DSP/BIOS I2C Device Driver

USER'S GUIDE

Document Revision History

Rev No	Author(s)	Revision History	Date	Approval(s)
0.8	Chandan Nath	Updated for adding compiler switches in build options	May 20, 2008	Updating
0.7	Nagarjuna K	Updated for DM6437/C6424 and DM648/C6453	November 14, 2007	Updating
0.6	Nagarjuna K	Updated BIOS and XDC versions	October 22, 2007	Updating
0.5	Nagarjuna K	Updating directory structure	July 10, 2007	Updating
0.4	Nagarjuna K	Updating for 0.7 release with changes of RTSC packaging	July 6, 2007	Corrections
0.3	Nagarjuna K	Correcting version numbers	June 15, 2007	Corrections
0.2	Nagarjuna K	Added to new formatting	May 23, 2007	Initial Draft
0.1	Vichu	Formatted to new template	May 9, 2007	Draft

implied or expressed warranties in this document and is not responsible for the products based from this document

TABLE OF CONTENTS

1 Introduction 5

1.1 Terms & Abbreviations 5

1.2 References 5

1.3 S/W Support 6

1.4 Driver Components 6

1.5 Driver Capabilities 7

1.6 System Requirements 7

2 Installation Guide 8

2.1 Component Folder 8

2.2 Build 9

2.3 Build Options 9

3 DSP/BIOS I2C DRIVER Structures 11

3.1.1 Initialization details 11

3.1.2 I2C_devParams 12

3.1.3 I2C Chan Params 12

4 I2C API's 12

4.1 Constants & Enumerations 13

4.1.1 Structure for data parameter passed to GIO_submit 13

4.1.2 Structure for i2cTrans in above data param 13

4.1.3 Enum for IOCTL 13

4.2 API Definition 14

4.2.1 GIO_ create 14

4.2.2 GIO_delete 15

4.2.3 GIO_control 16

4.2.4 GIO_Submit 17

4.2.5 GIO_read/GIO_write 18

4.2.6 GIO_flush/GIO_abort 19

5 Example Applications 20

5.1 Writing Applications for I2C 20

5.1.1. File Inclusion 20

5.2 Sample Applications 20

TABLE OF FIGURES

Figure 1. I2C driver architecture.....6
Figure 2. I2C Driver Directory Structure 8
Figure 3. I2C sample directory structure for DM648 and C6452 21
Figure 4. I2C sample directory structure for DM6437 and C6424 23

1 Introduction

This document is the reference guide for the i2c driver and it explains how to configure and use the driver.

DSP/BIOS applications use the driver typically through I2C APIs to perform read/write operations connected to the slaves. I2C was implemented as a simple wrapper on top of the GIO class driver and provides an application-specific interface. For more information on the DSP/BIOS device driver model and the GIO class driver, refer to the References section of this document.

1.1 Terms & Abbreviations

Term	Description
⚐	This bullet indicates important information. Please read such text carefully.
❑	This bullet indicates additional information.
API	Application Programming Interface
DDC	Device Driver Core
IOM	Device Driver Adapter
ISR	Interrupt Service Routine
OS	Operating System
ROM	Read Only Memory
SOC	System On Chip

1.2 References

1.	SPRU4.4g.pdf	DSP/BIOS Driver Developer's Guide
3.	spruek8_I2C.pdf	I2C specification

1.3 S/W Support

This I2C device driver has been developed for the DSP/BIOS operating system using the TI supplied Chip Support Library. For more details on the version numbers refer to the release notes in the root of the installation.

1.4 Driver Components

The I2C driver is constituted of following sub components:

I2C IOM – Application facing, OS Specific Adaptation of I2C Device Driver
I2C DDC –OS Independent part of I2C Driver Core
I2C CSLR–The low-level I2C h/w register overlay

System components:

PALOS – DSP/BIOS Abstraction

Below Figure shows I2C driver architecture.

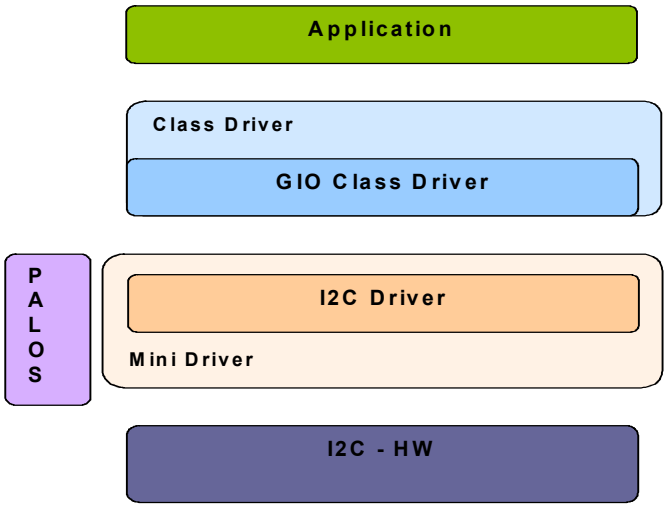

Figure 1. I2C driver architecture

1.5 Driver Capabilities

The DSP/BIOS I2C Device Driver is a multi-instantiable and re-entrant safe driver. In addition, it provides synchronous IO. The driver operates in the following modes: Polled and Interrupts modes. The driver has built-in software Ring Buffer for improved synchronous IO response times.

The significant driver features are:

- Isolates H/W and OS Accesses.
- Easy to maintain & re-target to new platforms.
- Can stack custom-functions along control and or data-path to realize “driver filters”.
- Supports Multiple Instances.

1.6 System Requirements

- Refer to release notes for the details on Target environment, BIOS version, XDC version…

2 Installation Guide

2.1 Component Folder

Upon installing the I2C driver the following directory structure is found in the driver's directory.

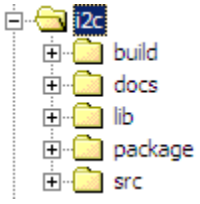

Figure 2. I2C Driver Directory Structure

This top level i2c folder contains i2c driver psp header file and XDC package files (package.bld, package.xdc and package.xs)

- **build:** This folder contains i2c driver library project file. The generated driver library shall be included in the application where I2C driver have to be used.
- **docs:** This folder contains architecture document, datasheet, release notes, user guide and doxugen compiled api reference document.

  Architecture document contains the driver details which can be helpful for the developers as well as consumers to understand the driver design.

  Datasheet gives the idea about the memory consumption by the driver and description of the top level APIs.

  Release Note gives the details about system requirements, steps to Install/Uninstall the package.This document list the known issues of the driver.

  User Guide provides information about how to use the driver. It contains description of sample applications which guide the end user to make their applications using this driver.

  API reference document gives the details about the API's used in I2C driver.
- **Lib:** This folder contains libraries generated in all the configuration modes(debug, idebug, irelease and release)
- **Package:** This folder contains files generated by XDC tool.
- **src:** This folder contains i2c driver source files. It also contains header files that are used by the driver.

2.2 Build

This section describes for each supported target environment, the applicable build options, supported configurations and how selected, the featured capabilities and how enabled, the allowed user customizations for the software to be installed and how the same can be realized.

The component might be delivered to user in different formats:

- Source-less i.e. binary executables and object libraries only.
- Source-inclusive i.e.The entire source code is used to implement the driver is included in the delivered product.
- Source-selective ie. Only a part of the overall source is included. This delivery mechanism might be required either because ;certain parts of the driver require soruce level extensions and/or customization at the user's end or because,specific parts of the driver is exposed to user at the source level to insure user's software development.

When source is included as part of the product delivery, the CCS project file is provided as part of the package. When object format is distributed, the driver header files are part of the "src" folder and the driver library is provided in "*pspdrivers**drivers**i2c**lib"* folder.

2.3 Build Options

To compile driver, change build options as mentioned below:

Optimization level should be configured for –o2 for release and irelease configurations.

The build folder contains a CCS project file that builds the driver into a library for debug, idebug, irelease, release mode.

Following compiler switches are used to compile for different options.

- **_DEBUG**
  This is used as a flag to compiler whether to include the debug statement inserted in the code into the final image. This flag helps to build DEBUG image of the program. For RELEASE images this is not passed to the compiler.
- **CHIP_XXXX**
  The CSL layer is written in a common file for all the variants of a SOC. This flag differentiates the variant we are compiling for, for e.g. - CHIP_DM648, and the CSL definitions for that variant appropriately gets defined for register base addresses, num of ports of a peripheral etc.
- **I2C_INSTRUMENTATION_ENABLED**
  This flag is passed to the compiler to include the instrumentation code parts into the final image/lib of the program. This helps build the iRelease/iDebug versions of the image/lib with a common code base.

- **I2C_DEBUG_PRINTF**
  They are the macros used to print debug messages. These expand a valid print statement.

3 DSP/BIOS I2C DRIVER Structures

This section discusses about the initialization details and initialization structures used in the i2c driver. Please note that for some structure member information/details the i2c peripheral API reference document might need to be referred.

3.1.1 Initialization details

To use I2C device driver, a device entry must be added and configured in the DSP/BIOS configuration tool.

To have I2C device driver included in the application, corresponding TCI file have to be included in BIOS TCF (i.e. *"dm648_i2c.tci" for DM648)* must be included in BIOS TCF file of the application for using I2C instance of the driver.

The following are the device configuration settings required to use the i2c driver.
Note: This has to be done for all of the required driver instances.

TCI Configuration Parameters	Description
initFxn - Init Function	Pointer to application function to initialize i2c like module clock enabling and enabling pin-mux.
fxnTable - Function Table Pointer	*I2C_Fxns*. This is a global variable which points to the I2C driver APIs.
fxnTableType - Function Table Type	*IOM_Fxns*
deviceId - Device Id	Specify which I2C to use. For example to use I2C 0 this should be given as 0.
params – Pointer to Port parameter	A pointer to an object of type *I2C_devParams* as defined in the header file *psp_i2c.h*. This pointer will point to a device parameter structure. In BIOS TCI files, this structure object is passed as an argument. Application should declare and initialize the structure object properly.
Device Global Data Pointer	N/A, not used by this driver

Final tci file should contain the following details which were explained above:

```
bios.UDEV.create("I2C0");

bios.UDEV.instance("I2C0").fxnTableType = "IOM_Fxns";

bios.UDEV.instance("I2C0").initFxn = prog.extern("I2C_INIT");

bios.UDEV.instance("I2C0").params = prog.extern("I2C_devParams");

bios.UDEV.instance("I2C0").fxnTable = prog.extern("I2CMD_FXNS");
```

3.1.2 I2C_devParams

"psp_i2c.h" file contains *PSP_I2cConfig_t* data structure that is passed while mdBindDev call which is defined with UDEV I2C parameters in *.tcf file of application. The members of this structure are explained below:

Structure Members	Description
opMode	Operational mode of the driver – Polled / Interrupt / DMA based. **Note:** The current implementation only supports Polled and Interrupt modes
moduleInputClkFreq	Input Frequency to I2C Module.
i2cBusFreq	Output Data rate, in Kbps, of the I2C controller. **Note:** I2C supports Standard and Fast modes (up to 400Kbps).
i2cOwnAddr	Own address of the device
NumBits	Number of bits to be sent or received
addressing	To set addressing mode 7bit or 10 bit addressing
dlb	To enable or disable digital loop back mode

3.1.3 I2C Chan Params

These are the parameters that are passed while GIO_create call. The members of this structure are currently "NULL"

4 I2C API's

This chapter describes the functions, data structures, enumerations and macros for the I2C driver module.

The following API functions are defined by the GIO module:

GIO_create	Allocate and initialize an I2C channel object
GIO_delete	De-allocate an I2C channel object
GIO_control	Send a control command to the mini-driver
GIO_submit	API used to transfer the data with slaves

4.1 Constants & Enumerations

4.1.1 Structure for data parameter passed to GIO_submit

The file **dda_iom_i2c.h** has the **DataParam** data structure that is passed to **i2c_mdSubmitChan** function of the driver. The params are explained below:

Structure Members	Description
i2cTrans	I2c transaction structure used to pass to submit in the addr parameter of IOM_Packet structure.
Timeout	Timeout value for transfer

4.1.2 Structure for i2cTrans in above data param

The file **psp_i2c.h** has the **PSP_I2cTransaction** data structure that is parameter of DataParam structure mentioned above. The params are explained below:

Structure Members	Description
`slaveAddr`	Address of the slave, master wants to communicate
`*buffer`	Data buffer pointer to where data needs to be received or to transmit
`bufLen`	Length of the data bytes to be receives or to transmit
`Flags`	Flags to indicate the various modes [read/write, start, stop, restart…]
`Param`	Extra parameter for future use

4.1.3 Enum for IOCTL

Following are the enumerations passed as command argument while GIO_control call.

Structure Members	Description
PSP_I2C_IOCTL_SET_BIT_RATE	Set the I2C clock
PSP_I2C_IOCTL_GET_BIT_RATE	Get the I2C clock
PSP_I2C_IOCTL_NACK	To enable or disable NACK
PSP_I2C_IOCTL_BIT_COUNT	To set bit count value
PSP_I2C_IOCTL_CANCEL_PENDING_IO	To cancel pending IO requests if any in the transmission
PSP_I2C_IOCTL_SET_OWN_ADDR	TO change I2C own address.
PSP_I2C_IOCTL_SET_OWN_ADDR	To get I2C own address

4.2 API Definition

4.2.1 GIO_ create

Syntax

```
GIO_Handle GIO_create (
        String name,
        Int mode,
        Int *status,
        Ptr chanParams,
        GIO_Attrs *attrs
        );
```

Parameters

name

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.
Note: strings are case sensitive.

For I2C drivers the string contains one token separated by *'/'*.

- I2C driver or port instance
  This identifies the I2C driver or port instance and this will be typically *"I2C0", "I2C1"* and so on, where suffix to I2C denotes instance ID. This string depends on the device registration string given in BIOS driver TCI file.

mode

The mode argument specifies the mode in which the device is to be opened. This will be IOM_INPUT, IOM_OUTPUT or IOM_INOUT. Generally I2C driver should be created in IOM_INOUT mode as both read/write happens in one channel

status

The status argument is an output parameter that this function fills with a pointer to the status that was returned by the mini-driver.

chanParams

Currently I2C driver does not use this parameter so application passes NULL for this parameter.

attrs

The attrs parameter is a pointer to a structure of type *GIO_Attrs*. This is not supported and NULL should be passed.

Return Value

It returns the handle of type *GIO_Handle* on successful opening of a device. It returns NULL if the device could not be opened.

Description

An application calls *GIO_create* to create and initialize an i2c driver channel to the driver.

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

Example
The example below shows creation of Channel for I2C

```
GIO_Handle chanHandle;
GIO_Attrs gioAttrs = GIO_ATTRS;

chanHandle = GIO_create("/I2C0", IOM_INOUT,
                        NULL, NULL, &gioAttrs);
if (NULL == chanHandle)
{
      printf(" Failed create I2C0 channel \r\n");
      return;
}
```

4.2.2 GIO_delete

Syntax
int GIO_delete(GIO_Handle gioChan);

Parameters
Handle of the i2c driver channel that was created with a call to GIO_create.

Return Value
On success driver returns IOM_COMPLETED, on failure/error condition IOM error code

Description
This function call will close the logical channel associated with GIO_create. It will also free the buffers allocated by driver.

Constraints
This function can only be called after the device has been loaded, initialized and created.

Example
The example below shows creation and deletion of Channel for I2C

```
GIO_Handle chanHandle;
Int status;

/*Create Handle */

status = GIO_delete(chanHandle);
if (IOM_COMPLETED != status)
{
      printf(" Failed to delete channel \r\n");
      return;
}
GIO_delete(chanHandle);
```

4.2.3 GIO_control

Syntax

status = GIO_control (gioChan, cmd, args);

Parameters

gioChan

Handle of the I2C driver channel that was created with a call to *GIO_create*.

cmd

The cmd argument specifies the control command

args

The args argument is a pointer to the argument or structure of arguments that are specific to the command being passed.

Return Value

On success driver returns IOM_COMPLETED, on failure/error condition IOM error code

Description

An application calls *GIO_control* to send device-specific control commands to the mini-driver.

IOCTL commands available for I2C driver are available in section 4.1.3

- For PSP_I2C_IOCTL_NACK: Command argument (cmdArg) value should be either TRUE (enable) or FALSE (disable). Any other value passed as command argument will be treated as invalid parameter, which will result in error.
- For PSP_I2C_IOCTL_BIT_COUNT: Command argument (cmdArg) value should vary from 2 to 8 only. Any other value passed as command argument will be treated as invalid parameter and it will result in error.
- For any other IOCTL cmdArg is don't care parameter.

Constraints

This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *GIO_create*.

Example

```
GIO_Handle chanHandle;
Uint32 bitCount=2;
/* channel creation should be done here */

/* set bit count to 2 */
status = GIO_control(chanHandle, PSP_I2C_IOCTL_BIT_COUNT, &bitCount);

if (IOM_COMPLETED != status)
{
      printf(" Failed to change bit count \r\n");
      return;
}
```

4.2.4 GIO_Submit

Syntax

```
status = GIO_submit (    GIO_Handle gioChan,
                         Uns cmd,
                         ptr bufP,
                         Uns* Psize,
                         GIO_AppCallBack* appCallback
                     )
```

Parameters

gioChan

Handle of the I2C driver channel that was created with a call to *GIO_create*.

cmd

The cmd argument specifies the control command which can be any of the following

- IOM_READ /* for read operation */
- IOM_WRITE /* for write operation */
- IOM_FLUSH /* for flushing the packets */
- IOM_ABORT /* for aborting the packets */

bufP

The bufP argument is a pointer to the structure defined in section 4.1.1, and this value should be used only when cmd is used as IOM_READ or IOM_WRITE. For other two commands it is NULL.

Psize

The Psize argument is a pointer to the argument of data type size_t, currently value passed should be 1.

appCallback

The Psize argument is a function pointer, which will be called once the submit operation is completed. Since I2C works in synchronous mode of operation this parameter will be NULL.

Return Value

On success driver returns IOM_COMPLETED, on failure/error condition IOM error code

Description

This function is called by the application to perform the read/write/flush/abort operation.

Usage of flags:

Valid flag combination options which can be passed to I2C driver are:

- PSP_I2C_DEFAULT_WRITE for performing write
- PSP_I2C_DEFAULT_READ for performing read

To perform read and write using restart condition:

- PSP_I2C_START | PSP_I2C_MASTER | PSP_I2C_WRITE (Do not use PSP_I2C_STOP as we are not intended to generate stop bit)
- In the next submit calls use PSP_I2C_RESTART instead of PSP_I2C_START.
- When stop condition needs to be generated to complete the transfer, use PSP_I2C_STOP along with the other necessary flag options.

To perform read/write operation in repeat mode:

- PSP_I2C_START | PSP_I2C_MASTER | PSP_I2C_WRITE | PSP_I2C_REPEAT (Do not use PSP_I2C_STOP as stop bit will be generated no sooner H/W detects repeat mode)

- In next submit calls use PSP_I2C_IGNORE_BUS_BUSY | PSP_I2C_REPEAT along with needed flag parameters (PSP_I2C_IGNORE_BUS_BUS is to be used as the bus is already in use).
- To generate stop condition, use PSP_I2C_IGNORE_BUS_BUSY | PSP_I2C_REPEAT | PSP_I2C_MASTER | PSP_I2C_STOP flag combinations.

Constraints
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *GIO_create*.

Example

```
GIO_Handle gioChan;
size_t size = 1;
DataParam buf;
Uint8 wBuffer[3] = {'0x03','0x02','0x01'}

/* channel creation and allocBuffer should be done here */

      buf.i2cTrans.buffer    = wBbuffer;
      buf.i2cTrans.bufLen    = 3u;
      buf.i2cTrans.flags     = PSP_I2C_DEFAULT_WRITE;
      buf.i2cTrans.param     = NULL;
      buf.i2cTrans.slaveAddr = 0X10;
      buf.timeout = SYS_FOREVER;

      status = GIO_submit (gioChan,IOM_WRITE,&buf, &size, NULL) ;
```

4.2.5 GIO_read/GIO_write

Syntax

```
status = GIO_read/GIO_write (      GIO_Handle gioChan,
                                   ptr bufP,
                                   Uns* Psize,
                            )
```

Parameters

gioChan

Handle of the I2c driver channel that was created with a call to *GIO_create*.

bufP

The bufP argument is a pointer to the structure defined in section 4.1.1, and this value should be used only when cmd is used as IOM_READ or IOM_WRITE. For other two commands it is NULL.

Psize

The Psize argument is a pointer to the argument of data type size_t, currently value passed should be 1.

Return Value
On success driver returns IOM_COMPLETED, on failure/error condition IOM error code

Description

- Similar to GIO_submit with cmd wither IOM_READ or IOM_WRITE with appCallBack value is NULL.

Constraints
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *GIO_create*.

Example

```
GIO_Handle gioChan;
size_t size = 1;
DataParam buf;
Uint8 wBuffer[3] = {‘0x03’,’0x02’,’0x01’}

/* channel creation and allocBuffer should be done here */

      buf.i2cTrans.buffer    = wBbuffer;
      buf.i2cTrans.bufLen    = 3u;
      buf.i2cTrans.flags     = PSP_I2C_DEFAULT_WRITE;
      buf.i2cTrans.param     = NULL;
      buf.i2cTrans.slaveAddr = 0X10;
      buf.timeout = SYS_FOREVER;

      Status = GIO_read (gioChan, &buf, &size);
```

4.2.6 GIO_flush/GIO_abort

Syntax
Status = GIO_flush/GIO_abort (GIO_Handle gioChan)

Parameters
gioChan
Handle of the I2C driver channel that was created with a call to *GIO_create*.

Return Value
On success driver returns IOM_COMPLETED, on failure/error condition IOM error code. In the current implementation of I2C driver this is not implemented so it returns IOM_ENOTIMPLEMENTED error.

Description
Similar to GIO_submit with cmd wither IOM_FLUSH or IOM_ABORT with appCallBack value is NULL.

Constraints
This function can only be called after the device has been loaded, initialized and created. The handle supplied as an argument to this function should have been obtained with a previous call to *GIO_create*.

Example

```
GIO_Handle gioChan;

/* channel creation and allocBuffer should be done here */

      status = GIO_flush (gioChan);
```

5 Example Applications

This section describes the example applications that are included in the package. These sample application can be run as is for quick demonstration, but the user will benefit most by using these samples as sample source code in developing new applications.

5.1 Writing Applications for I2C

This section provides guidance to user for writing their own application for I2C drivers.

5.1.1. File Inclusion

To write sample application user has to include following header files in the application:

1. **std.h**

   This file contains standard data types, macros and structures.

2. **gio.h**

   This file contains GIO layer macros and structures. These macros are wrapper macros to form a wrapper above GIO.

3. **tsk.h**

   This file contains all task module details.

4. **psp_i2c.h**

   This file contains i2c parameters which are passed to driver at the time of I2C driver registration with BIOS.

5.2 Sample Applications

5.2.1. Introduction

The sample application is a representative test program. Initialization of I2C driver is done by calling initialization function from BIOS.

5.2.2. Building the application

Please follow below steps to build sample application:

- Open CCS 3.3 setup. Import proper CCS configuration file. Set the proper CCS Gel file (Refer PSP_Release_Notes.pdf for details). Click on "Save & Quit" button and exit the setup.
- Open sample application as mentioned in *"<root>\pspdrivers\system\<soc>\bios\<evmNAME>\i2c\build\<i2c sample pjt>"*.
- For example for C6424, its *"<root>\pspdrivers\system\c6424\bios\evm6424\i2c\build\c6424_evm_i2c_st_sample.pjt"*.

- Compiler Switch "PSP_I2C_ASYNC_MODE_SUPPORT" is used to enable ASYNC MODE of operation of the sample application.

- Compile this project using Project->Build

- Note: Following Components needs to be linked for successful build and functionality of the application.
  - I2C
  - PAL_OS
  - SoC specific PAL_SYS

5.2.3. *Loading and running the application*

The sample application is loaded and executed via Code composer studio. It is good idea to reset the board before loading Code Composer. The application will print out the status messages and type of functionality the driver performs on the message log.

5.2.4. *Sample test for DM648/C6452*

❖ **Sample directory structure**

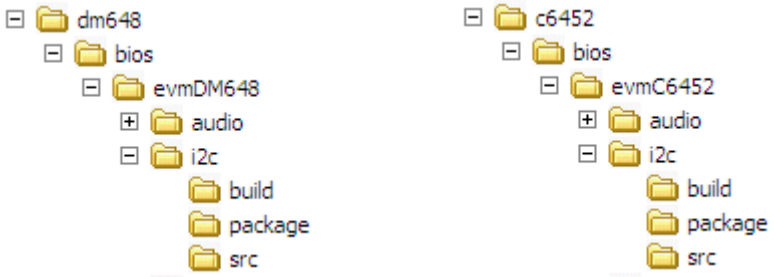

Figure 3. I2C sample directory structure for DM648 and C6452

Top level folder shown in the above figure contains header and tci files required specifically for sample application along with XDC packaging files(package.bld and package.xdc)

build: This subfolder in the sample folder contains project for sample appliaction.

src: This folder contains i2c sample application source files. It also contains header files related to i2c driver that are used by the sample application if any.

Package: This folder contains files generated by XDC tool

The sample application performs write operation to MSP430 to light the LED's connected to it.

This section describes how MSP430 to be configured.

❖ **I2C Configuration Parameters**

I2C_devParam used during I2C driver registration with BIOS using TCI files. I2C_devParam is configured as follows. Also refer section **"3.1.1 Initialization details"** for configuring TCI file.

```
PSP_I2cConfig I2C_devParams =   {
                        /** Driver operation mode                  */
                        PSP_OPMODE_POLLED,
```

```
    /**< Own address (7 or 10 bit)                */
    0x10,
    /**< Number of bits/byte to be sent/received    */
    8,
    /**< I2C Bus Frequency                         */
    200000,
    /**< Module input clock freq                   */
    11700000,
    /**< 7bit/10bit Addressing mode                */
    FALSE,
    /**< Digital Loop Back (DLB) mode enabled      */
    FALSE
};
```

❖ **I2C Data Param Configuration Parameters**

I2C data param structure (already explained at section 4.1.1). So following are the default configurations to be done for lighting LED's.

```
PSP_i2cDataParam    buf;
size_t              size;
Uint32              count   = 0;
Int                 retCode = 0;
Uint8               wBuffer[I2C_TVP_TRANSFER_SIZE];
Uint8               rBuffer[I2C_TVP_TRANSFER_SIZE];

wBuffer[0] = 0x07; /* offset for LED */
for (count = 1;count <I2C_TVP_TRANSFER_SIZE;count++)
{
   wBuffer[count] = 0x55;
   rBuffer[count] = 0x00;
}

     buf.i2cTrans.buffer    = wBuffer;
     buf.i2cTrans.bufLen    = 5;
     buf.i2cTrans.flags     = PSP_I2C_DEFAULT_WRITE;
     buf.i2cTrans.param     = NULL;
     buf.i2cTrans.slaveAddr = 0x70;
     buf.timeout            = 1000;
     size                   = 1u;
```

❖ **Driver naming convention used for Channel creation**

Application calls GIO_create to create and initialize a I2C driver channel.

Currently there are no channel specific parameters.

5.2.5. Sample test for DM6437/C6424

6. Sample directory structure

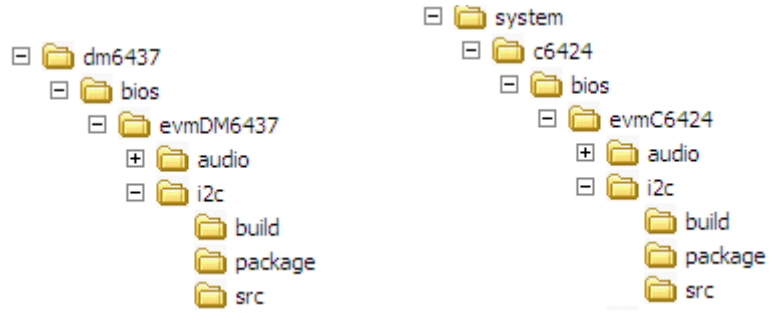

Figure 4. I2C sample directory structure for DM6437 and C6424

Top level folder shown in the above figure contains header and tci files required specifically for sample application along with XDC packaging files(package.bld and package.xdc)

build: This subfolder in the sample folder contains project for sample appliaction.

src: This folder contains i2c sample application source files. It also contains header files related to i2c driver that are used by the sample application if any.

Package: This folder contains files generated by XDC tool

The sample application performs write operation to LED slave to light the LED's connected to it.

This section describes how MSP430 to be configured.

❖ **I2C Configuration Parameters**

I2C_devParam used during I2C driver registration with BIOS using TCI files. I2C_devParam is configured as follows. Also refer section **"3.1.1 Initialization details"** for configuring TCI file.

```
PSP_I2cConfig I2C_devParams =  {
                        /** Driver operation mode                  */
                        PSP_OPMODE_POLLED,
                        /**< Own address (7 or 10 bit)              */
                        0x10,
                        /**< Number of bits/byte to be sent/received  */
                        8,
                        /**< I2C Bus Frequency                      */
                        200000,
                        /**< Module input clock freq                */
                        2400000,
                        /**< 7bit/10bit Addressing mode             */
                        FALSE,
                        /**< Digital Loop Back (DLB) mode enabled    */
                        FALSE
                    };
```

❖ I2C Data Param Configuration Parameters

I2C data param structure (already explained at section 4.1.1). So following are the default configurations to be done for lighting LED's.

```
PSP_i2cDataParam    buf;
size_t          size;
Uint32 count    =  0;
Int    retCode  =  0;
Int buffer;

for(count =0; count <20; count++)
{
   /*
    * Value to be written to LEDs
    */
    buffer              = 0x05;
    buf.i2cTrans.buffer    = (Uint8 *)&buffer;
    buf.i2cTrans.bufLen    = 1u;
    buf.i2cTrans.flags     = PSP_I2C_DEFAULT_WRITE | PSP_I2C_RESTART;
    buf.i2cTrans.param     = NULL;
    buf.i2cTrans.slaveAddr = 0x39u;
    buf.timeout     = -1;
    size          = (size_t)buf.i2cTrans.bufLen;
    retCode = GIO_write(i2cHandle,&buf,&size);

    TSK_sleep(500u);

   /*
    * Value to be written to toggle the LEDs
    */
    buffer              = 0x0A;
    buf.i2cTrans.buffer    = (Uint8 *)&buffer;
    buf.i2cTrans.bufLen    = 1u;
    buf.i2cTrans.flags     = PSP_I2C_DEFAULT_WRITE | PSP_I2C_RESTART;
    buf.i2cTrans.param     = NULL;
    buf.i2cTrans.slaveAddr = 0x39u;
    buf.timeout     = -1;
    size          = (size_t)buf.i2cTrans.bufLen;
    retCode = GIO_write(i2cHandle,&buf,&size);        size                = 1u;
 }
```

❖ Driver naming convention used for Channel creation

Application calls GIO_create to create and initialize a I2C driver channel.

Currently there are no channel specific parameters.