DSP/BIOS Driver Developer's Guide

DDK version 1.20

Literature Number: SPRU616A
August 2005

TEXAS INSTRUMENTS

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 that 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

Preface

Read This First

About This Manual

DSP/BIOS provides a recommended structure for the development and integration of drivers. This structure is described in this manual. For overview information about DSP/BIOS, see the DSP/BIOS online help and the *TMS320 DSP/BIOS User's Guide* (SPRU423).

Notational Conventions

This document uses the following conventions:

- In file paths, *BIOS_INSTALL_DIR* is the folder where you installed DSP/BIOS.
- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
Void copy(HST_Obj *input, HST_Obj *output)
{
    PIP_Obj     *in, *out;
    Uns         *src, *dst;
    Uns         size;
}
```

Software Version Requirements

This manual is intended for use with version 1.20 of the Driver Developer's Kit (DDK). You must have installed DSP/BIOS version 5.20 or higher to use this version of the DDK.

Related Documentation From Texas Instruments

The following books describe TMS320 devices and related support tools. You can find these books on the Texas Instruments web site at www.ti.com. Search for the literature number to find the book you want.

TMS320 DSP/BIOS User's Guide (literature number SPRU423) provides an overview and description of the DSP/BIOS real-time operating system.

TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide (literature number SPRU403) describes DSP/BIOS API functions, which are alphabetized by name. The API Reference Guide is the companion to this user's guide.

TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide (literature number SPRU404) describes DSP/BIOS API functions, which are alphabetized by name. The API Reference Guide is the companion to this user's guide.

TMS320C28x DSP/BIOS Application Programming Interface (API) Reference Guide (literature number SPRU625) describes DSP/BIOS API functions, which are alphabetized by name. The API Reference Guide is the companion to this user's guide.

TMS320C54x Chip Support Library API Reference Guide (literature number SPRU420) describes the Chip Support Library (CSL) API modules and functions.

TMS320C55x Chip Support Library API Reference Guide (literature number SPRU433) describes the Chip Support Library (CSL) API modules and functions.

TMS320C6000 Chip Support Library API Reference Guide (literature number SPRU401) describes the Chip Support Library (CSL) API modules and functions.

Related Documentation

You can use the following books to supplement this reference guide:

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

Programming in C, Kochan, Steve G., Hayden Book Company

Programming Embedded Systems in C and C++, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

Real-Time Systems, by Jane W. S. Liu, published by Prentice Hall; ISBN: 013099651, June 2000

Principles of Concurrent and Distributed Programming (Prentice Hall International Series in Computer Science), by M. Ben-Ari, published by Prentice Hall; ISBN: 013711821X, May 1990

American National Standard for Information Systems-Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C); (out of print)

Trademarks

Contents

1 About the Device Driver Developer's Kit 1-1
This chapter provides an overview of the DSP/BIOS Device Driver Developer's Kit.
1.1 Using this Book 1-2
1.2 Installing the DDK 1-3
1.3 DSP/BIOS Driver Development Kit Overview 1-3
1.4 Kit Contents and Organization 1-6
1.5 Using the Example Applications 1-8

2 DSP/BIOS Device Driver Architecture and Usage 2-1
This chapter describes the architecture of the DSP/BIOS device driver strategy.
2.1 Two-Level Device Driver Model 2-2
2.2 Driver Data Flow 2-5
2.3 Class Driver Overview 2-11

3 Using DSP/BIOS Device Drivers 3-1
This chapter describes how the DSP/BIOS device driver model can be configured for use with SIO, PIP, or GIO objects.
3.1 Registering the Mini-Driver 3-2
3.2 Configuring the DIO Class Driver (for SIO) 3-4
3.3 Configuring the PIO Class Driver (for PIP) 3-7
3.4 Configuring Applications to Use the GIO Class Driver 3-10

4 GIO Class Driver 4-1
This chapter describes the GIO module, which provides a class driver.
4.1 About the GIO Module 4-2
4.2 Implementation Details for GIO 4-3
4.3 Error Handling 4-5
4.4 Extending the GIO API 4-5

5 Developing a Mini-Driver Step-by-Step 5-1
This chapter shows a step-by-step process for developing a mini-driver.
5.1 Mini-Driver Design and Implementation 5-2
5.2 Examining the 'C5402 SBS Mini-Driver Example 5-6

A IOM Interface A-1
This appendix provides reference details for the IOM (I/O Mini-driver) interface.
A.1 Mini-Driver Interface Overview A-2

B PIO Adapter B-1
This appendix provides reference details for the PIO (Pipe I/O) adapter interface.
B.1 PIO Adapter Interface Overview B-2

C Porting from the LIO to IOM Model C-1
This appendix compares the LIO and IOM models and explains how users of the LIO device driver model can migrate applications to use the updated IOM device driver model.
C.1 Comparing the LIO and IOM Models C-2
C.2 Migrating an LIO Application to Use an IOM Mini-Driver C-4
C.3 Migrating an LIO Controller to an IOM Mini-Driver C-5

D The ASYNC Extension to the GIO API. D-1
This appendix describes the ASYNC extension to the GIO API.
D.1 ASYNC Module Overview D-2

E Glossary. E-1
This appendix provides definitions for terms related to device drivers.

Figures

1-1 Device Developer's Kit Folder Organization 1-7
2-1 Application Architecture Showing Device Driver Components 2-3
2-2 Mini-Driver Operation Flow 2-6
2-3 Call Flow for Blocking GIO_read and GIO_write 2-9
3-1 GIO Module Configuration 3-10
4-1 GIO Class Driver Interface 4-3
5-1 PIO Adapter Buffer Flow B-4
5-2 Call Flow for ASYNC_read and ASYNC_write D-3

Tables

5-1 Component Comparison for LIO and IOM C-2
5-2 Function Comparison for LIO and IOM C-3

Chapter 1

About the Device Driver Developer's Kit

This chapter provides an overview of the DSP/BIOS Device Driver Developer's Kit.

Topic **Page**

1.1 Using this Book 1–2
1.2 Installing the DDK 1–3
1.3 DSP/BIOS Driver Development Kit Overview 1–3
1.4 Kit Contents and Organization 1–6
1.5 Using the Example Applications 1–8

1.1 Using this Book

Depending on whether you are an application developer/integrator or a driver developer, different sections of this book will be of interest to you.

1.1.1 For Application Developers and Integrators

If you are an application developer who will integrate drivers developed by others, read the following sections of this manual:

- ❏ **Chapter 1.** This chapter provides an overview of the Driver Developer's Kit (DDK). In particular, note the sections that describe the functional device drivers included in the kit and how to build the example applications that use the drivers.
- ❏ **Chapter 2.** All sections are of interest to application developers.
- ❏ **Chapter 3.** Read this to learn how to configure and use the DIO adapter, PIO adapter, and GIO class driver.
- ❏ **Chapter 4.** Read this to learn about the GIO class driver module.
- ❏ **Appendix B.** Read this to learn about the PIP class driver adapter.
- ❏ **Appendix C.** If you have used the LIO driver model before, read Sections C.1 and C.2.
- ❏ **Appendix D.** Read this appendix if you want to use a mini-driver asynchronously (without blocking—for example, with SWI threads).
- ❏ **Appendix E.** See this glossary if you encounter unfamiliar terms.

1.1.2 For Driver Developers

If you are a driver developer, read the following sections of this manual:

- ❏ **Chapter 1.** This chapter provides an overview of the Driver Developer's Kit (DDK).
- ❏ **Chapter 2.** Sections 2.1 through 2.2 are of interest to driver developers.
- ❏ **Chapter 5.** Read this chapter for step-by-step instructions to create a mini-driver and a detailed overview of a sample mini-driver.
- ❏ **Appendix A.** Read this for details about the mini-driver specification.
- ❏ **Appendix C.** If you have used the LIO driver model before, read Sections C.1 and C.3.
- ❏ **Appendix E.** See this glossary if you encounter unfamiliar terms.

1.2 Installing the DDK

Note:

This manual is intended for use with version 1.20 of the Driver Developer's Kit (DDK). You must have installed DSP/BIOS version 5.20 or higher to use this version of the DDK.

Install the DDK by running the setup.exe file you downloaded. Answer the prompts as necessary.

After you install the DDK, be sure to define the environment variables DDK_INSTALL_DIR and CSL_INSTALL_DIR as described in the SetupGuide.html file. Also, define the BSL_INSTALL_DIR environment variable if you are using a board that requires Spectrum Digital's Board Support Library. You should have the BIOS_INSTALL_DIR environment variable defined as described in the SetupGuide.html file for DSP/BIOS.

1.3 DSP/BIOS Driver Development Kit Overview

The DSP/BIOS Driver Developer's Kit (DDK) is designed to simplify the development of device drivers for peripherals present on TMS320 DSPs and their associated evaluation boards. To achieve this goal, the DDK provides:

- ❏ Fully functional device drivers for many TMS320 DSP peripherals
- ❏ A documented driver model that standardizes the methodology for developing drivers
- ❏ A set of reusable driver modules that eliminate the need to develop all driver code from scratch

The DDK is complementary to the Chip Support Library (CSL), which is provided for each TMS320 DSP. The CSL provides low-level hardware abstractions of DSP peripheral registers and initialization functions. The drivers in the DDK make full use of the CSL for peripheral initialization and control.

The following subsections provide an overview of each of the DDK's elements in more detail to give a greater understanding of their purpose.

1.3.1 Functional Device Drivers

The DDK offers a number of complete device drivers for peripherals such as codecs, UARTs, and serial ports. These drivers are provided in both binary and source code form. Extensive documentation is provided for each driver, including memory requirements and a design overview that explains how the driver works.

DDK drivers are targeted specifically at on-chip DSP peripherals or external peripherals on DSP evaluation boards, such as audio codecs. Each driver is thoroughly tested on a specific board to validate correct real-time behavior. These drivers enable a developer to immediately begin using a supported peripheral with no driver development effort at all. The availability of source code makes it straightforward to port the driver to custom board configurations using the same peripherals.

1.3.2 Documented Driver Model

The DDK defines a standard driver model and set of APIs for developing drivers. To simplify driver development, the driver model breaks a driver into two components:

- **Class driver.** The upper layer is called a “class driver”. The class driver is device-independent and performs functions such as buffer management and application synchronization. A class driver typically includes both an API interface used by the application and an adapter layer that acts as an interface between the application calls and the mini-driver.
- **Mini-driver.** The lower layer is called the “mini-driver.” It handles all the device-specific control and initialization. All mini-drivers conform to a standard interface called the IOM (I/O Mini-driver) interface. This was designed after reviewing a wide range of devices including serial ports, video ports, flash cards, UARTs, and PCI controllers.

The layered driver model defined by the DDK provides several benefits to driver developers:

- A driver developer only has to learn a single mini-driver API to implement drivers for a full-range of DSP peripherals.
- A mini-driver is simpler to implement than a full driver, because the TI-developed class driver handles the common buffer management and synchronization activities.

This layered driver model also provides several benefits to application integrators:

- The standard mini-driver API allows class drivers to work with mini-drivers.
- A mini-driver developed for a codec can, for example, be used with any of the pre-defined DSP/BIOS I/O modules such as SIO or PIP. There is no need to rewrite a driver because a different I/O API is desired.
- If a developer wants to implement an alternative set of I/O APIs to those provided by DSP/BIOS, they can implement a new class driver or extend an existing class driver.

1.3.3 Reusable Class Driver Modules

As discussed in the previous sections, the driver model inherently enables modules to be easily reused. The DDK provides three class drivers that can be used with any mini-driver designed according to the model:

- **SIO/DIO.** The DIO adapter allows the DSP/BIOS SIO module to be used with a mini-driver. The combination of these two modules is a class driver. Rather than calling DIO APIs, the application calls SIO functions that use the DIO adapter internally.
- **PIP/PIO.** The PIO adapter allows the DSP/BIOS PIP module to be used with a mini-driver. The combination of these two modules is a class driver. The application calls both PIP and PIO API functions when using this class driver.
- **GIO.** The GIO module implements a set of I/O APIs that may be used with DSP/BIOS to interface to a mini-driver. The GIO module on its own is a class driver option.

The GIO class driver is designed to support easy extension of I/O APIs to meet the needs of specialized devices. For example, the ASYNC module provided in the DDK is an example extension designed for asynchronous data processing. In addition, specialized APIs—such as frame video APIs—can be easily be created.

In addition to the class drivers, a selection of mini-drivers is provided that clearly illustrate how to develop the device-specific code for a variety of peripherals. The existing mini-drivers can act as an initial template for developing support for a new peripheral.

To simplify the development of codec and data converter mini-drivers, the DDK provides a generic DMA-McBSP driver. This module can be re-used

as part of any codec mini-driver. As a result, the developer only need implement a very small amount of codec initialization and control code to generate a new driver. All of the audio codec mini-drivers that are included in the DDK use this approach. See the individual application notes provided for these mini-drivers for more details.

All the modules available through the DDK are provided in source-code form and may be customized if desired.

1.4 Kit Contents and Organization

The Device Developer's Kit (DDK) folder tree contains a number of mini-drivers and example applications. For each mini-driver, source files, a CCStudio project, and pre-built libraries are provided.

The following HTML files are provided to describe the DDK and its contents:

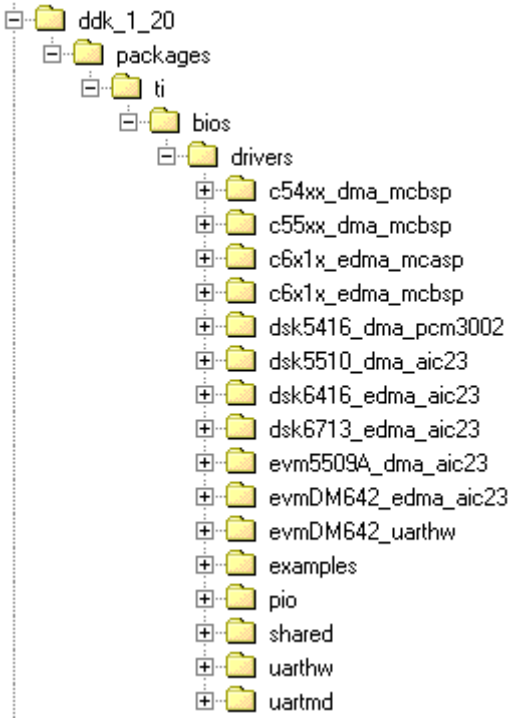
- **SetupGuide.html.** Contains setup instructions.
- **index_ddk_manuals.html.** Provides links to the various PDF documents in the kit.
- **release_notes.html.** Contains product release notes.

You can further explore the DDK by examining the folder tree. We recommend that you retain the provided structure for your development.

All files associated with a given driver are contained within a subfolder for that driver. This is a change from previous versions of the DDK, in which common files were stored in a common location and shared by drivers. The new folder structure is intended to allow for updates to drivers and the addition of new drivers without affecting other drivers.

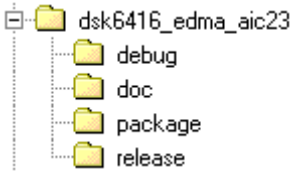
The DDK installation has a top-level folder called ddk_1_20. It is located where you choose to install it. Figure 1-1 shows the main folder organization of the DDK.

Figure 1-1. Device Developer's Kit Folder Organization

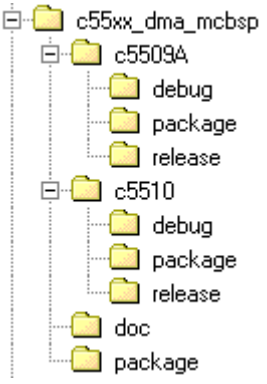


The individual drivers are stored in folders with names like c55xx_dma_mcbsp and dsk6713_edma_aic23. Within these folders, the file structure is similar to one of the following depending on whether one platform or multiple platforms can use this driver:

One Platform for driver



Multiple platforms



The top-level folder for each driver (and the platform-named folders for drivers with multiple platforms) contain the following files:

- **Source files.** These are typically a .c and .h file named to match the driver. Some drivers use additional source files.
- **CCStudio project file (*.pjt).** This contains settings for the debug and release versions to build a library file.
- **Package files.** You can ignore package files and folders. They are included to support packaging, versioning and RTSC build systems.

The debug and release folders contain pre-built libraries for the various drivers. Each driver folder contains a “doc” folder with an Adobe Acrobat *.pdf file that describes the driver.

In addition to the driver folders, the DDK contains the following additional folders:

- **shared.** Contains the aic23.h file, which supports the aic23 codec and is used by several drivers. Also contains the async.h file, which is used by the ASYNC module.
- **examples.** This tree contains source, project, and configuration scripts for several applications that use the provided mini-drivers. Pre-built executables are not provided. See the readme files and source code comments for descriptions of the examples.
- **pio.** Contains the PIO adapter API source, projects, and libraries.

1.5 Using the Example Applications

The DDK includes some example applications that integrate mini-drivers in various ways. These files are located in the examples folder tree. You can open the project file (*.pjt) for an example in Code Composer Studio and examine the source code and the DSP/BIOS configuration. Projects and configurations are provided for several mini-drivers for each example.

- **audio folder**
  - **tsk_audio.** This example demonstrates how an application can use a codec mini-driver via the SIO module in TSK threads. This is a loopback application. Audio is read from an input SIO, then sent back out on an output SIO. This application is configured to use the DIO adapter. This example is provided for a number of codec drivers. When you examine the configuration, notice the TSK, UDEV, and DIO objects. The SIO objects are created dynamically at run-time.

- **swi_audio.** This example demonstrates how an application can use a codec mini-driver via the SIO module in SWI threads. This is a loopback application. Audio is read from an input SIO, then sent back out on an output SIO. This application is configured to use the DIO adapter in callback mode. This example is provided for a number of codec drivers. When you examine the configuration, notice the SWI, UDEV, and DIO objects. The SIO objects are created dynamically at run-time.

- **pip_audio.** This example demonstrates how an application can use a codec mini-driver via the PIP module in SWI threads. This is a loopback application. Audio is read from an input PIP, then sent back out on an output PIP. This application uses the PIO adapter. This example is provided for a number of codec drivers. When you examine the configuration, notice the PIP, SWI, and UDEV objects.

❑ **uart folder**

- **uarttest.** This example demonstrates synchronous (blocking) use of the mini-drivers for the UART. It uses TSK threads and the GIO class driver API. When you examine the configuration, notice the TSK and UDEV objects. The GIO objects are created dynamically at run-time.

Additional test examples may be added to the DDK in the future to support additional types of devices

Chapter 2

DSP/BIOS Device Driver Architecture and Usage

This chapter describes the architecture of the DSP/BIOS device driver strategy.

Topic **Page**

2.1 Two-Level Device Driver Model 2–2
2.2 Driver Data Flow 2–5
2.3 Class Driver Overview. 2–11

2.1 Two-Level Device Driver Model

As DSP real-time systems become more complex and new technologies emerge, the variety and number of peripheral devices grows. Writing and porting device drivers for such peripheral devices has been an innately hardware- and OS-dependent task. Sometimes, it has been a difficult undertaking given DSP system constraints such as memory footprint, response time, and power management.

Device driver writers have benefited from a model that divides device driver functionally into separate hardware-independent and hardware-dependent layers. Using common interfaces for each layer allows software reuse of major portions of similar device drivers and simplifies the driver development process.

Such partitioning of device driver software is sometimes referred to as the “class/mini-driver model”.

- **Class driver.** The class driver typically provides serialization and synchronization of multi-threaded I/O requests. In addition, it handles device instance management. In typical real-time systems, only a few class drivers, at most, are needed to represent the types of devices used to the application, including block I/O, character I/O, and video.
- **Mini-driver.** The class driver uses a device-specific mini-driver to operate on a particular device on behalf of the application software.

The mini-driver writer must have the ability to efficiently represent a particular device to the class driver. For example, a video display device may have on-board frame buffer memory that an application may need to allocate from in order to perform the desired I/O operation efficiently. Also, a video class driver may need to represent an I/O request as a set of disjointed memory buffers, such as RGB or YUV components, so the lower level mini-driver can efficiently interact with the video hardware.

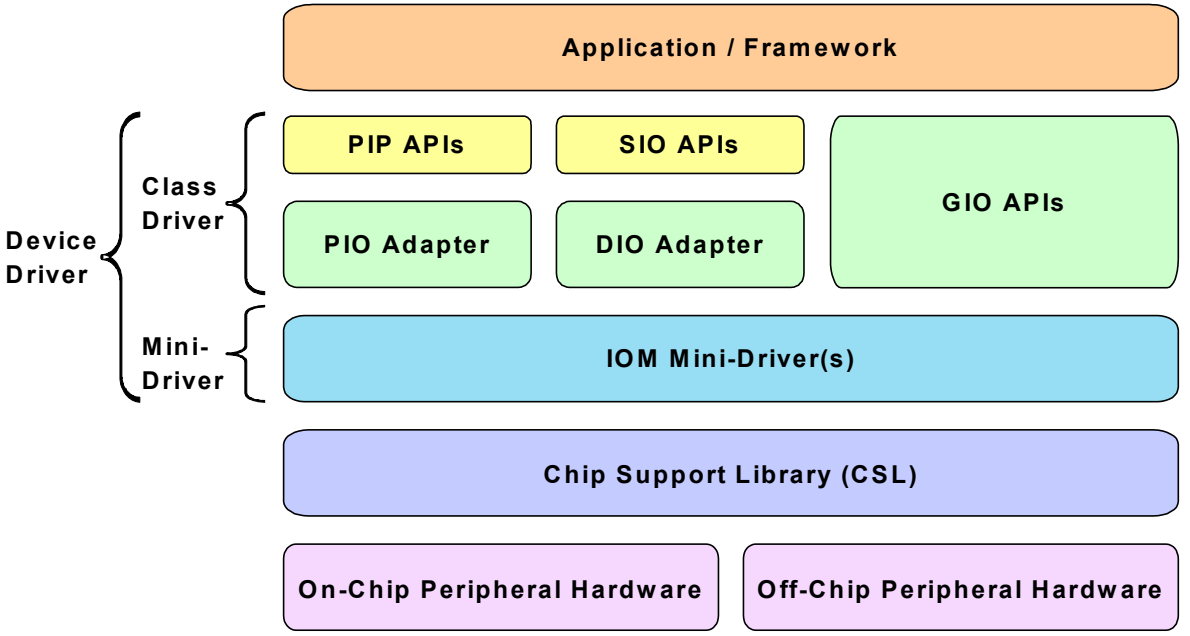
An effective driver model allows driver services to efficiently represent I/O requests to mini-drivers in the form of I/O request packets, containing information in a structure defined by the driver writer.

Device driver reusability can be enhanced by starting with the class driver/mini-driver partitioning and then adding the ability to richly express device I/O requests to the mini-driver.

2.1.1 Application Architecture Overview

Figure 2-1 shows relationships between the layers in an application that uses the two-layer device driver model described in this document.

Figure 2-1. Application Architecture Showing Device Driver Components



As this figure shows, the high-level application does not interact with mini-drivers directly. Instead, it uses one or more class drivers to interface with the mini-driver(s).

Each class driver presents an API to the application code and communicates with the IOM mini-driver interface. A class driver uses the DSP/BIOS API for OS services such as synchronization. It calls the standard mini-driver interface to access peripheral hardware devices.

DSP/BIOS currently defines the three class drivers shown in this figure: PIP/PIO, SIO/DIO, and GIO. In the case of the PIP/PIO and SIO/DIO class drivers, the APIs used by the application are the existing DSP/BIOS PIP and SIO functions. These APIs talk to the corresponding adapter, which communicates with the mini-driver. In the case of the GIO class driver, the application calls these APIs, which interfaces directly with the mini-driver.

More than one type of class driver may exist simultaneously in an application. Application writers can choose to use one or all of these in a system. Mini-driver writers typically do not need to write class drivers.

Each mini-driver exports standard mini-driver interface functions for class drivers to use to access the hardware and for DSP/BIOS device driver management. Mini-drivers use the Chip Support Library (CSL) to interface to the peripheral hardware's register, memory, and interrupt resources. Some mini-drivers may optionally include a codec-specific sub-driver.

2.1.2 Driver Initialization and Binding

Every DSP/BIOS module used by an application contains an initialization routine that is called by DSP/BIOS during DSP/BIOS initialization. Mini-drivers are treated like other DSP/BIOS modules. The initialization function of each registered mini-driver is called during the initialization of the DSP/BIOS DEV module.

The calling order of mini-driver initialization functions is determined by the order in which the mini-drivers were configured.

Each mini-driver's function table exports a binding (mdBindDev) function that gets called by DSP/BIOS after all the driver initialization functions have been called.

The parameters to the mdBindDev function (devp, devid, and devParams) are used to send the configuration parameters described in Section 3.1, *Registering the Mini-Driver*, page 3-2 to the mini-driver. For example, information about IRQ levels and statically-declared global memory may be passed via these parameters. The class driver also passes the devp and chanParams information to calls that create a channel (mdCreateChan) for the particular device. This allows device instances to share global data and allows a mini-driver to use a particular configured memory area.

2.1.3 Device and Channel Instances

The device driver model includes two types of object instances:

- **Device instances.** This is an abstraction of an actual peripheral device of some kind, like an audio codec or a video port peripheral. Device instances are registered (configured) in DSP/BIOS in the device table as described in Section 3.1, *Registering the Mini-Driver*, page 3-2. Applications refer to them by their "logical" names. If multiple device instances are configured, each has a unique logical name in the DSP/BIOS device table.
- **Channel instances.** This is an abstraction of a communication path between the application and a device instance. Channel instances are created through a call to the mini-driver’s mdCreateChan

function, which runs as a result of a call from the class driver (for example, SIO_create, PIO_create, or GIO_create).

A DSP system may have more that one physical device of the type abstracted by a device instance. Maintaining distinct driver states to allow the same mini-driver code to be used for multiple devices is an important feature of the driver. DSP/BIOS drivers are not required to support multi-device instancing, but it is strongly recommended if multiple hardware peripherals of the same type are possible. For example, most TI DSPs have multiple McBSP peripherals, so we support this in the DDK drivers that use the McBSP.

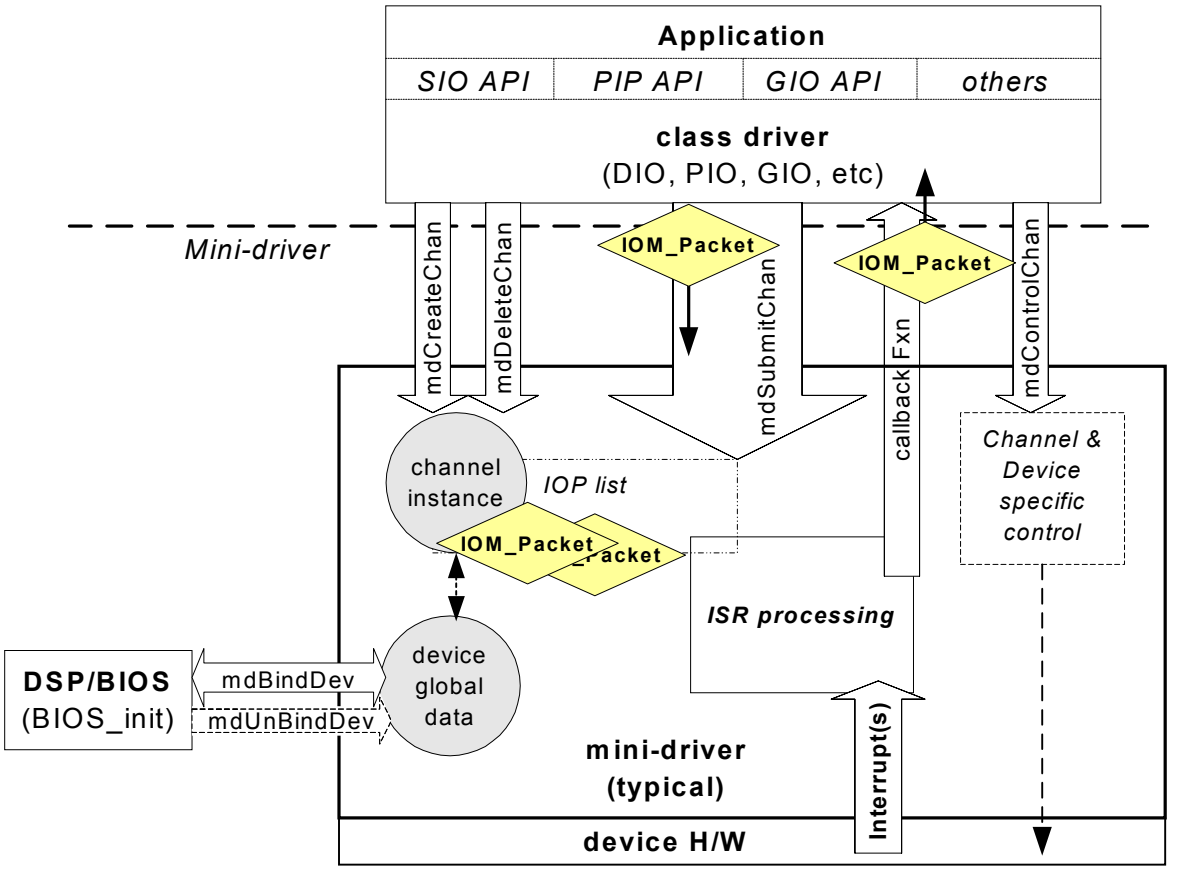
A given device instance may support multiple channel instances. An important attribute of a channel instance is the directional mode of operation. For mini-drivers, channel modes can be data input only, data output only, or both input and output (bi-directional). Mini-drivers not supporting one or more modes return an error status if an application attempts to create a channel with an unsupported channel mode.

2.2 Driver Data Flow

This section provides an overview of the flow of data from the application to the driver to the physical device.

Figure 2-2 shows the mini-driver call flow as used by the class driver. The IOM_Packets shown in this figure are standard data structures used to submit requests to a mini-driver. They contain a pointer to the data buffer.

Figure 2-2. Mini-Driver Operation Flow



2.2.1 Channel Instance Handles

Before data communication between an application and a device can begin, a channel instance handle must be returned to the application by the mini-driver's mdCreateChan function.

Depending on the class driver used by the application, the channel instance pointed to by a channel handle can represent point-to-point data streams (SIO) or pipes (PIP). In any case, the channel handle represents a unique communication path between the application and a mini-driver. All subsequent operations that talk to the driver use this channel handle.

The amount of resources—such as memory—consumed by each channel instance depends upon both the mini-driver and adapter's implementation of a channel. A channel object typically maintains data fields related to a channel's mode, I/O request queues, and possibly

driver state information. The total memory size required for each channel instance can vary greatly for different adapter and mini-driver implementations. Applications should relinquish channel resources by deleting all channel instances when they are no longer needed.

2.2.2 IOM Packets

The IOM_Packet data structure is used by IOM drivers to submit requests to a mini-driver. This standard data structure is equivalent to the DSP/BIOS DEV_Frame structure, which has been modified to include a command (cmd) field and a command status field. The IOM_Packet structure is defined as follows:

```
typedef struct IOM_Packet {  /* frame object */
   QUE_Elem   link;       /* queue link */
   Ptr        addr;       /* buffer address */
   Uns        size;       /* buffer size */
   Arg        misc;       /* reserved for driver */
   Arg        arg;        /* user argument */
   /* two fields added for use by IOM */
   Uns        cmd;        /* mini-driver command */
   Int        status;     /* status of command */
} IOM_Packet;
```

The two new fields extend the DSP/BIOS DEV_Frame structure to allow a richer set of I/O operations to be passed between the application and mini-driver. Applications themselves do not have access to IOM_Packet structures. Only class drivers, adapters, and mini-drivers have access to these packets. The mini-driver command (cmd) value constants allow the mdSubmitChan function to be called to read from, write to, abort, or flush the channel.

The IOM_Packet is a fixed-sized structure used by both the GIO and IOM modules. An IOM_Packet represents a single I/O request to the mini-driver. The mini-driver processes the request and returns the packet back to the class driver by calling the class driver's callback function. (See Section 2.2.4, *I/O Request Submissions*.)

IOM_Packet structures are created by class drivers and referenced in a mini-driver's channel object.

When an application submits an I/O request, the class driver fills in the fields of an IOM_Packet before submitting it to the mini-driver. GIO_submit (and related calls such as GIO_read and GIO_write) acts on the fields in the IOM packet as follows before submitting it:

- `QUE_Elem link;  /* queue link */`

  Used internally for queueing. May be used by the mini-driver.

- `Ptr addr;  /* buffer address */`

  The class driver sets this field to the bufp parameter passed to it. This parameter points to a data structure or buffer data. The mini-driver should preserve the value of this parameter.

- `Uns size;  /* buffer size */`

  The class driver sets this field to the size of the buffer data structure as provided by the pSize parameter. The mini-driver should update this size field if the actual size is different than requested.

- `Arg misc;  /* miscellaneous item */`

  The class driver sets this field to the application-specified callback function. The mini-driver should not change this field.

- `Arg arg;  /* user argument */`

  This field is not used by GIO. It is used by other class drivers.

- `Uns cmd;  /* command for mini-driver */`

  The class driver sets this field to the cmd parameter passed to it. The mini-driver uses this command code to determine which operation to perform in mdSubmitChan. The mini-driver may not change this field.

- `Int status;  /* status of command */`

  The mini-driver should set this status before calling the callback function. If the action was successful, the status should be set to IOM_COMPLETED. If failure occurred, the mini-driver should set the status to the appropriate IOM error code. The class driver returns this status to the application.

2.2.3 Channel Operations

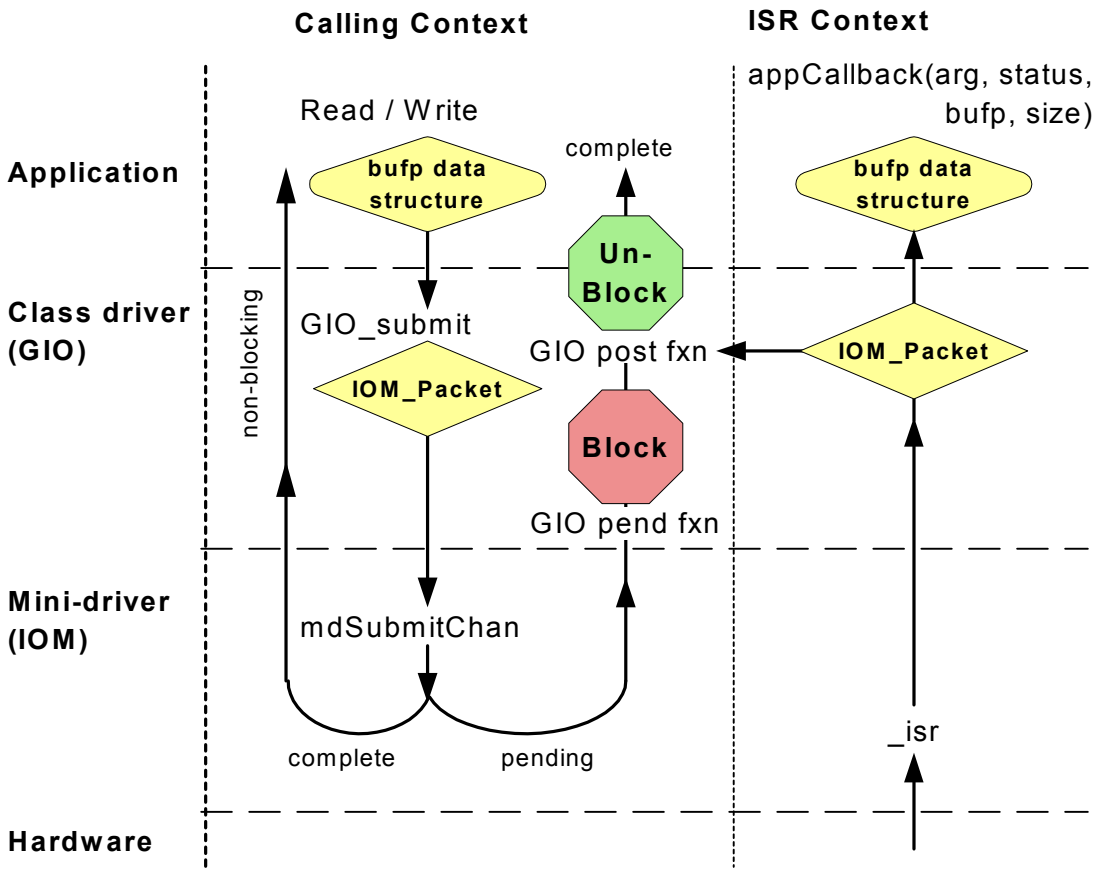
Once a channel has been created with mdCreateChan, the adapter can invoke channel operations in two ways:

- **mdSubmitChan.** Submits an I/O request in the form of an IOM_Packet.
- **mdControlChan.** Invokes a driver-specific control call.

Mini-driver channels are not required to serialize access from multiple calling contexts (for example, multiple tasks). Serialization is the responsibility of either the application or some higher-level software. This simplifies the mini-driver design.

Figure 2-3 shows the call flow resulting from a synchronous read or write call as it passes through the GIO class driver and IOM mini-driver levels of an application.

Figure 2-3. Call Flow for Blocking GIO_read and GIO_write



2.2.4 I/O Request Submissions

When a mini-driver receives an IOM_Packet via an mdSubmitChan call, the mini-driver can attempt to complete the request and if so return immediately to the caller, or it can queue the request for later completion.

When a mini-driver completes its processing, usually in an ISR context, it calls its associated callback function to pass the IOM_Packet back to the class driver. The submit/callback function pair handles the passing of IOM_Packets between the class driver and mini-driver. Before an IOM_Packet is passed back to the class driver, the mini-driver must set the completion status field and the data size field in the IOM_Packet. This

status value and size are returned to the application call that initially made the I/O request.

Note:

Mini-drivers must be written so that they can handle situations in which the class driver makes submit calls (via mdSubmitChan) faster than the mini-driver can complete them. In other words, the mini-driver should queue up IOM_Packets if necessary for later processing.

2.2.5 Device Control

Device control is device-dependent. The mini-driver's mdControlChan implements any control functionality that the mini-driver writer wishes to provide to the application designer. The application may invoke such control functionality through calls to GIO_control, SIO_ctrl, or PIO_ctrl.

The documentation for each mini-driver should provide a list of supported "control codes." Control codes not supported by a mini-driver must return the error code IOM_ENOTIMPL.

2.3 Class Driver Overview

DSP/BIOS supports two models for data transfer. One is the stream model, which is implemented by the SIO module. The other is the pipe model, which is implemented by the PIP module. Both of these models:

- ❏ Require that a pipe or stream has a single reader and single writer thread.
- ❏ Transfer buffers within the pipe or stream by copying pointers rather than by copying data.
- ❏ Are designed to manage block I/O (also called asynchronous I/O).

Both of these data transfer models can interface to IOM mini-drivers using adapters provided for use as part of a class driver.

The first adapter is the SIO adapter (DIO), which is used with the SIO module. The second adapter is the PIP adapter (PIO), which is used with the PIP module. See the *DSP/BIOS User's Guide* for an in-depth discussion of the DSP/BIOS stream and pipe models.

This DDK introduces a third data transfer model targeted more specifically to file system I/O and character based UART applications. This data transfer model is a stream-based synchronous I/O model that exposes more traditional read and write APIs to the application and is made available though the General I/O (GIO) class driver. The GIO class driver was written to adapt directly to the IOM interface and therefore has a built-in IOM adapter layer.

For many character I/O or file system application domains, GIO is a preferable data transfer model over the asynchronous block I/O models provided by SIO and PIP. Of course, there is overlap when deciding to use one model versus another. Ease of use and overhead should always be considered for any given application.

2.3.1 The SIO Adapter (DIO)

The DSP/BIOS streaming I/O (SIO) module provides a high-level device independent I/O mechanism for use with DSP/BIOS threads. SIO offers the ability to dynamically create SIO objects at run-time.

To provide this ability, SIO has its own device driver model, called DEV. DEV is described in detail in the DSP/BIOS manuals and on-line help. Writing a DEV is similar to writing an IOM mini-driver in that a small set of device-specific functions, such as open, close, and buffer management, are implemented and accessed by an SIO object through a function table. DEV drivers are generally more difficult to write than IOM

mini-drivers because they require a higher level of DSP/BIOS knowledge. In addition, a DEV driver can only be used with SIO—it cannot be used with PIP or on its own.

The SIO adapter, also called DIO, is designed to easily integrate streams with the IOM mini-driver model. Communication and synchronization are accomplished with minimal overhead and complexity. The DIO adapter uses the following basic types of functions:

- **Callback functions.** The callback functions are the signaling interface between the mini-driver and the adapter. During device driver channel creation, the adapter tells the mini-driver which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
- **Transfer function.** This function calls the mini-driver's mdSubmitChan function. The mdSubmitChan function of the mini-driver receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object. The DIO adapter uses these functions to communicate between the application and the mini-driver.

To configure an application to use the DIO adapter, see Section 3.2, *Configuring the DIO Class Driver (for SIO)*, page 3-4.

2.3.2 The PIP Adapter (PIO)

The DSP/BIOS PIP module provides a "data pipe" service to manage block I/O. Each pipe object maintains a buffer divided into a fixed number of fixed-length frames. The size and number of frames for a PIP are set in the DSP/BIOS Configuration Tool. Although each frame has a fixed length, the application may put less than a full frame of data into a pipe.

A pipe has two ends; the writer end is where the program writes frames of data. The reader end is where the program reads frames of data.

Typically, one end is a function that invokes an I/O device. Data notification functions are performed to synchronize data transfer and are triggered when a frame of data is read or written to notify the other end of the PIP of the availability of a full or empty frame. A writer gets a frame to put data into by calling the PIP_alloc function. After data is written to the frame, the writer calls PIP_put. This call results in the notifyReader function being called. When appropriate, the reader calls PIP_get to retrieve the frame of data and then calls PIP_free when the data is no longer required. The PIP_free call triggers the notifyWriter function and the cycle begins again. The notify functions associated with a PIP object are set by the user in the DSP/BIOS Configuration Tool.

The PIP adapter, also referred to as PIO, is designed to obtain a buffer from the application through the buffer manager and present it to the mini-driver for consumption. The adapter also recognizes when the mini-driver is finished processing the buffer and sends it back to the application through the buffer manager. This communication is accomplished with a minimal amount of overhead and complexity. The PIO adapter uses the following basic types of functions:

- **Prime functions.** The PIP buffer manager calls rxPrime and txPrime when the application sends a buffer to the device driver. These functions use DSP/BIOS API calls to obtain a buffer from the buffer manager and present it to the mini-driver. The "prime" functions are the signaling interface between the application and the adapter.
- **Callback functions.** The rxCallback and txCallback functions are the signaling interface between the mini-driver and the adapter. During driver set up, the adapter tells the mini-driver which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
- **Transfer function.** This function calls the device mini-driver's mdSubmitChan function. The mdSubmitChan function of the mini-driver receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object. The PIO adapter uses these functions to communicate between the application and the mini-driver.

To configure an application to use the PIO adapter, see Section 3.3, *Configuring the PIO Class Driver (for PIP)*, page 3-7. For details about the PIO adapter, see Appendix B, *PIO Adapter*.

2.3.3 The GIO Class Driver

The GIO class driver implementation was designed to minimize code and data size, while still providing the necessary common functionality for the synchronous read/write APIs and extensions. The GIO API may be used directly by applications to interface to IOM mini-drivers. These GIO APIs act as a class driver.

To configure an application to use the GIO class driver, see Section 3.4, *Configuring Applications to Use the GIO Class Driver*, page 3-10. For an overview of the GIO adapter, see Chapter 4, *GIO Class Driver*. For reference information about the GIO APIs, see the *DSP/BIOS API Reference* (literature number SPRU404 for C5000, SPRU403 for C6000, and SPRU625 for C2000).

Chapter 3

Using DSP/BIOS Device Drivers

This chapter describes how the DSP/BIOS device driver model can be configured for use with SIO, PIP, or GIO objects.

Topic **Page**

3.1 Registering the Mini-Driver 3–2
3.2 Configuring the DIO Class Driver (for SIO) 3–4
3.3 Configuring the PIO Class Driver (for PIP) 3–7
3.4 Configuring Applications to Use the GIO Class Driver 3–10

This chapter demonstrates the use of mini-drivers with DSP/BIOS I/O class drivers to build a DSP/BIOS application. As with most DSP/BIOS modules, the mini-drivers along with the class driver objects can be configured either in the DSP/BIOS Configuration Tool or by editing a DSP/BIOS Tconf script manually.

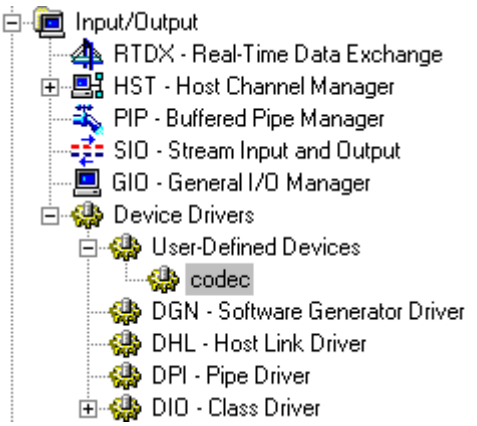
Section 3.1, *Registering the Mini-Driver*, page 3-2 describes how to configure an IOM mini-driver. This is required no matter which class driver you plan to use.

The remaining sections demonstrate how to use an IOM mini-driver with the DSP/BIOS SIO, PIP, and GIO class drivers. The class driver section you should follow depends on which class driver you plan to use.

3.1 Registering the Mini-Driver

To register an IOM mini-driver for use in a DSP/BIOS application, you must configure the application to use that mini-driver. This configuration is done statically, either though the DSP/BIOS Configuration Tool or a DSP/BIOS Tconf script.

To configure a mini-driver in the DSP/BIOS Configuration Tool, follow these steps:

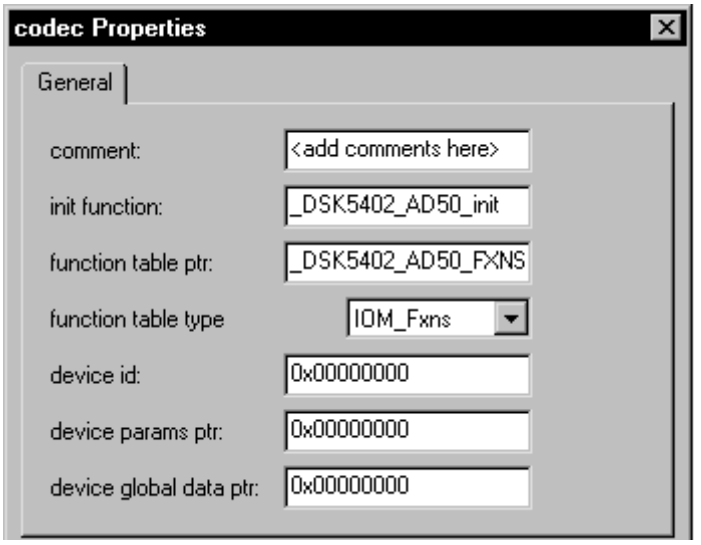


1) Create a new device object by right-clicking on User-Defined Devices (in the Input/Output tree) and selecting Insert UDEV from the pop-up menu.

2) Rename the object. This name identifies the peripheral in application code. For example, you might name it “codec” if the mini-driver drives the codec and the application code references it as “/codec”.

3) Right-click on the UDEV object you created and choose Properties.

4) In the Properties dialog, specify the properties to match those listed in the mini-driver documentation for your particular mini-driver (the *.pdf file in the doc folder). For example, the properties to specify for the DSK5402_AD50 mini-driver are as follows:



- **init function.** Specify the driver initialization function name.
- **function table pointer.** Specify the name of the driver function table. This table is mapped to the name given to this UDEV object. This allows applications to refer to mini-drivers using a name that need not be changed if a different mini-driver is used.
- **function table type.** Select IOM_Fxns if you plan to use a mini-driver created according to the specifications described in this document. The older Dxx drivers described in the *DSP/BIOS API Reference* use a function table of type DEV_Fxns.
- **device id.** If the mini-driver documentation indicates that one should be used, provide an integer value for the mini-driver to use to distinguish this instance of the device. For example, some DSPs have multiple devices of the same type (MCBSP0, MCBSP1, etc.) This maps to the mini-driver's mdBindDev devid parameter.
- **device params ptr.** If the mini-driver documentation specifies it, provide a pointer to a device-specific structure to be passed as an input parameter to the mini-driver. This maps to the mini-driver's mdBindDev devParams parameter.
- **device global data ptr.** If the mini-driver documentation specifies it, provide the memory location that contains the pointer to global data used by the mini-driver. When ROMing device drivers, it is useful to allow such memory to be allocated statically outside the actual driver, which permits it to avoid any hard-coded data references. This maps to the mini-driver's mdBindDev devp parameter.

Internally, devices configured as UDEV objects are included in a "device table" maintained by DSP/BIOS.

3.2 Configuring the DIO Class Driver (for SIO)

If your application uses the SIO (stream I/O) module DSP/BIOS APIs for I/O, you must use the DIO adapter to interface with a mini-driver. The combination of the SIO module and the DIO adapter creates a class driver. The SIO functions communicate with the DIO adapter, which in turn communicates with the mini-driver.

The DIO functions can either be configured for use with TSKs or SWIs via a switch for "callback version of DIO function table". This switch should be selected if using SWIs. The DIO functions can also be used with static or dynamically created objects.

A DIO object cannot be created until an IOM mini-driver has been registered.

3.2.1 Example Application

The tsk_audio example demonstrates how an application can use a codec mini-driver with SIO streams and TSK threads. This example application is configured to use the DIO adapter.

This example is provided for a number of codec drivers. When you examine the configuration, notice the TSK, UDEV, and DIO objects. The SIO objects are created dynamically at run-time. The example is located in ddk_1_20\packages\ti\bios\drivers\examples\audio.

The following DSP/BIOS objects are statically configured in this application example:

- ❏ The dio_codec class driver object, which uses the IOM mini-driver.
- ❏ A task, tskEcho, to run the echo function.
- ❏ A user-defined device (UDEV) object, named codec, to register the mini-driver.

The following DSP/BIOS objects are dynamically created in this application example:

- ❏ Two SIO streams, inStream and outStream, to exchange data between echo and the DIO Adapter.

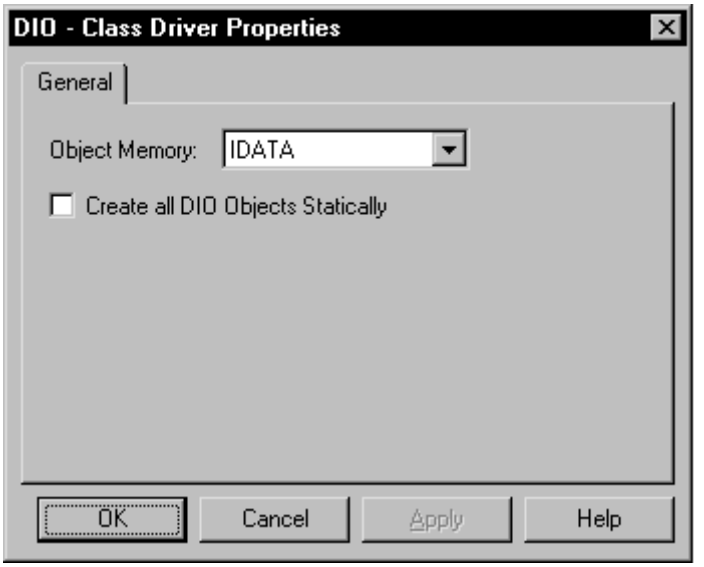
3.2.2 Configuration Steps

To configure the DIO adapter in the DSP/BIOS Configuration Tool, follow these steps:

1) Create a UDEV object for the mini-driver as described in Section 3.1, *Registering the Mini-Driver*, page 3-2 and in the mini-driver documentation for your particular mini-driver (see the docs folder).

2) Set the following property for the DIO - Class Driver manager:

   - **Create all DIO Objects Statically.** False (uncheck the box) unless you plan to use statically created SIO streams only

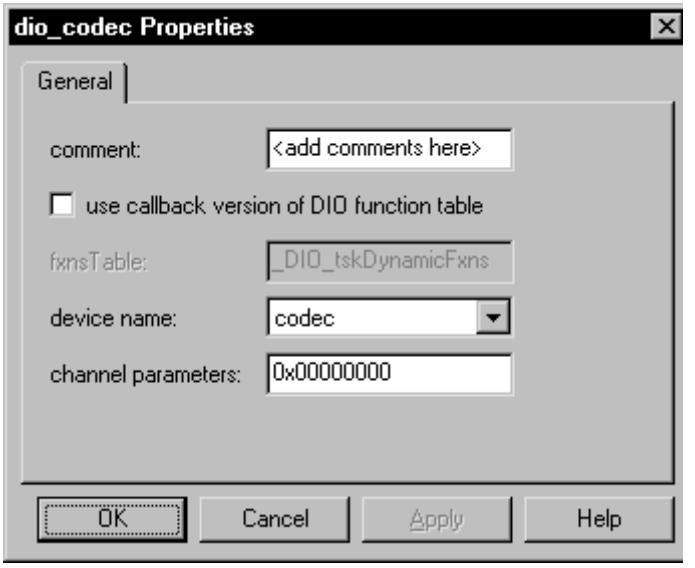


3) Right-click on DIO - Class Driver and select Insert DIO from the pop-up menu.

4) Rename the object as desired.

5) Right-click on the object you created and choose Properties. Set the following properties:

   - **use callback version of DIO functions.** False (uncheck the box) unless you plan to use SIO streams with SWI threads.
   - **device name.** Name of UDEV object created in Step 1.
   - **channel parameters.** 0x00000000 (or a pointer to a structure to be passed to mdCreateChan).



6) Create the TSK objects that will run your task threads. (It is also possible to use SWI threads with SIO stream objects. However, this use is not as common.)

7) Use the SIO module API functions as described in the DSP/BIOS API Reference Guide for creating and using SIO streams in your application example.

8) Link your application with the appropriate mini-driver library (located in the debug or release folder for your mini-driver).

3.3 Configuring the PIO Class Driver (for PIP)

If your application uses the PIP (pipe) module APIs for I/O, you should use the PIO adapter to interface with a mini-driver. The combination of the PIP module and the PIO adapter creates a class driver. The PIP functions communicate with the PIO adapter, which in turn communicates with the mini-driver.

Note:

Creating a PIO object in the DSP/BIOS configuration is not supported at this time. Currently, creating a DIO object with the appropriate parameters is equivalent to creating a PIO object.

The PIP module does not support the dynamic creation of object instances. As with other DSP/BIOS modules, if an application uses the PIP module, the initialization function for the PIP module is automatically called by DSP/BIOS during application startup.

3.3.1 Example Application

The pip_audio example demonstrates how an application can use a codec mini-driver via the PIP module in SWI threads. This application uses the PIO adapter.

This example is provided for a number of codec drivers. When you examine the configuration, notice the PIP, SWI, and UDEV objects. The example is in ddk_1_20\packages\ti\bios\drivers\examples\audio.

In this application example, an IOM mini-driver has been registered and both the PIP and SWI objects are statically created for use with the PIO adapter. The PIO objects are created dynamically.

The following DSP/BIOS objects are statically configured in this application example:

- A software interrupt, swiEcho, to run the echo function.
- Two data pipes pipRx and pipTx, to exchange data between echo and the PLIO Adapter.
- A user-defined device (UDEV) object, named codec, to register the mini-driver.

3.3.2 Configuration Steps

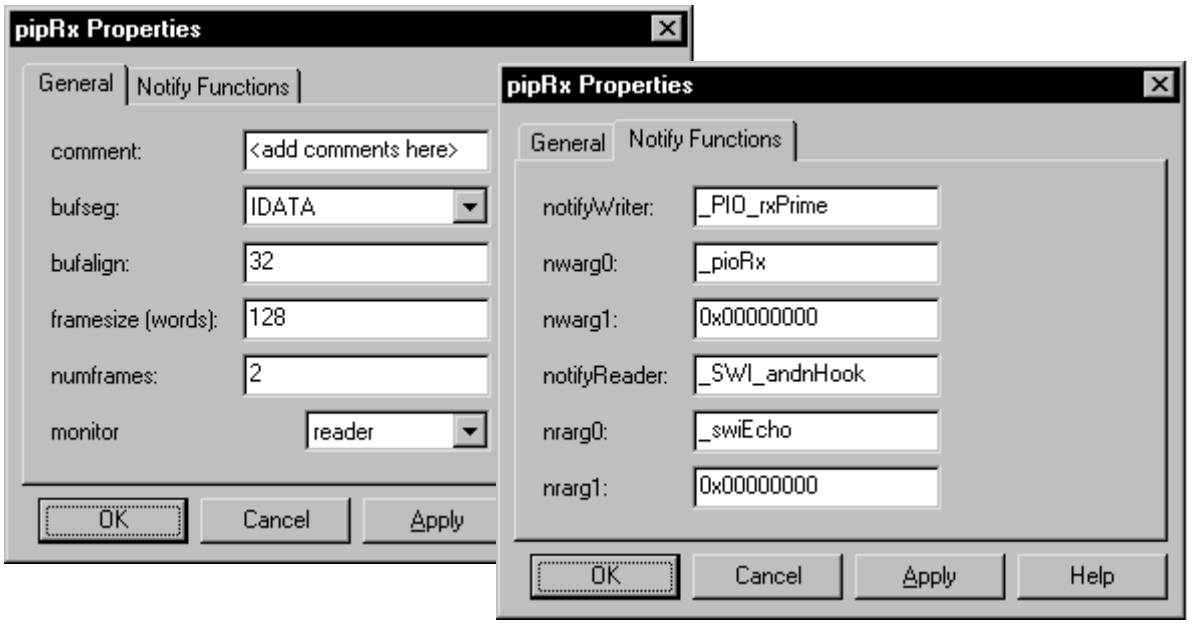
To configure the PIO adapter in the DSP/BIOS Configuration Tool, follow these steps:

1) Create a UDEV object for the mini-driver as described in Section 3.1, *Registering the Mini-Driver*, page 3-2 and in the mini-driver documentation for your particular mini-driver (see the docs folder).

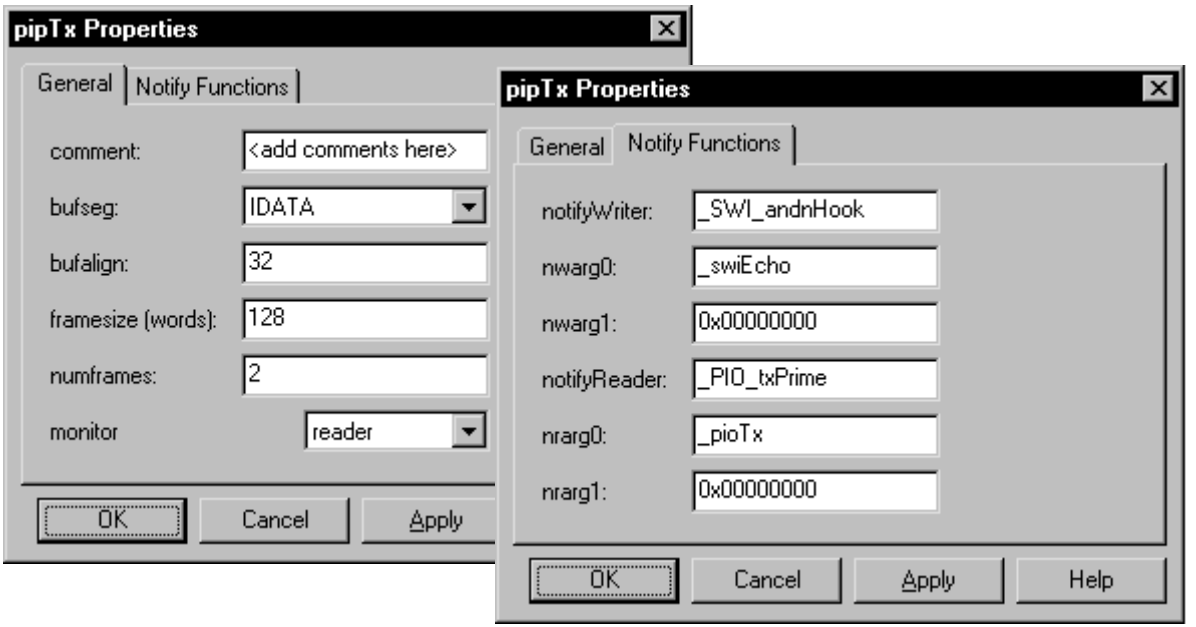
2) Create a SWI object using the SWI—Software Interrupt Manager and rename the SWI object. In the pip_audio example, it has been renamed swiEcho.

3) Create two PIP objects using the PIP—Buffered Pipe Manager. Rename the PIP objects. In the pip_audio example, the first pipe has been renamed pipRx, and the second pipe has been renamed pipTx.

4) Right-click on the first PIP (for example, pipRx) and select Properties. Set the following properties. Then click OK to save your changes.



5) Right-click on the second PIP (for example, pipTx) and select Properties. Set the following properties for pipTx:



6) Use the PIO module API functions as described in Appendix B to dynamically create two PIO channel objects—pioRx and pioTx—in your application. For example:

```
void main()
{
   /* Initialize PIO module */
   PIO_init();

   /* Bind PIPs to channels using PIO class drivers */
   PIO_new(&pioRx, &pipRx, "/codec", IOM_INPUT, NULL);
   PIO_new(&pioTx, &pipTx, "/codec", IOM_OUTPUT, NULL);
   ...
}
```

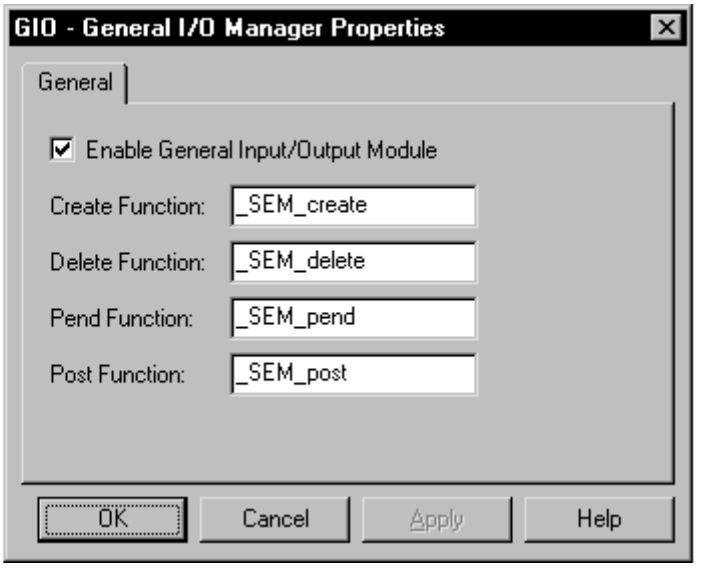
7) Link your application with the appropriate mini-driver library (located in the debug or release folder for your mini-driver).

3.4 Configuring Applications to Use the GIO Class Driver

Another option for your application is to use the GIO class driver to interface with a mini-driver.

As with most other DSP/BIOS modules, the GIO module can be configured in the DSP/BIOS Configuration Tool or a DSP/BIOS Tconf script. Figure 3-1 shows the configuration properties for the GIO module:

Figure 3-1. GIO Module Configuration



GIO has global properties that control how GIO blocks on I/O. By default, GIO uses the semaphore objects managed by the SEM module.

The GIO module does not support the creating of object instances. As with other DSP/BIOS modules, if an application uses the GIO module, the initialization function for the GIO module is automatically called by DSP/BIOS during application startup.

3.4.1 Example Application

The uarttest example demonstrates how an application can use the GIO class driver with TSK threads.

The following DSP/BIOS objects are statically configured in this application example:

- ❏ The GIO module object is enabled using the GIO General Input/Output Module properties.
- ❏ A task, tskEcho, to run the echo function.
- ❏ A user-defined device (UDEV) object, named uart, to register the mini-driver.

The following DSP/BIOS objects are dynamically created in this application example:

- ❏ A GIO channel for both input and output to exchange data between echo and the IOM Device Driver.

3.4.2 Configuration Steps

To use the GIO class driver in a DSP/BIOS application, follow these steps:

1) Create a UDEV object for the mini-driver as described in Section 3.1, *Registering the Mini-Driver*, page 3-2 and in the mini-driver documentation for your particular mini-driver (see the mini-driver's doc folder).

2) Use the GIO module API functions as described in Chapter 3 and in the *DSP/BIOS API Reference* manual to dynamically create GIO channel objects in your application. For example:

```
GIO_Handle input;

Void main() {
   input = GIO_create("/uart", IOM_INPUT, &status,
                      NULL, NULL);
...
}
```

3) Link your application with the appropriate mini-driver library (located in the debug or release folder for your mini-driver).

Chapter 4

GIO Class Driver

This chapter describes the GIO module, which provides a class driver.

Topic **Page**

4.1 About the GIO Module 4–2
4.2 Implementation Details for GIO 4–3
4.3 Error Handling 4–5
4.4 Extending the GIO API 4–5

4.1 About the GIO Module

The GIO module implements the GIO class driver, which is used to present a blocking (synchronous) Read/Write API to applications. By encapsulating this code, applications that use multiple IOM mini-drivers can reduce their overall code size.

The GIO module has the following features:

- ❏ Provides blocking (synchronous) read/write APIs.
- ❏ Uses the IOM interface to communicate with device-specific mini-driver implementations.
- ❏ Supports multiple device drivers.
- ❏ Supports bi-directional channels.
- ❏ Allows user configuration of blocking functions.
- ❏ Supports addition of APIs for new domains (such as video).

This last feature is important. The GIO_submit function supports a standard route for adding APIs customized for domains such as video.

The ASYNC module is an example of such customization. This module is provided with the DDK in order to support applications that use threads that cannot block. This module does not add to the code size of the application; it is implemented using macros that call existing GIO module APIs in specialized ways. For more details on the ASYNC module, see Appendix D, *The ASYNC Extension to the GIO API*, page D-1.

Examples of this type of customization include blocking read/write APIs for use with file systems, UARTs, and frame video DSP applications:

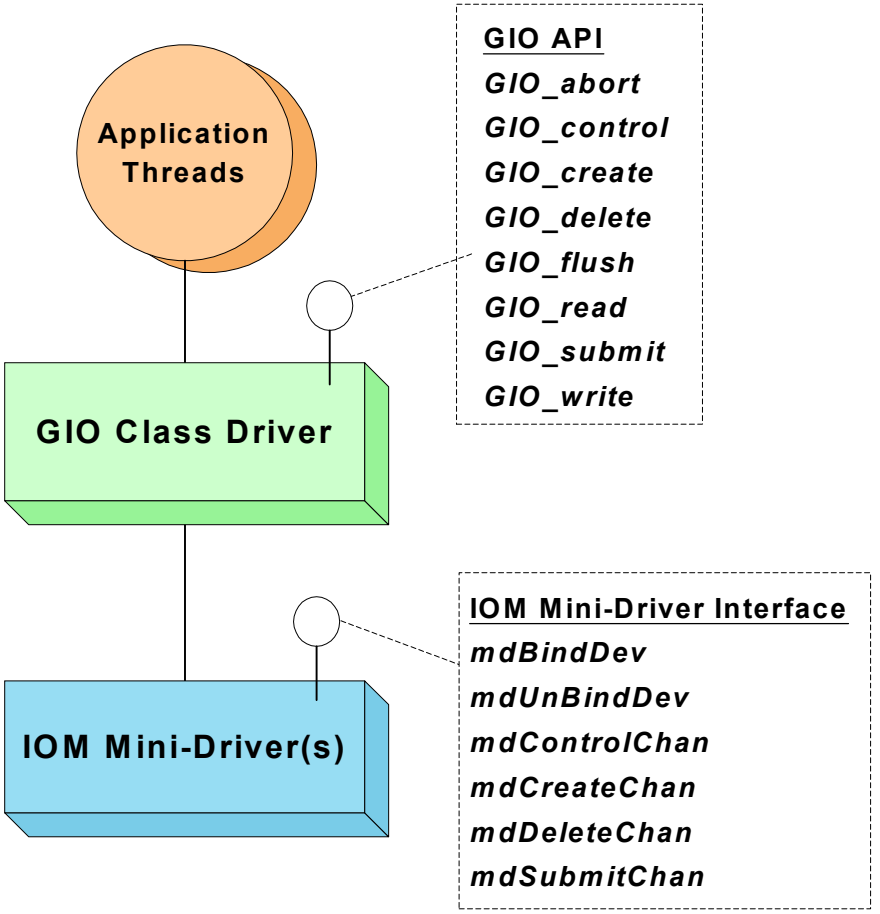
- ❏ File system stacks use "traditional" read/write APIs for application access to file data. They also need the ability to support bi-directional channels. This is a capability that may be supported through the GIO class driver and IOM mini-drivers.
- ❏ UART drivers also lend themselves to the read/write APIs. UARTs are typically character based or have variable line lengths, instead of fixed sized buffers.
- ❏ Extensions to the GIO interface can be made more “friendly” and efficient for use with video capture and video display drivers. For example, such extensions would typically address the need for video device memory allocations, such as special frame buffers. In addition, they might allow a single application call to "exchange" video buffers, providing synchronization of the most-recent video data between the video driver and application.

4.2 Implementation Details for GIO

The GIO class driver implementation was designed to minimize code and data size, while still providing the necessary common functionality for the synchronous read/write APIs and extensions. The GIO API may be used directly by applications to interface to IOM mini-drivers. These GIO APIs act as a class driver. The GIO module is described in the *DSP/BIOS API Reference* (literature number SPRU404 for C5000 and SPRU403 for C6000).

Figure 4-1 shows the relationship of the GIO class driver to other system components.

Figure 4-1. GIO Class Driver Interface



When creating device driver channels, the GIO class driver allocates additional state and I/O request state structures, IOM_Packets, and a GIO data object.

4.2.1 GIO_Obj Structure

The GIO_create function creates a GIO object for a particular IOM channel instance. A GIO_Obj has the following structure:

```
typedef struct GIO_Obj {
   IOM_Fxns *fxns;   /* pointer to function table */
   Uns      mode;    /* create mode */
   Uns      timeout; /* timeout for blocking */
   IOM_Packet syncPacket; /* for synchronous use */
   QUE_Obj    freeList;   /* frames for asynch I/O */
   Ptr      syncObj; /* ptr to synchronization obj */
   Ptr      mdChan;  /* ptr to channel obj */
} GIO_Obj, *GIO_Handle;
```

In particular, the GIO object provides storage for the following important information:

- ❏ The IOM mini-driver function table (fxns) the application communicates with through this GIO object.
- ❏ The mode to use when creating a channel. The options are IOM_INPUT, IOM_OUTPUT or IOM_INOUT.
- ❏ Any IOM_Packets (freeList) used between the class driver and mini-driver for asynchronous operation. These are allocated and freed by GIO.
- ❏ The synchronization object used for a particular channel (syncObj). For example, a semaphore when using the GIO blocking APIs.
- ❏ A pointer to the specific IOM mini-driver channel object (mdChan).

4.3 Error Handling

In general, the GIO class driver code ignores mini-driver error codes returned to the application. Some error codes are returned to the application by GIO. For example, if dynamic allocations fail IOM_EALLOC is returned, or if the IOM_Packet allocation fails the more specific IOM_ENOPACKETS error code is returned.

GIO performs additional processing to aid mini-driver writers in case of a GIO channel timeout. If a GIO channel timeouts while blocking, a calldown to the mini-driver via the mdControlChan function is made with a command code of IOM_CHAN_TIMEDOUT. This calldown is performed before the application thread returns. This gives the mini-driver the opportunity to perform any necessary timeout cleanup, such as returning the IOM_Packet to the class driver if necessary.

4.4 Extending the GIO API

As mentioned in Section 4.1, *About the GIO Module*, one important feature of GIO is its ability to support the addition of APIs for new application domains.

This section presents an example of a video API extension to the GIO class driver. This example addresses the real-world need for a video API that is more efficient than using the existing SIO or PIP APIs. This video API specifically addresses the needs of frame-by-frame video capture and display device drivers. An entire frame video module (FVID) can be created through simple macro definitions using the GIO APIs. As a result, no new code is added to the application.

In this example, frame-by-frame video capture and display involves the following additional requirements:

- The API should not be difficult to understand and use in video applications.

  By minimizing or eliminating the need for device-specific control calls in applications, the reusability and understandability of the code is enhanced.

- Address video frame buffer synchronization between the application and device driver.

  For example, a video display driver must always be displaying video data. It does not return the frame buffer to the application until the application gives it another to display. The most efficient way to perform this type of synchronization is to define a single API call that performs the frame buffer exchange.

Likewise, a video capture driver should always return the latest "captured" video data. This requires the driver to retain at least one frame buffer until a buffer exchange can occur.

- Address frame buffer memory management.

  Frame buffer memory can be allocated from system memory or from device memory depending on the video device hardware. Some video devices have on-board video RAM that applications can use for video frame buffers. Applications should not have to change the usage of an API depending on where the frame buffer memory is allocated.

To address additional frame video APIs to support the above requirements, we introduce the following new FVID module APIs:

Example 4.1. Example Video Macro Definitions using GIO:

```
/* base video command code */
#define FVID_BASE  IOM_USER   /* base command value */
/* video command codes */
#define FVID_ALLOC (FVID_BASE + 0) /* alloc buffer */
#define FVID_FREE  (FVID_BASE + 1) /* free buffer */
#define FVID_EXCHANGE (FVID_BASE + 2) /* exchange */

/* frame-base video(FVID) API */

/* Get a video frame buffer from the mini-driver */
#define FVID_alloc(gioChan, bufp, psize) \
GIO_submit(gioChan, FVID_ALLOC, bufp, psize, NULL)

/* Return the frame buffer back to mini-driver */
#define FVID_free(gioChan, bufp, psize) \
GIO_submit(gioChan, FVID_FREE, bufp, psize, NULL)

/* Swap frame buffer between app and mini-driver */
#define FVID_exchange(gioChan, bufp, psize) \
GIO_submit(gioChan, FVID_EXCHANGE, bufp, psize, NULL)
```

Each of these video APIs is actually a macro that uses the GIO_submit function with the command (cmd) parameter set to a special video command. The class driver does not interpret the command code; it simply passes the command to the mini-driver in the IOM_Packet. The mini-driver is responsible for appropriate processing. This simple mechanism of defining domain-specific command values for the GIO_submit API provides the flexibility to define additional APIs that can be used with particular mini-drivers.

Chapter 5

Developing a Mini-Driver Step-by-Step

This chapter shows a step-by-step process for developing a mini-driver.

Topic **Page**

5.1 Mini-Driver Design and Implementation 5–2
5.2 Examining the ‘C5402 SBS Mini-Driver Example 5–6

5.1 Mini-Driver Design and Implementation

Appendix A, *IOM Interface*, provides a reference for the functions that are implemented in order to create a mini-driver. Figure 2-2 provides an overview of the mini-driver call flow.

This section describes functionality usually required when developing a mini-driver and provides some advice for implementation. The following topics are discussed:

- ❏ Binding Channels—mdBindDev
- ❏ Creating and Deleting Channels—mdCreateChan/mdDeleteChan
- ❏ Submitting I/O Requests—mdSubmitChan
- ❏ Servicing Interrupts and Completing I/O—ISRs
- ❏ Controlling a Device—mdControlChan

5.1.1 Binding Channels—mdBindDev

The binding function (mdBindDev) for each configured IOM mini-driver is called by DSP/BIOS during device initialization. This function should typically perform the following actions:

- ❏ Set device defaults and perform setup based on the configured device parameters and optional global device data.
- ❏ Plug device ISRs.
- ❏ Acquire driver resources such as memory, McBSPs, and DMAs.

5.1.1.1 Single or Multiple Device Instances

If the mini-driver is designed to allow multiple device instances, then DSP/BIOS calls the mdBindDev function for each configured device. The device's devid parameter is typically used to distinguish between multiple instances of a particular device type.

If the mini-driver does not support multiple device instances, the mdBindDev function should check to see if it is called more than once by DSP/BIOS. It should return an error if a user attempts to configure more than one instance of this mini-driver. For example:

```
static Bool curinit = FALSE;
if (curinit) {
   return (IOM_EBADIO);  /* mdBindDev already called */
}
curinit = TRUE;
```

5.1.1.2 Global Device Data

It is often desirable for mini-drivers to use static data to eliminate the need for dynamic data allocations at run-time. A mini-driver can use the in/out data pointer (devp) as passed instead of returning a global data pointer to the driver data it allocates dynamically. The devp parameter is also used to allow global data to be accessed by each channel instance of a device instance. This devp pointer is passed to the mdCreateChan function as the first parameter.

5.1.2 Creating and Deleting Channels—mdCreateChan/mdDeleteChan

From a application's perspective, a logical communication channel is created between the application and the device instance where driver data may be exchanged. The application creates one or more of these logical channels, which is represented by the mini-driver as a channel object.

The mdCreateChan function should allocate a channel object and set fields in the channel object to their initial values as needed. The mdDeleteChan function should deallocate the specified channel object.

The structure of the channel object is defined differently by each mini-driver. The following example shows a sample channel object structure:

```
typedef struct ChanObj {
   Bool     inuse; /* TRUE => channel is open */
   Int      mode;  /* IOM_INPUT or IOM_OUTPUT only */
   IOM_Packet *dataPacket;  /* active I/O packet */
   QUE_Obj  pendList; /* list of packets for I/O */
   Uns      *bufptr;  /* pointer in current buffer */
   Uns      bufcnt;   /* samples left to handle */
   IOM_TiomCallback cbFxn; /* to notify client */
   Ptr      cbArg;
} ChanObj, *ChanHandle;
```

While the structure may vary, some fields are necessary in nearly all mini-drivers. These important fields are:

- **mode.** Must have a value of IOM_INPUT, IOM_OUTPUT, or IOM_INOUT. The mdCreateChan function should return an error of IOM_EBADMODE for any unsupported modes.
- **pendList.** A mini-driver must be able to handle or hold multiple I/O requests due to multiple mdSubmitChan calls by the class driver. By queuing these I/O requests in the mini-driver, a more efficient implementation can be realized when processing multi-queued requests. Essentially, the mini-driver can initiate the next queued request without having to initiate the completion callback first

(because requests are queued within the mini-driver). Also, this eliminates the case of the class driver calling the mdSubmitChan mini-driver function in the second calling context of an ISR, that is, the callback context.

- **cbFxn.** The callback function pointer stores which function is called as a result of a completed I/O request. This is typically the callback function implemented as part of the class driver.
- **cbArg.** The callback argument is a pointer that is an input parameter to the callback function. This allows the class driver to associate the I/O request submitted to the specific callback.

5.1.3 Submitting I/O Requests—mdSubmitChan

The mdSubmitChan mini-driver function must handle command code passed to it as part of the IOM_Packet structure. Depending on the command code, it either handles the code or returns the IOM_ENOTIMPL (not implemented) error code.

The currently supported mini-driver command codes are: IOM_READ, IOM_WRITE, IOM_ABORT, and IOM_FLUSH.

- **IOM_READ.** Drivers that support input channels must implement IOM_READ.
- **IOM_WRITE.** Drivers that support output channels must implement IOM_WRITE.
- **IOM_ABORT and IOM_FLUSH.** To abort or flush I/O requests already submitted, all I/O requests pending in the mini-driver must be completed and returned to the class driver. The mdSubmitChan function should dequeue each of the I/O requests from the mini-driver's channel queue. It should then set the size and status fields in the IOM_Packet. Finally, it should call the cbFxn for the channel.
  - When aborting, all input and output requests are discarded.
  - When flushing, all output requests and processed normally and all input requests are discarded.This requires the processing of each IOM_Packet in the original order they were queued up to the channel.

5.1.4 Servicing Interrupts and Completing I/O—ISRs

The role of mini-drivers is to process actual device events, such as a periodic interrupt. For devices such as audio codecs, these events are typically interrupts that signal the completion of a data sample input to or output from the device. Whether DMA or a sample-by-sample device

interrupt is used to synchronize the data transfer, the driver must handle and synchronize these events.

Many devices have separate Tx and Rx interrupts that the mini-driver must handle. In general, the IOM mini-driver must perform the following activities in these ISR(s):

- ❏ Dequeue the IOM_Packet request.
- ❏ Set up the next transfer or service request.
- ❏ Call the class driver callback to synchronize with the calling application, and return the IOM_Packet.

5.1.5 Controlling a Device—mdControlChan

The control operations supported by a mini-driver are device-specific. However, IOM does define some standard control codes that may optionally be implemented by driver writers. Control codes not handled by the mdControlChan function must return a status of IOM_ENOTIMPL.

Driver writers should implement additional command codes using values greater than IOM_CNTL_USER (128). The IOM control codes currently defined by IOM are:

- ❏ **IOM_CHAN_RESET.** Channel-specific. Perform processing to place the device channel in its initial state. After calling mdControlChan with this command code, normal I/O operations should succeed.
- ❏ **IOM_CHAN_TIMEDOUT.** Channel-specific. When a timeout occurs in a class driver or application, this control code can handle timeout processing. For example, after a timeout the IOM_Packet may not have been returned to the class driver if the callback was not called.
- ❏ **IOM_DEVICE_RESET.** Global. Perform a global device reset. This affects all channels of this device.

A mini-driver's supported control codes and operations performed should be documented for application developers who integrate that mini-driver. When describing the processing for a control code, be careful to specify the scope of that processing as channel-specific or global. For example, if the baud rate is changed, it may influence the baud rate of all channels or a single channel. The mini-driver documentation should also specify the context or constraints on when each control code may be called.

5.2 Examining the ‘C5402 SBS Mini-Driver Example

To learn how to write a mini-driver, it is useful to examine the code for a simple mini-driver.

This section shows portions of the code for a McBSP sample-by-sample device driver for the ‘C5402 DSK. This relatively simple mini-driver is not included in the DDK distribution. This code is for illustrative purposes only. More realistic example of audio drivers can be found in the DDK.

The sections that follow describe the actions performed by this mini-driver. These can serve as an example for the actions other mini-drivers should perform. The source code files for this mini-driver would be c54xx_dma_mcbsp.c and c54xx_dma_mcbsp.h.

The sample-by-sample driver described in this section copies samples between a McBSP and a buffer. When receiving a buffer, it calls the callback function if the buffer is full. When transmitting a buffer, it calls the callback function if the buffer is empty.

5.2.1 Constants, Types, and Structures

The header file for this driver sets up the following structures and declarations:

- **IOM_Fxns.** Each IOM mini-driver must provide an IOM_Fxns table to be referenced by the “function table ptr” property of the UDEV object for this mini-driver. The mini-driver declares the following table, and the header file makes an extern declaration for the table name. Each of the functions referenced in this table must be created as part of the mini-driver.

```
/* Public IOM interface table */
IOM_Fxns DSK5402_MCBSP_AD50_FXNS = {
   mdBindDev,
   IOM_UNBINDDEVNOTIMPL,
   mdControlChan,
   mdCreateChan,
   mdDeleteChan,
   mdSubmitChan
};
```

- **Initialization function.** The initialization function for the mini-driver is not included in the IOM_Fxns table. Instead, a separate extern is created for use by DSP/BIOS. In the configuration, this function must be referenced by the “init function” property of the UDEV object for this mini-driver.

```
extern Void DSK5402_MCBSP_AD50_init( Void );
```

- **devParams.** If the devParams parameter passed to the mdBindDev function is null, then the mini-driver may elect to use default device parameter values.
- **ChanObj.** Each mini-driver must declare and create one or more channel objects. In this example, the ChanObj has the following structure. It also declares constants for the mode field and for the number of channels to create. See Section 5.1.2, *Creating and Deleting Channels—mdCreateChan/mdDeleteChan* for more on channel objects.

```
typedef struct ChanObj {
   Bool inuse;  /* TRUE => channel is open */
   Int mode;    /* IOM_INPUT or IOM_OUTPUT only */
   IOM_Packet *dataPacket; /* active I/O packet */
   QUE_Obj pendList; /* list of packets for I/O */
   Uns *bufptr; /* pointer in current buffer */
   Uns bufcnt;  /* samples left to handle */
   IOM_TiomCallback cbFxn; /* to notify client */
   Ptr cbArg;
} ChanObj, *ChanHandle;
```

In this example, the channel objects are initially declared as follows:

```
#define NUMCHANS 2 /* INPUT and OUTPUT */

static ChanObj chans[NUMCHANS] = {
   {FALSE, INPUT, NULL, {NULL, NULL}, NULL, 0,
    NULL, NULL},
   {FALSE, OUTPUT, NULL, {NULL, NULL}, NULL, 0,
    NULL, NULL}
};
```

5.2.2 Initialization Function

In the case of this sample-by-sample driver, the initialization function performs no actions. In other mini-drivers, the initialization function may need to set registers or initialize data used globally by this mini-driver.

5.2.3 mdBindDev Function

The mdBindDev function is called by DSP/BIOS during device initialization. For the sample-by-sample driver, it is called after the DSK5402_MCBSP_AD50_init function.

The following example shows the mdBindDev function for the McBSP sample-by-sample device driver for the 'C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Int mdBindDev(Ptr *devp, Int devid,
                     Ptr devParams)
{
Step 1:    DSK5402_MCBSP_AD50_DevParams *params =
               (DSK5402_MCBSP_AD50_DevParams *)devParams;
           static Bool curinit = FALSE;

Step 2:    /* CSL handle to the McBSP. The McBSP is shared
              between the two channels */
           static MCBSP_Config mcbspCfg0 = {
             0x0021, /* Serial Port Control Register 1 */
             0x0201, /* Serial Port Control Register 2 */
             0x0040, /* Receive Control Register 1 */
             0x0000, /* Receive Control Register 2 */
             0x0040, /* Transmit Control Register 1 */
             0x0000, /* Transmit Control Register 2 */
             0x0000, /* Sample Rate Generator Register 1 */
             0x0000, /* Sample Rate Generator Register 2 */
             0x0000, /* Multichannel Control Register 1 */
             0x0000, /* Multichannel Control Register 2 */
             0x000c, /* Pin Control Register */
             0x0000, /* Rx Chan Enable Reg Partition A */
             0x0000, /* Rx Chan Enable Reg Partition B */
             0x0000, /* Tx Chan Enable Reg Partition A */
             0x0000  /* Tx Chan Enable Reg Partition B */
           };

Step 3:    /* for CPLD CTRL 2 */
           static volatile ioport unsigned port04;

Step 4:    if (curinit) {
               return (IOM_EBADIO);
           }
           curinit = TRUE;

Step 5:    /* use default parameters if none are given */
           if (params == NULL) {
               params = &DSK5402_MCBSP_AD50_DEVPARAMS;
           }
```

Step 6:

```
    /* open the McBSP */
    hMcbsp = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
    MCBSP_config(hMcbsp, &mcbspCfg0);
```

Step 7:

```
    /*
     * DSK5402 board setup ...
     * Select McBSP1 mapped to Audio Codec (CPLD Reg)
     * and FC bit = 0 (secondary control off)
     */
    port04 &= 0xf5;
```

Step 8:

```
    /* start the McBSP */
    MCBSP_start(hMcbsp, MCBSP_XMIT_START |
                        MCBSP_RCV_START, 0x0);
```

Step 9:

```
    /* set codec params (also initializes codec) */
    AD50_setParams(hMcbsp, &(params->ad50) );
```

Step 10:

```
    /* bind Rx/Tx interrupts and
       use DSP/BIOS HWI dispatcher */
    HWI_dispatchPlug(IRQ_EVT_RINT1, (Fxn)rxIsr, NULL);
    HWI_dispatchPlug(IRQ_EVT_XINT1, (Fxn)txIsr, NULL);
```

Step 11:

```
    *devp = chans;
```

Step 12:

```
    return (IOM_COMPLETED);
}
```

In this example, the mdBindDev function performs the following actions:

1) Casts the devParams passed to the function to the DSK5402_MCBSP_AD50_DevParams type.
2) Configures the CSL handle to the McBSP by setting various registers. The McBSP is shared by the input and output channels.
3) Configures the port for CPLD CTRL 2.
4) Returns a status value of IOM_EBADIO if mdBindDev has already been called.
5) Sets the parameters to their default values if the devParams passed to this function was null.
6) Opens and configures the McBSP using calls to MCBSP_open and MCBSP_config.
7) Sets up the DSK5402 board by mapping McBSP1 to the audio codec (CPLD Register) and by setting the FC bit to 0 (secondary control off).
8) Starts the McBSP by calling MCBSP_start.

9) Sets codec parameters to initialize the codec by calling AD50_setParams.

10) Binds the Rx and Tx interrupts to use the DSP/BIOS HWI dispatcher by calling HWI_dispatchPlug.

11) Sets the devp parameter to point to the globally declared chans array.

12) Returns the IOM_COMPLETED status.

The mdUnBindDev function is not implemented for this sample-by-sample driver.

5.2.4 mdControlChan Function

The mdControlChan function is called in response to calls to SIO_ctrl, PIO_ctrl, and GIO_control. In addition, if a class driver reaches a timeout, a calldown is made to mdControlChan with the cmd set to IOM_CHAN_TIMEDOUT.

The following example shows the mdControlChan function for the McBSP sample-by-sample device driver for the ‘C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Int mdControlChan(Ptr chanp, Uns cmd, Ptr args)
{
   /* If a channel timeouts (in GIO class driver),
    * a calldown is made to mdControlChan with
    * cmd = IOM_CHAN_TIMEDOUT. Timeout processing
    * is optionally implemented here.
    * If cmd not performed return IOM_ENOTIMPL.
    */
Step 1:  if (cmd == IOM_CHAN_TIMEDOUT) {
            /* Timed out. Perform channel cleanup. */
Step 2:     abortio(chanp);
         }
Step 3:  else {
            /* return IOM_ENOTIMPL for codes not handled */
            return (IOM_ENOTIMPL);
         }

Step 4:  return (IOM_COMPLETED);
}
```

1) For this sample-by-sample driver, only the IOM_CHAN_TIMEDOUT command is handled.

2) When a timeout occurs, this function runs the abortio function, which discards uncompleted I/O packet requests.

3) All other commands cause this function to return a status of IOM_ENOTIMPL.

4) After successfully processing a command, the mdControlChan function returns IOM_COMPLETED.

5.2.5 mdCreateChan Function

The mdCreateChan function is called in response to calls to SIO_create, PIO_new, GIO_create, and similar class driver actions. In this example, the mdCreateChan function performs the following actions:

The following example shows the mdCreateChan function for the McBSP sample-by-sample device driver for the ‘C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Int mdCreateChan(Ptr *chanp, Ptr devp,
                        String name, Int mode,
                        Ptr chanParams,
                        IOM_TiomCallback cbFxn,
                        Ptr cbArg)
{
Step 1:     ChanHandle  chans = (ChanHandle)devp;
            ChanHandle  chan;

Step 2:     if (mode == IOM_INPUT) {
                chan = &chans[INPUT]; /* input only channel */
            }
            else if (mode == IOM_OUTPUT) {
               chan = &chans[OUTPUT]; /* output only channel */
            }
            else {
                /* bi-directional channels not supported */
                return (IOM_EBADMODE);
            }

Step 3:     /* Check if channel is already in use.
             * Use ATM_setu() for atomic test-and-set.
             */
            if (ATM_setu((Uns *)&chan->inuse, TRUE)) {
                /* ERROR! channel is already open! */
                return (IOM_EBADIO);
            }
```

Step 4:
```
    QUE_new(&chan->pendList);
```
Step 5:
```
    chan->dataPacket = NULL;
    /* no need to initialize chan->bufptr */
```
Step 6:
```
    chan->cbFxn = cbFxn;
    chan->cbArg = cbArg;
```

Step 7:
```
    if (chan->mode == INPUT) {
       IRQ_enable(IRQ_EVT_RINT1); /* Rx Intr enable */
    }
    else {
       IRQ_enable(IRQ_EVT_XINT1); /* Tx intr enable */
    }
```

Step 8:
```
   *chanp = chan;          /* return channel handle */
```

Step 9:
```
    return (IOM_COMPLETED);       /* success */
}
```

1) Creates a handle to the declared chans array and a handle to the specific channel being created within that array.
2) Uses the mode parameter to determine which channel the handle should point to. If the mode is unsupported, it returns a status of IOM_EBADMODE.
3) Checks to see if the channel is already in use. This is indicated by the inuse field in the channel object, which it check atomically. If the channel is already in use, it returns a status of IOM_EBADIO.
4) Initializes the pendList QUE object. This object is created as part of the channel object structure.
5) Sets the dataPacket field of the channel object to null.
6) Sets the cbFxn and cbArg fields of the channel object to the values passed to the mdCreateChan function.
7) If this is an input channel, it enables IRQ_EVT_RINT1. If this is an output channel, it enables IRQ_EVT_XINT1.
8) Sets the chanp parameter to the channel handle created by this function.
9) Returns a status of IOM_COMPLETED.

5.2.6 mdDeleteChan Function

The mdDeleteChan function is called in response to calls to SIO_delete, GIO_delete, and similar class driver actions.

The following example shows the mdDeleteChan function for the McBSP sample-by-sample device driver for the ‘C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Int mdDeleteChan(Ptr chanp)
{
    ChanHandle chan = (ChanHandle)chanp;

Step 1:     chan->inuse = FALSE;

Step 2:     if (chan->mode == INPUT) {
                IRQ_disable(IRQ_EVT_RINT1);
            }
            else {
                IRQ_disable(IRQ_EVT_XINT1);
            }

Step 3:     return (IOM_COMPLETED);
}
```

1) Sets the inuse field in the channel object to FALSE.
2) If this was an input channel, it disables IRQ_EVT_RINT1. If this is an output channel, it disables IRQ_EVT_XINT1.
3) Returns a status of IOM_COMPLETED.

5.2.7 mdSubmitChan Function

The mdSubmitChan function is called in response to calls to GIO_submit or similar class driver actions.

The following example shows the mdSubmitChan function for the McBSP sample-by-sample device driver for the ‘C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Int mdSubmitChan(Ptr chanp, IOM_Packet *packet)
{
   ChanHandle chan = (ChanHandle)chanp;

Step 1:
   /* Check if command is to abort or flush */
   /* Note: For this audio codec we toss output
      data even when flushing */
   if ( packet->cmd == IOM_FLUSH ||
        packet->cmd == IOM_ABORT) {
      abortio(chan);

      /* flush/abort pkt completed */
      packet->status = IOM_COMPLETED;
      return (IOM_COMPLETED);
   }

Step 2:
   if (chan->dataPacket == NULL) {
      chan->bufptr = (Uns *)packet->addr;
      chan->bufcnt = packet->size;

      /* dataPacket must be set last to synch w/ ISR */
      chan->dataPacket = packet;
   }
   else {
Step 3:
      QUE_put(&chan->pendList, packet);
   }

Step 4:
   return (IOM_PENDING);
}
```

1) Checks to see if the command in the IOM_Packet is to abort or flush the channel. In either case, it calls the abortio function, which discards uncompleted I/O packet requests by getting all packets from the pendList queue and setting the status field of each IOM_Packet to IOM_ABORTED.

   Note that for this particular audio mini-driver, both the abort and flush commands are handled by discarding output data. Typically, flush command handling differs from the abort command in that output packets are handled normally.

   If it aborted or flushed the channel, it sets the status field of the current IOM_Packet to IOM_COMPLETED and returns a status of IOM_COMPLETED.

2) Continues if the command in the IOM_Packet is to read or write (the remaining commands). If the dataPacket field for the channel object is null, the packet does not need to be queued. This function sets the bufptr and bufcnt fields of the channel object using values in the

IOM_Packet. It then sets the dataPacket field to point to the IOM_Packet itself.

3) If the dataPacket field is non-null, this function puts the packet on the pendList queue.

4) Returns a status of IOM_PENDING.

Actual processing of packets is handled by the mini-driver ISR functions, which are described in Section 5.2.9, *ISR Functions*, page 5-16.

5.2.8 mdUnBindDev Function

The mdUnBindDev function is not implemented for the McBSP sample-by-sample device driver for the ‘C5402 DSK. Instead, the following example shows the mdUnBindDev function for the generic McBSP driver for the TMS320C6x1x series. The numbers in the left column next to the example correspond to the step numbers in the list that follows the example.

```
static Int mdUnBindDev(Ptr devp)
{
    PortHandle port = (PortHandle) devp;

Step 1:     port->inUse = FALSE;

Step 2:     /* Close the McBSP */
            MCBSP_close(port->hMcbsp);

Step 3:     return (IOM_COMPLETED);
}
```

1) Sets the port’s inUse field to false.

2) Closes the McBSP by calling MCBSP_close.

3) Returns a status of IOM_COMPLETED.

5.2.9 ISR Functions

The mini-driver's ISR functions are called in response hardware interrupts. In this example, the sample-by-sample driver has two ISRs: rxIsr and txIsr. These functions are bound to interrupts in the mdBindDev function.

5.2.9.1 rxIsr Function

The rxIsr function handles receive (input) interrupts.

The following example shows the rxIsr function for the McBSP sample-by-sample device driver for the 'C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Void rxIsr(Void)
{
   ChanHandle  chan = &chans[INPUT];

Step 1:
   if (chan->dataPacket == NULL) {
      MCBSP_read(hMcbsp);   /* toss data */
      return;
   }

Step 2:
   *chan->bufptr = MCBSP_read(hMcbsp);

Step 3:
   updateChan(chan);
}
```

1) If the dataPacket field for the channel is null, it discards the input data from the McBSP and returns.
2) Otherwise, it reads a new sample from the McBSP and copies it the the location pointed to by the bufptr field in the channel object.
3) It then calls the updateChan function that follows.

```
static Void updateChan(ChanHandle chan)
{
   IOM_Packet *tmpPacket;

Step 4:
   chan->bufptr++;
   chan->bufcnt--;

   /* Is this buffer finished? */
Step 5:
   if (chan->bufcnt == 0) {
Step 6:
      chan->dataPacket->status = IOM_COMPLETED;

      tmpPacket = chan->dataPacket;

Step 7:
      chan->dataPacket = QUE_get(&chan->pendList);
      if (chan->dataPacket == (
                  IOM_Packet *)&chan->pendList) {
         chan->dataPacket = NULL;
      }
      else {
         chan->bufptr = chan->dataPacket->addr;
         chan->bufcnt = chan->dataPacket->size;
      }

Step 8:
      (*chan->cbFxn)(chan->cbArg, tmpPacket);
   }
}
```

4) The updateChan function increments the bufptr field of the channel object and decrements the bufcnt field of the channel object.

5) If the bufcnt field has reached zero, it continues processing the following steps. Otherwise updateChan returns to the ISR, which has finished its processing.

6) The updateChan function sets the status field in the current IOM_Packet to IOM_COMPLETED.

7) The updateChan function attempts to get another IOM_Packet from the pendList queue. If another IOM_Packet is available, it sets the bufptr and bufcnt fields of the channel object to the addr and size values in the IOM_Packet.

8) The updateChan function calls the callback function specified for the channel. This callback function is passed to mdCreateChan by the class driver used by the application.

5.2.9.2 txIsr Function

The txIsr function handles transmit (output) interrupts.

The following example shows the rxIsr function for the McBSP sample-by-sample device driver for the 'C5402 DSK. The numbers in the left column next to the following example correspond to the step numbers in the list that follows the example.

```
static Void txIsr(Void)
{
   ChanHandlechan = &chans[OUTPUT];

Step 1:   if (chan->dataPacket == NULL) {
             MCBSP_write(hMcbsp, 0);/* output dummy sample */
             return;
          }

Step 2:   MCBSP_write(hMcbsp, *chan->bufptr & 0xfffe);

Step 3:   updateChan(chan);
}
```

1) If the dataPacket field for the channel is null, it writes a dummy sample to the McBSP and returns.
2) Otherwise, it writes the contents of the bufptr field of the channel object to the McBSP.
3) It then calls the updateChan function, which performs the same functions described previously for the rxIsr.

Appendix A

IOM Interface

This appendix provides reference details for the IOM (I/O Mini-driver) interface.

Topic **Page**

A.1 Mini-Driver Interface Overview A–2
mdBindDev A–5
mdControlChan A–6
mdCreateChan A–7
mdDeleteChan A–9
mdSubmitChan A–10
mdUnBindDev A–12

A.1 Mini-Driver Interface Overview

The mini-driver interface specifies how to implement a mini-driver.

Functions

A mini-driver should implement the following functions:

- ❏ mdBindDev. Bind device to mini-driver.
- ❏ mdControlChan. Perform channel control command.
- ❏ mdCreateChan. Create a device channel.
- ❏ mdDeleteChan. Delete a channel.
- ❏ mdSubmitChan. Submit a packet to a channel for processing.
- ❏ mdUnBindDev. Unbind device from mini-driver.

Description

A mini-driver contains the device-specific portions of the driver. Once you create the specified functions for your mini-driver, application integrators can easily use your mini-driver through the DIO adapter, PIO adapter, and/or GIO class driver.

The sections that follow describe how to implement the mini-driver functions in detail. Once implemented, these functions should be referenced in an interface table of type IOM_Fxns, which applications will reference to integrate the mini-driver. For example:

```
IOM_Fxns DSK5402_MCBSP_AD50_FXNS = {
    mdBindDev,
    IOM_UNBINDDEVNOTIMPL,
    mdControlChan,
    mdCreateChan,
    mdDeleteChan,
    mdSubmitChan
};
```

Note:

Any mini-driver functions you choose not to implement should either plug the mini-driver function table with IOM_xxxNOTIMPL, where xxx corresponds to the function name. Alternately, you may implement a function that returns a status of IOM_ENOTIMPL.

Constants, Types, and Structures

The following code can be found in iom.h, which is located in the *BIOS_INSTALL_DIR*\packages\ti\bios\include folder.

```
/* Modes for mdCreateChan */
#define IOM_INPUT    0x0001
#define IOM_OUTPUT   0x0002
#define IOM_INOUT    (IOM_INPUT | IOM_OUTPUT)

/* IOM Status Codes */
#define IOM_COMPLETED      0   /* I/O successful */
#define IOM_PENDING        1   /* I/O queued and pending */
#define IOM_FLUSHED        2   /* I/O request flushed */
#define IOM_ABORTED        3   /* I/O aborted */

/* IOM Error Codes */
#define IOM_EBADIO        -1   /* generic failure */
#define IOM_ETIMEOUT      -2   /* timeout occurred */
#define IOM_ENOPACKETS    -3   /* no packets available */
#define IOM_EFREE         -4   /* unable to free resources */
#define IOM_EALLOC        -5   /* unable to alloc resource */
#define IOM_EABORT        -6   /* I/O aborted uncompleted*/
#define IOM_EBADMODE      -7   /* illegal device mode */
#define IOM_EOF           -8   /* end-of-file encountered */
#define IOM_ENOTIMPL      -9   /* operation not supported */
#define IOM_EBADARGS      -10  /* illegal arguments used */
#define IOM_ETIMEOUTUNREC -11  /* unrecoverable timeout */
#define IOM_EINUSE        -12  /* device already in use */

/* Command codes for IOM_Packet */
#define IOM_READ           0
#define IOM_WRITE          1
#define IOM_ABORT          2
#define IOM_FLUSH          3
#define IOM_USER         128  /* 0-127 reserved for system */

/* Command codes for GIO_control and mdControlChan */
#define IOM_CHAN_RESET      0 /* reset channel only */
#define IOM_CHAN_TIMEDOUT   1 /* channel timeout occurred */
#define IOM_DEVICE_RESET    2 /* reset entire device */

typedef struct IOM_Fxns
{
    IOM_TmdBindDev       mdBindDev;
    IOM_TmdUnBindDev     mdUnBindDev;
    IOM_TmdControlChan   mdControlChan;
    IOM_TmdCreateChan    mdCreateChan;
    IOM_TmdDeleteChan    mdDeleteChan;
    IOM_TmdSubmitChan    mdSubmitChan;
} IOM_Fxns;
```

```
#define IOM_BINDDEVNOTIMPL (IOM_TmdBindDev)IOM_mdNotImpl
#define IOM_UNBINDDEVNOTIMPL (IOM_TmdUnBindDev)IOM_mdNotImpl
#define IOM_CONTROLCHANNOTIMPL (IOM_TmdControlChan)IOM_mdNotImpl
#define IOM_CREATECHANNOTIMPL (IOM_TmdCreateChan)IOM_mdNotImpl
#define IOM_DELETECHANNOTIMPL (IOM_TmdDeleteChan)IOM_mdNotImpl
#define IOM_SUBMITCHANNOTIMPL (IOM_TmdSubmitChan)IOM_mdNotImpl

            typedef struct IOM_Packet {  /* frame object */
               QUE_Elem   link;       /* queue link */
               Ptr        addr;       /* buffer address */
               Uns        size;       /* buffer size */
               Arg        misc;       /* reserved for driver */
               Arg        arg;        /* user argument */
               Uns        cmd;        /* mini-driver command */
               Int        status;     /* status of command */
            } IOM_Packet;

            /* Mini-driver's callback function. */
            Void (*IOM_TiomCallback)(Ptr arg, IOM_Packet *packet);
```

Example For example mini-driver implementations, see the source and header files in DDK driver folders.

mdBindDev

Bind device to mini-driver

C Interface

Syntax

```
status = mdBindDev(*devp, devid, devParams);
```

Parameters

```
Prt     *devp;      /* address for global device data pointer */
Int     devid;      /* device id */
Ptr     devParams;  /* pointer to config parameters */
```

Return Value

```
Int     status;     /* success or failure code */
```

Description

The mdBindDev function is called by DSP/BIOS during device initialization. It is called once per configured device and is called after the mini-driver's initialization function. For example, it would be called after DSK5402_MCBSP_AD50_init().

This function is typically used to specify device-specific global data, such as interrupts IDs and global data structures (for ROM-ability). Additional system resources may be allocated by the mini-driver at runtime.

The devp parameter provides the address where the function should place the global device data pointer.

The devid parameter is used to identify specific devices for systems that have more than one device of a specific type. For example, several McBSP mini-drivers use the devid parameter to specify which McBSP port to allocate and configure.

The devParams parameter is a pointer to the configuration parameters to be used to configure the device. For example, the C54XX_DMA_MCBSP mini-driver declares device parameters as follows:

```
typedef struct C54XX_DMA_MCBSP_DevParams {
    Uns           rxDmaId;
    Uns           txDmaId;
    MCBSP_Config  *mcbspCfg;
} C54XX_DMA_MCBSP_DevParams;
```

This function should return IOM_COMPLETED if it is successful. If a failure occurs, it should return one of the a negative error codes listed in Section A.1, *Mini-Driver Interface Overview*. If this function returns a failure code, the DSP/BIOS initialization fails with a call to SYS_abort.

Example

See Section 5.2.3, *mdBindDev Function*, page 5-8 for an example mdBindDev function.

mdControlChan *Perform channel control command*

C Interface

Syntax

status = mdControlChan (chanp, cmd, arg);

Parameters

```
Ptr     chanp;   /* channel handle */
Uns     cmd;     /* control functionality to perform */
Ptr     arg;     /* optional device-defined data structure */
```

Return Value

```
Int     status;  /* success or failure code */
```

Description

A class driver calls this function to cause the mini-driver to perform some type of control functionality. For example, it may cause the mini-driver to reset the device or get the device status. Calling SIO_ctrl, PIO_ctrl, or GIO_control results in execution of the appropriate mini-driver's mdControlChan function.

The chanp parameter provides a channel handle to identify the device instance.

The cmd parameter indicates which control functionality should be carried out.

```
/* Command codes for GIO_control and mdControlChan */
#define IOM_CHAN_RESET    0  /* reset channel only */
#define IOM_CHAN_TIMEDOUT 1
                          /* channel timeout occurred */
#define IOM_DEVICE_RESET  2
                          /* reset entire device */
```

The arg parameter is an optional device-defined data structure used to pass control information between the device and the application

If successful, this function should return IOM_COMPLETED. If the cmd value provided is unsupported, this function should return a status of IOM_ENOTIMPL.

Example

See Section 5.2.4, *mdControlChan Function*, page 5-10 for an example mdControlChan function.

mdCreateChan *Create a device channel*

C Interface

Syntax

```
status = mdCreateChan (*chanp, devp, name, mode, chanParams,
                       cbFxn, cbArg);
```

Parameters

Ptr	*chanp;	/* channel handle */
Ptr	devp;	/* device global data structure */
String	name	/* name of device instance */
Int	mode	/* input or output mode */
Ptr	chanParams	/*pointer to channel parameters */
IOM_TiomCallback	cbFxn	/* pointer to callback function */
Ptr	cbArg	/* callback function argument */

Return Value

Int	status;	/* success or failure code */

Description

A class driver calls this function to create a channel instance. Calling GIO_create, SIO_create, or PIO_new results in execution of the appropriate mini-driver's mdCreateChan function.

The chanp parameter provides an address at which this function should place a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See Section 5.2.3, *mdBindDev Function*, page 5-8 for an example.

The devp parameter is a pointer to the device's global data structure. This is the value returned by the mini-driver's mdBindDev call.

The name parameter is the name of the device instance. This is the remainder of the device name after getting a complete match from the DSP/BIOS device driver table. For example, this might contain channel parameters.

The mode parameter specifies whether the device is being opened in input mode, output mode, or both. The mode may be IOM_INPUT, IOM_OUTPUT, or IOM_INOUT. If your driver does not support one or more modes, this function should return IOM_EBADMODE for unsupported modes.

The chanParams parameter is used to pass device- or domain-specific arguments to the mini-driver. For example, the C54XX_DMA_MCBSP driver declares a structure for channel parameters as follows:

```
typedef struct C54XX_DMA_MCBSP_ChanParams {
    DMA_Config      *dmaCfg;
} C54XX_DMA_MCBSP_ChanParams;
```

The cbFxn parameter is a function pointer that points to the callback function to be called by the mini-driver when it has completed a request.

The cbArg parameter is an argument to be passed back by the mini-driver when it invokes the callback function.

Typically, the mdCreateChan function places the callback function and its argument in the device-specific data structure. For example:

```
chan->cbFxn = cbFxn;
chan->cbArg = cbArg;
```

If successful, this function should return `IOM_COMPLETED`. If unsuccessful, this function should return one of the a negative error codes listed in Section A.1, *Mini-Driver Interface Overview*.

Example

See Section 5.2.5, *mdCreateChan Function*, page 5-11 for an example mdCreateChan function.

mdDeleteChan

Delete a channel

C Interface

Syntax

status = mdDeleteChan (Ptr chanp)

Parameters

Ptr	chanp;	/* channel handle */

Return Value

Int	status;	/* success or failure code */

Description

A class driver calls this function to delete the specified channel instance. Calling SIO_delete or GIO_delete results in execution of the appropriate mini-driver's mdDeleteChan function.

The chanp parameter provides a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See the mdBindDev topic for an example.

If successful, this function should return IOM_COMPLETED. If unsuccessful, this function should return one of the a negative error codes listed in Section A.1, *Mini-Driver Interface Overview*.

Example

See Section 5.2.6, *mdDeleteChan Function*, page 5-13 for an example mdDeleteChan function.

mdSubmitChan

Submit a packet to a channel for processing

C Interface

Syntax

status = mdSubmitChan (chanp, *packet);

Parameters

Ptr	chanp;	/* channel handle */
IOM_Packet	*packet;	/* pointer to IOM_Packet */

Return Value

Int	status;	/* success or failure code */

Description

A class driver calls this function to cause the mini-driver to process the IOM_Packet. Calls to SIO_issue, PIP_put, GIO_submit, GIO_read, GIO_write, GIO_abort, and GIO_flush result in execution of the appropriate mini-driver's mdSubmitChan function.

Note:

The mini-driver function mdSubmitChan must be written to be reentrant to allow it to be called from multiple thread contexts.

The chanp parameter provides a channel handle to identify the device instance. The channel handle is a pointer to a device-specific data structure. See the mdBindDev topic for an example.

The packet parameter points to a structure of type IOM_Packet. This structure is defined as follows:

```
typedef struct IOM_Packet {  /* frame object */
   QUE_Elem   link;       /* queue link */
   Ptr        addr;       /* buffer address */
   Uns        size;       /* buffer size */
   Arg        misc;       /* reserved for driver */
   Arg        arg;        /* user argument */
   /* two fields added for use by IOM */
   Uns        cmd;        /* mini-driver command */
   Int        status;     /* status of command */
} IOM_Packet;
```

The value for the cmd code may be one of the following:

```
#define IOM_READ        0
#define IOM_WRITE       1
#define IOM_ABORT       2
#define IOM_FLUSH       3
```

Additional cmd codes may be added for domain-specific commands. Such codes should be constants with values greater than 127. See the iom.h file for these cmd codes.

If the cmd code is IOM_READ or IOM_WRITE, this function should queue the packet on the pending list.

If the cmd code is IOM_ABORT, this function should abort both read and write packets.

If the cmd code is IOM_FLUSH, this function should complete queued writes, but abort queued reads.

If this function successfully completes a read or write IOM_Packet request, it should return IOM_COMPLETED as the return status. If this function queues up a read or write request, it should return IOM_PENDING. If this function successfully aborts or flushes a packet, it should return IOM_COMPLETED. If unsuccessful, this function should return one of the a negative error codes listed in Section A.1, *Mini-Driver Interface Overview*.

Example

See Section 5.2.7, *mdSubmitChan Function*, page 5-13 for an example mdSubmitChan function.

mdUnBindDev

Unbind device from mini-driver

C Interface

Syntax

status = mdUnBindDev(devp);

Parameters

Ptr	devp;	/* global device data pointer */

Return Value

Int	status;	/* success or failure code */

Description

This function should free resources allocated by the mdBindDev function.

Currently, this function is not called as a result of any GIO functions or used by the PIO or DIO class drivers. It may be used in the future to support dynamic device driver loading and unloading.

The devp parameter is a pointer to the device's global data structure. This is the value returned by the mini-driver's mdBindDev call.

If successful, this function should return IOM_COMPLETED. If unsuccessful, this function should return a negative error code.

Example

See Section 5.2.8, *mdUnBindDev Function*, page 5-15 for an example mdSubmitChan function.

Appendix B

PIO Adapter

This appendix provides reference details for the PIO (Pipe I/O) adapter interface.

Topic **Page**

B.1 PIO Adapter Interface Overview B–2
PIO_ctrl B–5
PIO_init B–6
PIO_new B–7
PIO_rxPrime B–8
PIO_rxStart B–9
PIO_txPrime B–10
PIO_txStart B–11
rxCallback B–12
txCallback B–13

B.1 PIO Adapter Interface Overview

The PIP/PIO class driver is used to create an interface between the PIP (pipe) module and an IOM mini-driver.

Functions

The PIO module provides the following functions:

- ❏ PIO_ctrl. Control call to mini-driver
- ❏ PIO_init. Initialize the PIO module
- ❏ PIO_new. Initialize PIO object for a channel
- ❏ PIO_rxPrime. Supply a frame to the receiver
- ❏ PIO_rxStart. Start receive with one or more frames
- ❏ PIO_txPrime. Supply a frame to the transmitter
- ❏ PIO_txStart. Start transmit with one or more frames
- ❏ rxCallback. Notify PIO of receive completion
- ❏ txCallback. Notify PIO of transmit completion

Description

The PIO adapter obtains a buffer from the application through the buffer manager and presents it to the mini-driver for consumption. The adapter recognizes when the mini-driver is finished processing the buffer and sends it back to the application through the buffer manager. This communication is accomplished with a minimal amount of overhead and complexity.

The pio.h file is provided in ddk_1_20\packages\ti\bios\drivers\pio. Applications that use the PIO module must include this header file.

To use the PIO adapter, create the following objects in the configuration for the application:

- ❏ **SWI.** Add a software interrupt from the SWI - Software Interrupt Manager and choose Insert SWI. Rename the new SWI0 object. For example, in the pip_audio example, the object is called swiEcho. In the object's properties window, enter the name of the function and the desired mailbox value. For example, the pip_audio example uses _echo as the function and 3 as the mailbox value.
- ❏ **PIP.** Add pipe objects from the PIP - Buffered Pipe Manager and choose Insert PIP twice. For example, in the pip_audio example, rename the first pipe to pipRx and the second pipe to pipTx. The length of the buffers should be the same and can be any size. In the example, the pipRx notifyWriter function setting should be PIO_rxPrime(pioRx) and the notifyReader function settings should be SWI_andn(swiEcho,1). The pipTx notifyWriter function setting

should be SWI_andn(swiEcho,2) and notifyReader function settings should be PIO_txPrime(pioTx).

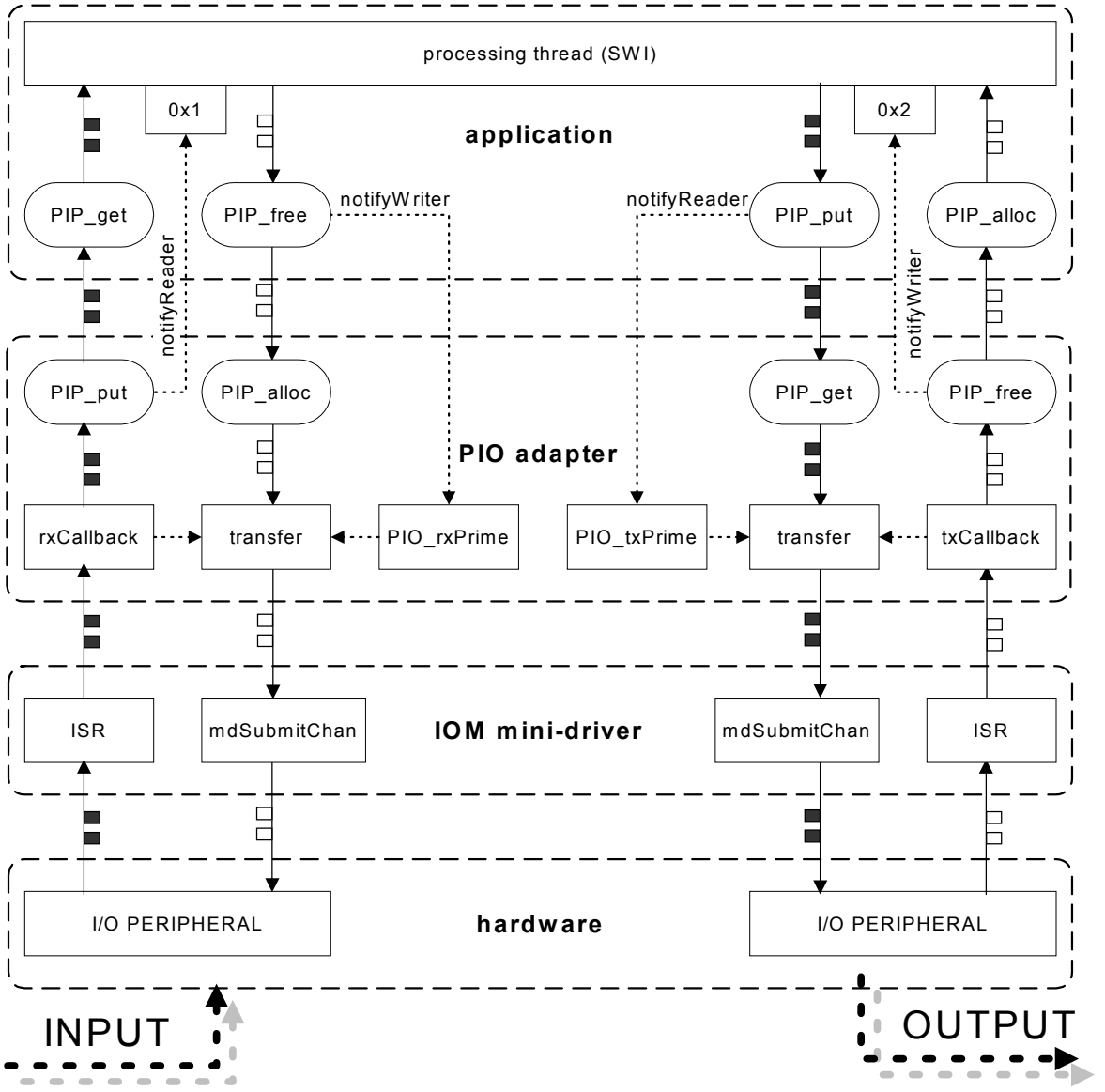
The PIO adapter uses the following basic types of functions:

- **Prime functions.** The PIP buffer manager calls rxPrime and txPrime when the application sends a buffer to the device driver. These functions use DSP/BIOS API calls to obtain a buffer from the buffer manager and present it to the mini-driver. The "prime" functions are the signaling interface between the application and the adapter.
- **Callback functions.** The rxCallback and txCallback functions are the signaling interface between the mini-driver and the adapter. During driver setup, the adapter tells the mini-driver which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
- **Transfer function.** This function calls the device mini-driver's mdSubmitChan function. The mdSubmitChan function of the mini-driver receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object.

The PIO adapter uses these functions to communicate between the application and the mini-driver. This is shown in Figure 5. Arrows with full or empty boxes attached indicate buffer flow; simple arrows indicate critical function calls.

For an example that uses the PIO module, see the pip_audio.c example in the ddk_1_20\packages\ti\bios\drivers\examples\audio folder.

Figure 5-1. PIO Adapter Buffer Flow



Constants, Types, and Structures

```
typedef struct PIO_Attrs {
    Ptr openArgs;
} PIO_Attrs;

/* default PIO_Attrs struct */
extern PIO_Attrs PIO_ATTRS;
```

PIO_ctrl

Control call to mini-driver

C Interface

Syntax

status = PIO_ctrl (pio, cmd, arg);

Parameters

```
PIO_Handle  pio;     /* pointer to channel's PIO object */
Uns         cmd;     /* IOM command */
Ptr         arg;     /* IOM argument */
```

Return Value

```
Bool        status;  /* success or failure code */
```

Reentrant

No

Description

PIO_ctrl is implemented as a macro and will pass its arguments directly to the IOM ctrl function.

Constraints and Calling Context

- ❏ Specific to the IOM mini-driver in use.

PIO_init

Initialize the PIO module

C Interface

Syntax PIO_init();

Parameters None

Return Value Void

Reentrant No

Description This function initializes the PIO adapter module.

Constraints and Calling Context

- ❑ Must be called before other PIO calls may be made.

Example The following example initializes the PIO module.

```
PIO_init();
```

PIO_new

Initialize PIO object for a channel

C Interface

Syntax

PIO_new (pio, pip, name, mode, *attrs);

Parameters

```
PIO_Handle  pio;    /* pointer to channel's PIO object */
PIP_Handle  pip;    /* pointer to channel's PIP object */
String      name;   /* name of the IOM device driver */
Int         mode;   /* input, output, or input/output mode */
PIO_Attrs   *attrs; /* pointer to attributes data structure */
```

Return Value

Void

Reentrant

No

Description

PIO_new initializes the PIO object associated with a new channel, and calls the mdCreateChan function of the IOM mini-driver for the channel. PIO_new must be called to initialize the I/O prior to any transfers occurring. PIO_new is typically called from main.

Pass the pointer to the PIO object and the pointer to the PIP object to associate with that channel.

The last argument is for various attributes, including a generic argument to be passed to IOM open. If no attributes are specified, the attrs parameter may be NULL.

If the mdCreateChan call fails, then PIO_new calls SYS_abort.

Constraints and Calling Context

- PIO_new must be called to initialize the I/O prior to any transfers.

Example

The following calls bind the Rx and Tx PIPs to IOM channels:

```
/* Bind PIPs to channels using the PIO class drivers */
PIO_new(&pioRx, &pipRx, "/codec", IOM_INPUT, NULL);
PIO_new(&pioTx, &pipTx, "/codec", IOM_OUTPUT, NULL);
```

PIO_rxPrime

Supply a frame to the receiver

C Interface

Syntax

PIO_rxPrime (pio);

Parameters

PIO_Handle pio; /* pointer to channel's PIO object */

Return Value

Void

Reentrant

No

Description

PIO_rxPrime submits one frame to the IOM mini-driver if the receiver can accept it. The receiver's ability to accept another frame is determined by comparing the number of frames submitted with the maximum number of frames the mini-driver can accept ("submit limit"). The submit limit is initially set to two (2) frames. The submit limit is updated to reflect the maximum frames the mini-driver can accept if and when the mini-driver fails to accept a frame.

Constraints and Calling Context

- PIO_rxPrime is typically configured as the notifyWriter function of receive pipes.

Example

The following example calls PIO_rxPrime:

```
PIO_rxPrime(&pioRx);
```

PIO_rxStart

Start receive with one or more frames

C Interface

Syntax

PIO_rxStart (pio, frameCount);

Parameters

```
PIO_Handle  pio;          /* pointer to channel's PIO object */
Uns         frameCount;   /* number of frames to submit */
```

Return Value

Void

Reentrant

No

Description

PIO_rxStart submits as many frames as the mini-driver can accept, up to the number specified by frameCount.

Constraints and Calling Context

- ❏ PIO_rxStart may only be called in main before interrupts are enabled.

Example

The following example primes the receive side with empty buffers to be filled:

```
PIO_rxStart(&pioRx, PIP_getWriterNumFrames(&pipRx));
```

PIO_txPrime

Supply a frame to the transmitter

C Interface

Syntax

PIO_txPrime (pio);

Parameters

PIO_Handle pio; /* pointer to channel's PIO object */

Return Value

Void

Reentrant

No

Description

PIO_txPrime submits one frame to the IOM receiver if the transmitter can accept it. The transmitter's ability to accept another frame is determined by comparing the number of frames submitted with the maximum number of frames the mini-driver can accept ("submit limit"). The submit limit is initially set to two (2) frames. The submit limit is updated to reflect the maximum frames the mini-driver can accept if and when the mini-driver fails to accept a frame.

Constraints and Calling Context

- PIO_txPrime is typically configured as the notifyReader function of transmit pipes.

Example

The following example calls PIO_txPrime:

```
PIO_txPrime(&pioTx);
```

PIO_txStart

Start transmit with one or more frames

C Interface

Syntax

PIO_txStart (pio, frameCount, Uns initialValue);

Parameters

```
PIO_Handle  pio;          /* pointer to channel's PIO object */
Uns         frameCount;   /* number of frames to submit */
Uns         initialValue; /* value used to fill frame */
```

Return Value

Void

Reentrant

No

Description

PIO_txStart submits as many frames as the mini-driver can accept, up to the number specified by frameCount.

Constraints and Calling Context

- PIO_txStart may only be called in main before interrupts are enabled.

Example

The following example primes the transmit side with buffers of silence:

```
PIO_txStart(&pioTx, PIP_getWriterNumFrames(&pipTx), 0);
```

rxCallback

Notify PIO of receive completion

C Interface

Syntax

rxCallback (arg, *packet);

Parameters

```
Arg          arg;      /* pointer to channel's PIO object */
IOM_Packet   *packet;  /* packet pointing to buffer to be processed */
```

Return Value

Void

Reentrant

No

Description

The rxCallback function completes the operation started by the call to the IOM mini-driver's mdSubmitChan function. The pipe associated with the receive channel is updated to show the received data. Another mdSubmitChan operation is started to keep the flow of data uninterrupted.

The rxCallback function is typically called from the ISR routine that services the associated IOM mini-driver.

Constraints and Calling Context

- ❑ Interrupts for the channel must be disabled.

txCallback

Notify PIO of transmit completion

C Interface

Syntax

txCallback (arg, *packet);

Parameters

```
Arg          arg;     /* pointer to channel's PIO object */
IOM_Packet   *packet; /* packet pointing to buffer to be processed */
```

Return Value

Void

Reentrant

No

Description

The txCallback function completes the operation started by the call to the IOM mini-driver's mdSubmitChan function. The pipe associated with the transmit channel is updated so the frame can be reused. Another mdSubmitChan operation is started to keep the flow of data uninterrupted.

The txCallback function is typically called from the ISR routine that services the associated IOM mini-driver.

Constraints and Calling Context

- ❑ Interrupts for the channel must be disabled.

Appendix C

Porting from the LIO to IOM Model

This appendix compares the LIO and IOM models and explains how users of the LIO device driver model can migrate applications to use the updated IOM device driver model.

Topic **Page**

C.1 Comparing the LIO and IOM Models C–2
C.2 Migrating an LIO Application to Use an IOM Mini-Driver C–4
C.3 Migrating an LIO Controller to an IOM Mini-Driver C–5

C.1 Comparing the LIO and IOM Models

In February 2002, Texas Instruments released an application note titled *Writing DSP/BIOS Device Drivers For Block I/O* (SPRA802). This application note introduced a standard method for implementing block I/O device drivers (such as those used with streaming audio codecs) to work with applications using DSP/BIOS.

The DSP/BIOS IOM device driver model is quite similar to the LIO model. In fact, LIO was the starting point for designing the IOM interfaces. However, unlike LIO, the IOM model was defined to accommodate not just block I/O device drivers, but also to support devices such as video ports, UART interfaces, and bus drivers (such as PCI and USB).

An important point to remember is that both LIO and IOM device drivers can co-exist in a single system. There should be no problem in integrating drivers of both types to be used with a single application.

Note:

Because of the IOM model's greater flexibility, we recommend its use for the implementation of new drivers going forward.

C.1.1 Review of LIO Concepts

The LIO device driver model is very similar to what has been described in this manual. That is, an LIO device "controller" (which the IOM model calls a mini-driver) has a standard set of interfaces so that it can be used by DSP/BIOS applications. Two adapters called DLIO and PLIO (which are similar to the IOM model's DIO and PIO adapters) allow the controller to work with SIO-based and PIP-based applications, respectively.

Table 5-1. Component Comparison for LIO and IOM

Description	LIO Components	IOM Components
DSP/BIOS I/O Modules	SIO or PIP	SIO or PIP
Generic portion	device adapter (DLIO or PLIO)	adapter (DIO or PIO)
Hardware-specific portion	device controller	mini-driver

The LIO adapters (DLIO and PLIO) call the controller when a new buffer is submitted by the application for reading or writing. The controller then presents this buffer to the underlying device hardware—usually through an interrupt service routine. When the ISR is finished with the buffer, it

signals the adapter via a "callback", meaning that it calls a user-specified function appropriate for the operation being performed (read or write). The controller also provides other functionality, such as initializing the device and aborting data buffers that are still owned by the device driver.

C.1.2 LIO Adapters vs. IOM Class Drivers

As mentioned earlier, both the LIO and IOM models specify a two-level architecture. This consists of an adapter and a controller in the LIO case, and a class driver (which includes an adapter) and a mini-driver in the IOM case. Both models use this architecture to simplify the implementation process and to maximize the re-use of code across different driver instances.

One important architectural difference is that, unlike LIO controllers, IOM mini-drivers must be implemented to handle multiple submit calls and so need to queue read and write requests. As a result, IOM class drivers are smaller in size than LIO adapters, while IOM mini-drivers are larger in size than LIO controllers. This change was made for performance reasons. Rather than executing the callback and submit functions in the ISR (as LIO drivers do), jobs are queued in the mini-driver.

C.1.3 LIO Interface vs. IOM Interface Functions

The LIO controller interface and the IOM mini-driver interface have many similarities. Table 5-2 shows the individual interface calls for both and how one maps into the other:

Table 5-2. Function Comparison for LIO and IOM

LIO Controller Functions	IOM Mini-Driver Functions	Description
<controller>_init()	XXX_init()	Static driver initialization
<controller>_setup()	mdBindDev()	Driver setup
open()	mdCreateChan()	Open a device channel
close()	mdDeleteChan()	Close a device channel
submit()	mdSubmitChan()	Submit a buffer for I/O
cancel()	mdSubmitChan()	Cancel a buffer in process
ctrl()	mdControlChan()	Driver control functions
ISR	ISR	Interrupt service routine

As Table 5-2 indicates, most LIO controller functions have a direct analogue in the IOM interface definitions. Channel creation and deletion are handled in a similar manner, as is driver control. The buffer submit() function, which is the heart of device driver functionality, is also similar between the two models. The IOM mdSubmitChan call also allows Flush and Abort commands to be passed.

C.2 Migrating an LIO Application to Use an IOM Mini-Driver

This section does not provided detailed step-by-step descriptions for turning an application that uses an LIO controller into an application that uses an IOM mini-driver. The following are some important points to remember if you are going through this process.

C.2.1 Configuration

Certain objects used by both LIO and IOM device drivers are configured through the DSP/BIOS Configuration Tool. The key difference between configuring the two is support for GIO configuration. You can see a GIO module icon in the Input/Output tree. If you wish, you can configure the GIO module's default blocking mechanism to use a something other than the DSP/BIOS SEM module.

C.2.2 Initialization

LIO device drivers are initialized by explicitly calling the *driver*_init() function followed by the *driver*_setup() function. The first function runs the driver's hardware initialization code, while the second performs the basic setup of the driver's data structures to either default values or to user-specified values depending on what was passed as an argument to the function.

Using IOM device drivers does not require explicit calls to *driver*_init() and *driver*_setup() functions. Instead, the *driver*_init() function is called automatically during the DSP/BIOS initialization sequence. The IOM mini-driver can also set up data structures based on either default values or values passed by the class driver. The mini-driver function mdBindDev() handles this and is automatically called by DSP/BIOS after each configured device’s *driver*_init() function.

C.3 Migrating an LIO Controller to an IOM Mini-Driver

This section does not provided detailed step-by-step descriptions for turning an LIO controller into an IOM mini-driver. The following are some important points to remember if you are going through this process:

- Refer to Table 5-2 to see how LIO controller functions map to the IOM mini-driver functions. For the most part, functions should be able to be converted in a straightforward manner.
- Unlike LIO controllers, IOM mini-drivers must be able to queue up mdSubmitChan calls that are made before the previous I/O submit job has finished.
- IOM mini-drivers receive an I/O packet instead of a raw buffer pointer. This packet contains not only a pointer to an application supplied buffer, but also additional information such as a command field, and size and callback information.
- Unlike LIO controllers, IOM mini-drivers can handle Flush and Abort commands in the mdSubmitChan call to allow for purging pending input and output processing.

Both the LIO application note (SPRA802) and the IOM DDK product provide example codec drivers for the ‘C5402 and ‘C6711 DSKs. You can use these implementations to compare and contrast the two device driver models.

Appendix D

The ASYNC Extension to the GIO API

This appendix describes the ASYNC extension to the GIO API.

Topic **Page**

D.1 ASYNC Module Overview D–2
ASYNC_abort D–5
ASYNC_control D–6
ASYNC_create D–7
ASYNC_delete D–9
ASYNC_flush D–10
ASYNC_read D–11
ASYNC_write D–13

D.1 ASYNC Module Overview

The ASYNC macros are provided with the Driver Developer's Kit for use in applications where threads cannot block for synchronization.

Functions

The ASYNC module provides the following macros:

- ASYNC_abort. Abort all input and output.
- ASYNC_control. Perform device-specific control call.
- ASYNC_create. Allocate and initialize a GIO object.
- ASYNC_delete. Delete underlying mini-drivers and free GIO object and its structures.
- ASYNC_flush. Drain output buffers and discard any pending input.
- ASYNC_read. Submit an asynchronous (non-blocking) read request to mini-driver.
- ASYNC_write. Submit an asynchronous (non-blocking) write request to mini-driver.

These macros correspond to the GIO API functions, which are described in the DSP/BIOS API Reference for your DSP family.

Description

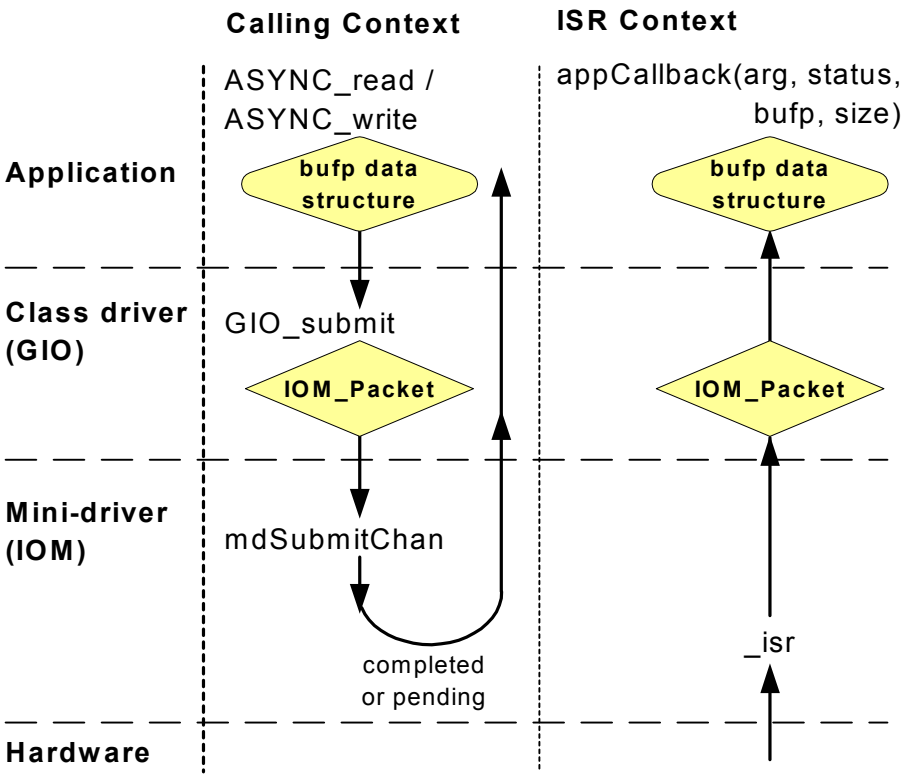
The ASYNC module supports asynchronous (non-blocking) use of the GIO APIs and mini-drivers. Multiple requests can be sent to a mini-driver without causing the thread to block while waiting for resources. This is necessary if the thread that reads from or writes to the mini-driver is a SWI (software interrupt) thread, which cannot block.

Non-blocking calls are accomplished through the use of an application-specified callback function. This function runs when an I/O request made by ASYNC_read or ASYNC_write has been completed.

When a callback function is specified, the GIO class driver does not perform its usually synchronization using a SEM_pend and SEM_post (or whatever synchronization functions are specified in the GIO configuration properties). Since the GIO class driver does not perform synchronization when a callback function is used, the application is responsible for synchronization.

Figure 5-2 shows the call flow resulting from ASYNC_read or ASYNC_write as it passes through the ASYNC, GIO, and mini-driver (IOM) levels of an application.

Figure 5-2. Call Flow for ASYNC_read and ASYNC_write



In applications that use SWI threads, the callback function contains a call to SWI_andnHook or a similar API that posts a SWI after I/O is complete.

Mini-drivers typically do not perform any blocking internally. The intent of the mini-driver architecture is to keep mini-drivers as simple as possible.

Note:

ASYNC_abort and ASYNC_flush are synchronous functions. They may block while waiting for requests to be aborted or flushed. As a result, these functions must be called from the context of TSK thread.

The ASYNC module provides macros built on top of GIO functions. While one could use the GIO_submit function with its callback parameter

instead of the ASYNC_read and ASYNC_write functions, this module is provided to clarify the intent of an application that uses the IOM model and mini-drivers asynchronously.

Constants, Types, and Structures

Uses GIO module constants, types, and structures. See Section A.1, *Mini-Driver Interface Overview*.

ASYNC_abort

Abort all input and output

C Interface

Syntax

```
status = ASYNC_abort(gioChan);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

ASYNC_abort is a synchronous call. It returns only when all I/O has been successfully aborted.

The gioChan parameter is a handle to the GIO object created by the call to ASYNC_create.

When this call is made, all pending packets are completed with a status of IOM_ABORTED. An application uses this call to return the device to its initial state. Usually this is done in response to an unrecoverable error at the device level.

ASYNC_abort returns IOM_COMPLETED upon successfully aborting all input and output requests. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_abort calls GIO_abort, which calls the associated mini-driver's mdSubmitChan function. The IOM_ABORT command is passed to the mdSubmitChan function.

Constraints and Calling Context

- ❏ This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.
- ❏ ASYNC_abort cannot be called from a SWI or HWI unless the GIO configuration properties are set to use non-blocking synchronization methods.

ASYNC_control *Perform device-specific control call*

C Interface

Syntax

```
status = ASYNC_control(gioChan, cmd, args);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
Int         cmd;      /* control functionality to perform */
Ptr         args;     /* data structure to pass control information */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

An application calls ASYNC_control to configure or perform control functionality on the communication channel.

The gioChan parameter is a handle to the GIO object created by the call to ASYNC_create.

The cmd parameter may be one of the command codes for GIO_control listed in Section A.1, *Mini-Driver Interface Overview*. A mini-driver may add command codes for additional functionality.

The args parameter points to a data structure defined by the device to allow control information to be passed between the device and the application. This structure can be generic across a domain or specific to a single mini-driver. In some cases, this argument may point directly to a buffer that holds the control data. In other cases, there may be a level of indirection if the mini-driver expects a data structure to package many components of data required for the control operation. In the simple case where no data is required, this parameter may just be a predefined command value.

ASYNC_control returns IOM_COMPLETED upon successfully carrying out the functionality requested. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_control calls GIO_control, which calls the associated mini-driver's mdControlChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.

ASYNC_create *Allocate and initialize a GIO object*

C Interface

Syntax

```
gioChan = ASYNC_create(name, mode, *status, optArgs, *attrs)
```

Parameters

```
String      name      /* name of the device to open */
Int         mode      /* mode in which the device is to be opened */
Int         *status   /* address location to place driver return status */
Ptr         optArgs   /* optional domain/device-specific arguments */
GIO_Attrs   *attrs    /* pointer to an GIO_Attrs structure */
```

Return Value

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
```

Reentrant

Yes

Description

ASYNC_create allocates and initializes a GIO object.

The name argument is the name specified for the device when it was created in the configuration or at runtime. It is used to find a matching name in the device table.

The mode argument specifies the mode in which the device is to be opened. This may be IOM_INPUT, IOM_OUTPUT, or IOM_INOUT.

If the status parameter is non-NULL, a status value is placed at the address specified by the status param.

The optArgs parameter is a pointer that may be used to pass device or domain-specific arguments to the mini-driver. The contents at the specified address are interpreted by the mini-driver in a device-specific manner.

The attrs parameter is a pointer to a structure of type GIO_Attrs.

```
typedef struct GIO_Attrs  {
  Int  nPackets; /* number of asynch I/O packets */
  Uns  timeout;  /* for blocking calls (SYS_FOREVER) */
} GIO_Attrs;
```

The ASYNC_create call allocates a list of IOM_Packet items as specified by the nPackets member of the GIO_Attrs structure and stores them in the GIO object it creates.

ASYNC_create returns a handle to the GIO object created upon a successful open. The handle returned by this call should be used by the application in subsequent calls to ASYNC functions. This function returns

a NULL handle if the device could not be opened. For example, if a device is opened in a mode not supported by the device, this call returns a NULL handle.

Internally, ASYNC_create calls GIO_create, which calls the associated mini-driver's mdCreateChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized.

Example

The following example uses ASYNC_create:

```
/* create GIO objs */
for (i = 0; i < CHAN_NUM; i++) {
   gio[i] = ASYNC_create("/vt1423pci", IOM_INOUT,
                  NULL, &pciAttrs[i], &gioAttrs[i]);
   if (gio[i] == NULL) {
     LOG_printf(&trace, "ERROR!!! GIO_create NULL!");
   }
}
```

ASYNC_delete

Delete underlying mini-drivers and free GIO object and its structures

C Interface

Syntax

```
status = ASYNC_delete(gioChan);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to device instance to be closed */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

ASYNC_delete closes a communication channel opened prior to this call with ASYNC_create. This function deallocates all memory allocated for this channel and closes the underlying device. All pending input and output are cancelled and the corresponding interrupts are disabled.

The gioChan parameter is the handle returned by ASYNC_create.

This function returns IOM_COMPLETED if the channel is successfully closed. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_delete calls GIO_delete, which calls the associated mini-driver's mdDeleteChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.

ASYNC_flush

Drain output buffers and discard any pending input

C Interface

Syntax

```
status = ASYNC_flush(gioChan);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

An application calls ASYNC_flush to flush the input and output channels of the device. All input data is discarded; all pending output requests are completed. When this call is made, all pending input calls are completed with a status of IOM_FLUSHED, and all output calls are completed routinely.

ASYNC_flush is a synchronous call. It returns only when all I/O has been successfully flushed.

The gioChan parameter is the handle returned by ASYNC_create.

This call returns IOM_COMPLETED upon successfully flushing all input and output. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_flush calls GIO_flush, which calls the associated mini-driver's mdSubmitChan function. The IOM_FLUSH command is passed to the mdSubmitChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.
- ASYNC_flush cannot be called from a SWI or HWI unless the GIO configuration properties are set to use non-blocking synchronization methods.

ASYNC_read

Submit an asynchronous (non-blocking) read request to mini-driver

C Interface

Syntax

```
status = ASYNC_read(gioChan, bufp, *pSize, *appCallback);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
Ptr         bufp      /* pointer to data structure for buffer data */
Uns         *pSize    /* pointer to size of bufp structure */
GIO_AppCallback  *appCallback  /* pointer to callback structure */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

An application calls ASYNC_read to asynchronously read a specified number of bytes from the communication channel.

The gioChan parameter is the handle returned by ASYNC_create.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the read data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be read from, and other device-dependent data. For example, for video capture devices this structure may contain pointers to R, G, B buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of bytes read from the device. This parameter relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

In order for ASYNC_read to be performed asynchronously, the appCallback parameter must point to a callback structure that contains the callback function to be called when the queued request is completed.

ASYNC_read returns IOM_COMPLETED upon successfully reading the requested number of bytes from the device. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_read calls GIO_submit, which calls the associated mini-driver's mdSubmitChan function. The IOM_READ command is passed to the mdSubmitChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.

Example

The following example uses ASYNC_read:

```
status = ASYNC_read(gio[chanNum], curReq, &size,
                    &gioCallback[submitReqIndex]);
```

ASYNC_write

Submit an asynchronous (non-blocking) write request to mini-driver

C Interface

Syntax

```
status = ASYNC_write(gioChan, bufp, *pSize, *appCallback);
```

Parameters

```
GIO_Handle  gioChan;  /* handle to an instance of the device */
Ptr         bufp      /* pointer to data structure for buffer data */
Uns         *pSize    /* pointer to size of bufp structure */
GIO_AppCallback  *appCallback  /* pointer to callback structure */
```

Return Value

```
Int         status;   /* returns IOM_COMPLETED if successful */
```

Reentrant

Yes

Description

The application uses this function to write a specified number of bytes to the communication channel.

The gioChan parameter is the handle returned by GIO_create.

The bufp parameter points to a device-defined data structure for passing buffer data between the device and the application. This structure may be generic across a domain or specific to a single mini-driver. In some cases, this parameter may point directly to a buffer that holds the write data. In other cases, this parameter may point to a structure that packages buffer information, size, offset to be written to, and other device-dependent data. For example, for video capture devices this structure may contain pointers to R, G, B buffers, their sizes, video format, and a host of data required for reading a frame from a video capture device. Upon a successful read, this argument points to the returned data.

The pSize parameter points to the size of the buffer or data structure pointed to by the bufp parameter. When the function returns, this parameter points to the number of bytes written to the device. This parameter relevant only if the bufp parameter points to a raw data buffer. In cases where it points to a device-defined structure it is redundant—the size of the structure is known to the mini-driver and the application. At most, it can be used for error checking.

In order for ASYNC_write to be performed asynchronously, the appCallback parameter must point to a callback structure that contains the callback function to be called when the queued request is completed.

GIO_write returns IOM_COMPLETED upon successfully writing the requested number of bytes to the device. If an error occurs, the device returns a negative value. For a list of error values, see Section A.1, *Mini-Driver Interface Overview*.

Internally, ASYNC_write calls GIO_submit, which calls the associated mini-driver's mdSubmitChan function. The IOM_WRITE command is passed to the mdSubmitChan function.

Constraints and Calling Context

- This function can be called only after the device has been loaded and initialized. The handle supplied should have been obtained with a prior call to ASYNC_create.

Example

The following example uses ASYNC_write:

```
status = ASYNC_write(gio[chanNum], curReq, &size,
                     &gioCallback[submitReqIndex]);
```

Appendix E

Glossary

This appendix provides definitions for terms related to device drivers.

- **adapter.** The interface between application-level calls and a mini-driver. The term adapter was also used for this purpose in the LIO model. The adapter is part of a class driver. An adapter may or may not provide an API.
- **ASYNC module.** An extension to the GIO module for use when asynchronous I/O processing is needed.
- **asynchronous.** Processing that occurs without blocking.
- **blocking.** Waiting for a resource to be available or for the end of a timeout if one was specified. TSK threads can block, and therefore can be processed synchronously. SWI and HWI threads cannot, and therefore must be processed asynchronously.
- **channel.** The type of object managed by a mini-driver. Each channel corresponds to an instance of a device.
- **Chip Support Library (CSL).** The API library provided with Code Composer Studio for interaction with on-chip and off-chip peripherals such as McBSPs. The library is different for each chip family. The objects used by the CSL can be configured in the DSP/BIOS Configuration Tool or in a DSP/BIOS Tconf script.
- **class driver.** The interface between application-level calls and a mini-driver. A class driver may include both an API module and an adapter (for example, PIP/PIO or SIO/DIO). Alternately, a class driver may be implemented as an API interface only (for example, GIO).
- **controller.** The term used for the device-specific portion of a driver in the LIO model. The corresponding portion of the IOM model is the mini-driver.
- **DEV module.** A DSP/BIOS module that manages user-defined devices that must be configured in order to use an IOM mini-driver. The DEV module also specified a DEV_Fxns structure, which is not used with the IOM model.
- **device.** A hardware input or output peripheral accessible by the DSP.

- **DIO adapter.** The adapter to use when the application makes SIO calls for stream I/O. This adapter does not include API functions.
- **DLIO adapter.** The SIO-based adapter provided with the LIO model.
- **DMA.** (Direct Memory Access) A mechanism whereby a device other than the host processor contends for, and receives, mastery of the memory bus so that data transfers can take place independent of the host.
- **driver.** A software mechanism for sending input or output data to a device.
- **Driver Developer's Kit (DDK).** The software distribution that provides files for creating an integrating mini-drivers that use the IOM model.
- **EDMA.** (Enhanced DMA) A controller that acts as a highly-efficient data transfer engine, controlling all of the data movement beyond the level-two memory of the device.
- **GIO module.** A DSP/BIOS module that provides an extensible class driver API. An application may call GIO module APIs to interface to a mini-driver.
- **HWI module.** A DSP/BIOS module used to manage hardware interrupt threads (also known as ISRs). Hardware interrupt threads cannot block while waiting for resources.
- **IOM interface.** The module used to implement a mini-driver. Several IOM structures are also used by the class-driver. The entire class driver/mini-driver model described in this book is called the "IOM model".
- **IOM_Fxns table.** The table structure that references the functions implemented for a particular mini-driver.
- **IOM_Packet.** A structure created by the class driver and used to pass commands, data buffers, and status information between the class driver and mini-driver.
- **LIO module.** The module that corresponds to IOM in the LIO device driver model.
- **McBSP.** (Multi-Channel Buffered Serial Port) Typically the EDMA or DMA is used to perform read/write transfers from/to the McBSP. These transfers are read/write synchronized and the McBSP provides these synchronization events.
- **McASP.** (Multi-Channel Audio Serial Port) This port functions as a general-purpose audio serial port optimized for the needs of multichannel audio applications.

- **mini-driver.** The term used for the device-specific portion of a driver in the IOM model. Mini-drivers are implemented using the IOM module.
- **PCI controller.** (Peripheral Component Interconnect) A high-speed local bus that supports data-transfer operating at 33 MHz or 66 MHz.
- **PIO adapter.** The adapter to use when the application makes PIP calls for stream I/O. This adapter includes API calls that the application must make.
- **PIP module.** A DSP/BIOS module used to manage pipe I/O objects. Pipe objects can have callback functions specified for the reader and writer. Pipe objects are typically used with SWI threads and non-blocking execution.
- **PLIO adapter.** The PIP-based adapter provided with the LIO model.
- **queue.** An object type managed by the DSP/BIOS QUE module. Queues maintain a list of elements that can be manipulated in various ways.
- **semaphore.** An object type managed by the DSP/BIOS SEM module. A semaphore synchronizes access to resources by allowing various threads to pend on and post the semaphore.
- **SIO module.** A DSP/BIOS module used to manage stream I/O objects. Stream objects are typically used with TSK threads and blocking execution, though use with SWI threads is now possible.
- **SWI module.** A DSP/BIOS module used to manage software interrupt threads. Software interrupt threads cannot block while waiting for resources.
- **synchronous.** Processing that occurs with blocking while waiting for resource availability.
- **TSK module.** A DSP/BIOS module used to manage task threads. Task threads can block while waiting for resources.
- **UART.** (Universal Asynchronous Receiver-Transmitter) A UART handles asynchronous serial communication. A key component of the serial communications port.
- **UDEV object.** An object type managed by the DEV module.

Index

A

abort requests 5-4
adapter 2-11, E-1
addr field 2-8
application developer 1-2
architecture 2-3
arg field 2-8
ASYNC module 4-2, D-1, E-1
ASYNC_abort D-5
ASYNC_control D-6
ASYNC_create D-7
ASYNC_delete D-9
ASYNC_flush D-10
ASYNC_read D-11
ASYNC_write D-13
asynchronous E-1
audio folder 1-8

B

benefits 1-4
binding 2-4
BIOS_INSTALL_DIR environment variable 1-3
blocking E-1
BSL_INSTALL_DIR environment variable 1-3

C

call flow 2-5, 2-9
callback functions 2-12, 2-13, B-3
cbArg field 5-4
cbFxn field 5-4
channel E-1
channel handle 2-6
channel instances 2-4
channel operations 2-8
channel parameters 3-6
ChanObj structure 5-3, 5-7
Chip Support Library (CSL) E-1
class driver 1-4, 1-5, 2-2, 2-11, E-1
cmd field 2-8
controller C-2, E-1
Create all DIO Objects Statically 3-5
Create Function 3-10
CSL_INSTALL_DIR environment variable 1-3

D

data flow 2-5
DDK 1-3
DDK_INSTALL_DIR environment variable 1-3
debug folder 1-8
Delete Function 3-10
DEV module 2-11, E-1
DEV_Fxns table type 3-3
device E-1
device control 2-10
device global data ptr 3-3
device id 3-3
device instances 2-4
device name 3-6
device params ptr 3-3
devid parameter 5-2
devParams parameter 5-7
DIO adapter 2-11, 3-4, E-2
DLIO adapter C-2, E-2
DMA E-2
doc folder 1-8
driver E-2
driver developer 1-2
Driver Developer's Kit (DDK) 1-3, E-2
DSP/BIOS
  environment variables 1-3
  version requirements 1-3

E

EDMA E-2
Enable General Input/Output Module 3-10
environment variables 1-3
error codes 4-5

example applications 1-8
examples folder 1-8

F

file system stacks 4-2
flush requests 5-4
folders 1-6
frame video module (FVID) 4-5
freeList field 4-4
function table pointer 3-3
function table type 3-3
FVID module 4-5
FVID_alloc 4-6
FVID_exchange 4-6
FVID_free 4-6

G

GIO class driver 1-5, 2-13, 3-10, 4-2
GIO module 3-10, E-2
GIO_Obj structure 4-4
GIO_submit
  video macros 4-6
GIO_submit macros D-2
global device data 5-3

H

HWI module E-2

I

init function 3-3
initialization 2-4, 5-6, 5-7
installation 1-3
IOM interface A-1, E-2
IOM model C-1
IOM_ABORT 5-4
IOM_CHAN_RESET 5-5
IOM_CHAN_TIMEDOUT 5-5
IOM_CNTL_USER 5-5
IOM_DEVICE_RESET 5-5
IOM_ENOTIMPL 2-10, 5-5
IOM_FLUSH 5-4
IOM_Fxns structure 5-6, A-2
IOM_Fxns table E-2
IOM_Fxns table type 3-3
IOM_Packet 2-7, E-2
IOM_READ 5-4
IOM_WRITE 5-4
ISR functions
  examples 5-16
ISRs 5-4

L

library files 1-8
link field 2-7
LIO device controller C-2
LIO model C-1
LIO module E-2

M

McASP E-2
McBSP E-2
  multiple instances 2-5
mdBindDev function 5-2, A-5
  example 5-8
mdChan field 4-4
mdControlChan function 5-5, A-6
  example 5-10
mdCreateChan function 5-3, A-7
  example 5-11
mdDeleteChan function 5-3, A-9
  example 5-13
mdSubmitChan function 5-4, A-10
  example 5-13
mdUnBindDev function A-12
  example 5-15
migration
  application C-4
  LIO controller C-5
mini-driver 1-4, 2-2, E-3
  registering 3-2
misc field 2-8
mode field 4-4, 5-3
multiple device instances 5-2

N

notifyReader function 2-12, 3-8
notifyWriter function 2-12, 3-8

P

package folders 1-8
packets 2-7
partitioning 2-2

PCI controller E-3
Pend Function 3-10
pendList field 5-3
PIO adapter 2-12, 3-7, B-1, E-3
pio folder 1-8
PIO module 3-9
PIO_ctrl B-5
PIO_init B-6
PIO_new B-7
PIO_rxPrime B-8
PIO_rxStart B-9
PIO_txPrime B-10
PIO_txStart B-11
PIP module 2-11, E-3
PIP objects 3-8
PIP/PIO class driver 1-5, 2-12, 3-7
pip_audio application 1-9, 3-7
pipes 2-12
pjt files 1-8
PLIO adapter C-2, E-3
Post Function 3-10
prime functions 2-13, B-3
project files 1-8

Q

queue E-3

R

registering mini-driver 3-2
release folder 1-8
rxCallback function B-12
rxIsr function
  example 5-16

S

sample-by-sample device driver 5-6
semaphore E-3
setup 1-3
shared folder 1-8
SIO module 2-11, 3-6, E-3
SIO/DIO class driver 1-5, 2-11, 3-4
size field 2-8
status field 2-8
streaming 2-11
SWI module E-3
SWI objects 3-8
swi_audio application 1-9
synchronous E-3
syncObj field 4-4
syncPacket field 4-4

T

timeout field 4-4
transfer function 2-12, 2-13, B-3
TSK module E-3
TSK objects 3-6
tsk_audio application 1-8, 3-4
txCallback function B-13
txIsr function
  example 5-18

U

UART drivers 4-2, E-3
uart folder 1-9
uarttest application 1-9, 3-11
UDEV object 3-2, E-3
use callback version of DIO functions 3-6

V

video API 4-5
video capture drivers 4-2
video display drivers 4-2