)

Appendix A Device Driver Data Sheet

A.1 Device Driver Library Name (Generic Interface)

- uartmd.l54 (near mode) and uartmd.l54f (far mode) for the TMS320C54xx DSPs.
- uratmd.l55 (small model) and uartmd.l55l (large model) for the TMS320C55xx DSPs.
- uartmd.l62 for the TMS320C621x and TMS320C671x DSPs.
- uartmd.l64 for the TMS320C641x DSPs.

A.2 Device Driver Library Name (Hardware Specific)

- uarthw_c5402_mcbsp.l54 (near mode) and uarthw_c5402_mcbsp.l54f (far mode) for theTMS320C5402 Software UART.
- uarthw_c5509_mcbsp.l55 (small model) and uarthw_c5509_mcbsp.l55l (large model) for the TMS320C5509 software UART.
- uarthw_c5510_mcbsp.l55 (small model) and uarthw_c5510_mcbsp.l55l (large model) for the TMS320C5510 Software UART.
- uarthw_c6x1x_mcbsp.l62 for the TMS320C671x Software UART
- uarthw_c6x1x_mcbsp.l64 for the TMS320C641x Software UART
- uarthw_dsk5402.l54 (near mode) and uarthw_dsk5402.l54f (far mode) for the TMS320C5402 Hardware UART.

A.3 DSP/BIOS Modules Used (Generic Interface)

- HWI – Hardware Interrupt Manager
- QUE – Queue Manager
- IOM – I/O Manager
- ATM – Atomic Manager

A.4 DSP/BIOS Modules Used (Hardware Specific)

- uarthw_c5402_mcbsp
  - HWI – Hardware Interrupt Manager
  - IOM – I/O Manager
- uarthw_c5509_mcbsp and uarthw_c5510_mcbsp
  - HWI – Hardware Interrupt Manager
  - IOM – I/O Manager
- uarthw_c6x1x_mcbsp
  - HWI – Hardware Interrupt Manager
  - IOM – I/O Manager
- uarthw_dsk5402

  - HWI – Hardware Interrupt Manager
  - IOM – I/O Manager

A.5 DSP/BIOS Objects Used (Generic Interface)

QUE_Obj

A.6 DSP/BIOS Objects Used (Hardware Specific)

- uarthw_c5402_mcbsp – none
- uarthw_c5509_mcbsp – none
- uarthw_c5510_mcbsp – none
- uarthw_c6x1x_mcbsp – none
- uarthw_dsk5402 – none

A.7 CSL Modules Used (Generic Interface)

None

A.8 CSL Modules Used (Hardware Specific)

- uarthw_c5402_mcbsp
  - McBSP module
  - DMA module
  - IRQ module
- uarthw_c5509_mcbsp and uarthw_c5510_mcbsp
  - McBSP module
  - DMA module
  - IRQ module
- uarthw_c6x1x_mcbsp
  - McBSP module
  - EDMA module
  - IRQ module
- uarthw_dsk5402
  - IRQ module

A.9 CPU Interrupts Used (Hardware Specific)

None

A.10 CPU Interrupts Used (Generic Interface)

- uarthw_c5402_mcbsp

  - DMA interrupt
- uarthw_c5510_mcbsp
  - DMA interrupt
- uarthw_c6x1x_mcbsp
  - EDMA interrupt
- uarthw_dsk5402
  - UART interrupt

A.11 Peripherals Used (Generic Interface)

None

A.12 Peripherals Used (Hardware Specific)

- uarthw_c5402_mcbsp
  - McBSP
  - DMA
- uarthw_c5509_mcbsp and uarthw_c5510_mcbsp
  - McBSP
  - DMA
- uarthw_c6x1x_mcbsp
  - McBSP
  - EDMA
- uarthw_dsk5402
  - UART

A.13 Interrupt Disable Time

Maximum time that hardware interrupts can be disabled by the driver:

- 217 cycles – uarthw_c5402_mcbsp
- 153 cycles – uarthw_c5509_mcbsp and uarthw_c5510_mcbsp
- 156 cycles – uarthw_c6x1x_mcbsp
- 217 cycles – uarthw_dsk5402

This measurement is taken using the compiler option –O3.

A.14 Memory Usage

Table A–1. Uarthw_dsk5402 Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	369 (16-bit words)
DATA	6 (16-bit words)	7 (16-bit words)

Table A–2. Uarthw_c5402_mcbsp Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	912 (16-bit words)
DATA	98 (16-bit words)	10 (16-bit words)

Table A–3. Uarthw_c5509_mcbsp and Uarthw_c5510_mcbsp Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	1790 (8-bit bytes)
DATA	64 (8-bit bytes)	60 (8-bit bytes)

Table A–4. Uarthw_c6x1x_mcbsp62 Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	3392 (8-bit bytes)
DATA	152 (8-bit bytes)	28 (8-bit bytes)

Table A–5. Uarthw_c6x1x_mcbsp64 Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	2816 (8-bit bytes)
DATA	156 (8-bit bytes)	28 (8-bit bytes)

NOTE: This data was gathered using the sectti command utility.
Uninitialized data: .bss
Initialized data: .cinit + .const
Initialized code: .text + .text:init

Table A–6. Uartmd Device Driver Memory Usage on 54 Platform

	Uninitialized memory	Initialized memory
CODE	—	1088 (words)
DATA	108 (16-bit words)	16 (16-bit words)

Table A–7. Uartmd Device Driver Memory Usage on 55 Platform

	Uninitialized memory	Initialized memory
CODE	—	1386 (8-bit bytes)
DATA	248 (8-bit bytes)	56 (8-bit bytes)

Table A–8. Uartmd Device Driver Memory Usage on 62 Platform

	Uninitialized memory	Initialized memory
CODE	—	3520 (8-bit bytes)
DATA	224 (8-bit bytes)	144 (8-bit bytes)

Table A–9. Uartmd Device Driver Memory Usage on 64 Platform

	Uninitialized memory	Initialized memory
CODE	—	2880 (8-bit bytes)
DATA	224 (8-bit bytes)	144 (8-bit bytes)

NOTE: This data was gathered using the sectti command utility.
Uninitialized data: .bss
Initialized data: .cinit + .const
Initialized code: .text + .text:init

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265