DM648/C6452 SPI Driver

USER'S GUIDE

Document Revision History

Rev No	Author(s)	Revision History	Date	Approval(s)
0.7	Chandan Nath	Updated for adding compiler switches in build options	May 20, 2008	Updating
0.6	Nagarjuna K	Updated IOCTL's	January 17, 2008	Updating
0.5	Nagarjuna L	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 mirror typo errors and updating 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

TABLE OF FIGURES

1 Introduction

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

DSP/BIOS applications use the driver typically through SPI APIs to perform read/write operations connected to the slaves. SPI 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.	sprue32_SPI.pdf	SPI Driver Documentation
2.	SPRU-404g.pdf	DSP/BIOS Driver Guide

1.3 S/W Support

This SPI 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 SPI driver is constituted of following sub components:

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

System components:

PALOS – DSP/BIOS Abstraction

Below Figure shows DM648/C6452 SPI driver architecture.

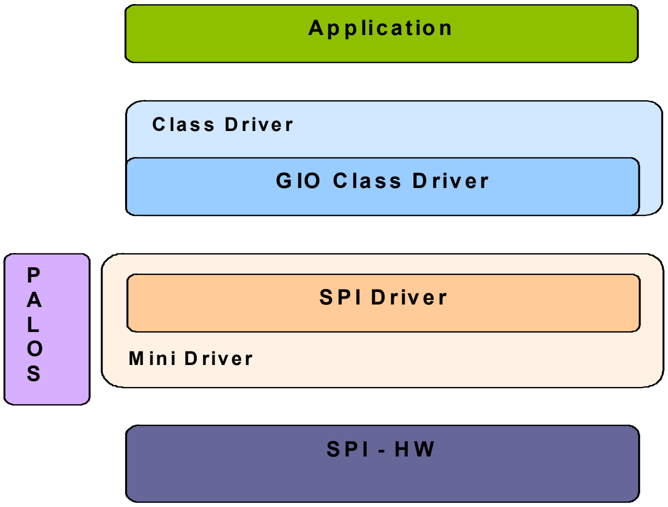


Figure 1. DM648/C6452 SPI driver architecture

1.5 Default Driver Configuration

1.1. Default Driver Configuration

By default the driver is configured as follows:

Lsb First (Data direction)	False
Delay	0
Parity	False
Character Length	8
Pin configuration	PSP_SPI_OPMODE_4PINCS
Operating Mode	Polled
Phase In	False
Wait Enable	False
SPI_BUS_FREQ	4000000
Mode	Master
Interrupt level	False

Note: Driver has separate configuration data for each of the SPI instances. This section represents for one instance only (driver contains separate configuration for each instance of the hardware).

1.6 Driver Capabilities

The DSP/BIOS SPI 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.

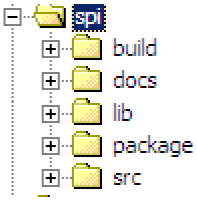
1.7 System Requirements

Details about the tools and the BIOS version that the driver is compatible with can be found in the system Release Notes.

2 Installation Guide

2.1 Component Folder

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



spi
build
docs
lib
package
src

Figure 2. SPI Driver Directory Structure

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

- **build:** This folder contains spi driver library project file. The generated driver library shall be included in the application where SPI 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 SPI 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 spi 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\lib"* folder.

2.3 Build Options

To compile driver for DM648/C6452, change build options as mentioned below:

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

Following compiler switches are used to compile for different options.

- ❑ **_DEBUG**
  This is used as a option 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.
- ❑ **SPI_INSTRUMENTATION_ENABLED**
  This option 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
- ❑ **PSP_SPI_EDMA_SUPPORT**
  This option is passed to the compiler for EDMA mode.

- **SPI_DEBUG_PRINTF**

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

3 DSP/BIOS SPI DRIVER Structures

This section discusses about the initialization details and initialization structures used in the spi driver. Please note that for some structure member information/details the DM648/C6452 SPI peripheral API reference document might need to be referred.

3.1.1 Initialization details

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

To have SPI device driver included in the application, corresponding TCI file have to be included in BIOS TCF i.e. *"dm648_spi.tci"* must be included in BIOS TCF file of the application for using SPI instance of the driver.

The following are the device configuration settings required to use the spi 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 DM648/C6452 SPI like module clock enabling and enabling pin-mux and sets init time parameters.
fxnTable - Function Table Pointer	*SPI_Fxns*. This is a global variable which points to the SPI driver APIs.
fxnTableType - Function Table Type	*IOM_Fxns*
deviceId - Device Id	Specify which SPI to use. For example to use SPI 0 this should be given as 0.
params – Pointer to Port parameter	A pointer to an object of type *SPI_devParams* as defined in the header file *psp_spi.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("SPI0");

bios.UDEV.instance("SPI0").fxnTableType = "IOM_Fxns";

bios.UDEV.instance("SPI0").initFxn = prog.extern("SPI_INIT");

bios.UDEV.instance("SPI0").params = prog.extern("SPI_devParams");

bios.UDEV.instance("SPI0").fxnTable = prog.extern("SPIMD_FXNS");
```

3.1.2 SPI_devParams

"psp_spi.h" file contains *PSP_SpiConfig_t* data structure that is passed while mdBindDev call which is defined with UDEV SPI 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. Note: The current implementation only supports Polled and Interrupt modes
moduleInputClkFreq	Input Frequency to SPI Module.
spiBusFreq	Output Data rate, in Kbps, of the SPI controller.
edmaHandle	Edma handle
loopBack	To Enable/Disable loop back mode
spiHWCfgData	To configure SPI HW parameters

3.1.2.1 PSP_spiHWConfigData

Following table describes the parameters contained in PSP_spiHWConfigData (spiHWCfgData) data structure available in "psp_spi.h" file:

Structure Members	Description
masterOrSlave	Select Master/Slave operation.(Master : slaveOperation = False)
pinOpModes	Pin configuration. SPI supports 3, 4 and 5pin modes.
intrLevel	Arm interrupts level 0 & 1.
clkInternal	External clock or internal clock.
powerDown	SPI in Active/Deactive state State
delay	SPI delay registers value
waitDelay	enable format delay between 2 consecutive transfers
csDefault	Default chip select pattern
configDatafmt	Data Format Configuration values

3.1.2.2 PSP_SpiConfigDataFmt

Following table describes the parameters contained in PSP_SpiConfigDataFmt (configDatafmt) data structure, which is a parameter spiHWCfgData and details of the same is available in "psp_spi.h" file:

Structure Members	Description
wDelay	Delay between data transfer.
char Length	Character length of data transfer.
lsbFirst	Data transfer direction.
clkHigh	Clock polarity.
phaseIn	Clock phase.
parityEnable	Parity enables.
oddParity	Odd and even parity selection.
waitEnable	Wait enable delay, master waits for ENA signal from slave.

3.1.2.3 PSP_spiType

Following table describes the parameters contained in PSP_spiType (masterOrSlave) enum, which is a parameter spiHWCfgData and details of the same is available in "psp_spi.h" file:

Structure Members	Description
PSP_SPI_TYPE_MASTER	Master mode
PSP_SPI_TYPE_SLAVE	Slave mode

3.1.2.4 PSP_spiPinOpMode

Following table describes the parameters contained in PSP_spiPinOpMode (pinModes) enum, which is a parameter spiHWCfgData and details of the same is available in "psp_spi.h" file:

Structure Members	Description
PSP_SPI_OPMODE_3PIN	3-Pin mode
PSP_SPI_OPMODE_SPISCS_4PIN	4-Pin Chip Select mode
PSP_SPI_OPMODE_SPIENA_4PIN	4-Pin Enable mode
PSP_SPI_OPMODE_5PIN	5-Pin mode

3.1.2.5 PSP_spiDelayParam

Following table describes the parameters contained in PSP_spiDelayParam (delay) data structure, which is a parameter spiHWCfgData and details of the same is available in "psp_spi.h" file:

Structure Members	Description
c2TDelay	Chip-select-active-to-transmit-start-delay
t2CDelay	Transmit-end-to-chip-select-inactive-delay
t2EDelay	Transmit-data-finished-to-ENA-pin-inactive-time-out
c2EDelay	Chip-select-active-to-ENA-signal-active-time-out

3.1.3 SPI Chan Params

These are no specific channel parameters that are passed while GIO_create call.

4 SPI API's

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

The following API functions are defined by the GIO module:

GIO_create	Allocate and initialize an SPI channel object
GIO_delete	De-allocate an SPI 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_spi.h** has the **DataParam** data structure that is passed to **spi_mdSubmitChan** function of the driver. The params are explained below:

Structure Members	Description
spiTrans	Spi 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 spiTrans in above data param

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

Structure Members	Description
*outBuffer	Data buffer pointer to where data needs to be transmitted
*inBuffer	Data buffer pointer to where data needs to be received
bufLen	Length of the data bytes to be receives or to transmit
chipSelectNo	Chip select number to be selected
dataFormatt	Data Format selection
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_SPI_IOCTL_CANCEL_PENDING_IO	To cancel pending IO requests if any in the transmission
PSP_SPI_IOCTL_SET_CS_POLARITY	To set or reset CS Polarity

4.1.4 Enum for dataParam in above PSP_SpiTransaction data structure

Following table describes the parameters contained in PSP_spiDataFormat (dataFormat) enum, which is a parameter PSP_spiTransaction and details of the same is available in “psp_spi.h” file:

Structure Members	Description
PSP_SPI_FMTSEL_0	Format selection 0
PSP_SPI_FMTSEL_1	Format selection 1
PSP_SPI_FMTSEL_2	Format selection 2
PSP_SPI_FMTSEL_3	Format selection 3

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 SPI drivers the string contains one token separated by '/'.

- SPI driver or port instance
  This identifies the SPI driver or port instance and this will be typically *"SPI0", "SPI1"* and so on, where suffix to SPI 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 SPI driver should be created in IOM_INOUT mode as spi is a transceive device.

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 SPI 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 a spi 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 SPI

GIO_Handle chanHandle;
GIO_Attrs gioAttrs = GIO_ATTRS;

chanHandle = GIO_create ("/SPI0", IOM_INOUT,
                         NULL, NULL, &gioAttrs);
if (NULL == chanHandle)
{
      printf (" Failed create SPI0 channel \r\n");
      return;
}
```

4.2.2 GIO_delete

Syntax

int GIO_delete (GIO_Handle gioChan);

Parameters

Handle of the spi 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 SPI

GIO_Handle chanHandle;
GIO_Attrs gioAttrs = GIO_ATTRS;
Int status;
/* Create SPI handle */

Status = GIO_delete (chanHandle);

if (IOM_COMPLETED != status)
{
      printf(" Failed deleting channel \r\n");
      return;
}
```

4.2.3 GIO_control

Syntax

status = GIO_control (gioChan, cmd, args);

Parameters

gioChan

Handle of the SPI 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 SPI driver are available in section 4.1.3

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;
/* create SPI handle */
status = GIO_control(chanHandle, PSP_SPI_IOCTL_CANCEL_PENDING_IO, NULL);

if (IOM_COMPLETED != status)
{
	printf(" Failed to cancel IO \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 SPI 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 */

Note:- Since SPI driver is a transceiver device, the commands IOM_READ/IOMWRITE perform only transceiver operation.

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 SPI 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 SPI driver are:

- PSP_SPI_CSHOLD is used to set CSHOLD.
- PSP_SPI_CSH_ENABLE is used to allow the driver to hold the CS pin even after submit calls ends successfully. If there is an error during transceive operation even if the flag is set CSHOLD will be disabled.
- PSP_SPI_CSH_ENABLE option with out PSP_SPI_CSHOLD options is not a valid case and results an error condition.

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'}
Uint8 rBuffer[3];

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

        buf.spiTrans.outBuffer  = wBbuffer;
        buf.spiTrans.inBuffer  = rBuffer;
        buf.spiTrans.bufLen   = 3u;
        buf.spiTrans.flags      = PSP_SPI_CSHOLD;
        buf.spiTrans.param     = NULL;
        buf.spiTrans.chipSelectNo = 2;
        buf.spiTrans.dataFormat = PSP_SPI_FMTSEL_0;
        buf.timeout = 1000;

        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 Spi 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.spiTrans.outBuffer  = wBbuffer;
        buf.spiTrans.inBuffer  = rBuffer;
        buf.spiTrans.bufLen   = 3u;
        buf.spiTrans.flags      = PSP_SPI_CSHOLD;
        buf.spiTrans.param    = NULL;
        buf.spiTrans.chipSelectNo = 2;
        buf.spiTrans.dataFormat = PSP_SPI_FMTSEL_0;
        buf.timeout = 1000;

        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 SPI 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 SPI 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 SPI

This section provides guidance to user for writing their own application for SPI 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_spi.h**

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

5.2 Sample Applications

5.2.1. Introduction

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

The sample application performs read/write operation to EEPROM connected to it.

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 DM648/C6452_BIOS_PSP_Release_Notes.doc for details). Click on “Save & Quit” button and exit the setup.
- Open sample application as mentioned in *“<root>\pspdrivers\system\<soc>\bios\<evmNAME>\spi\build\<spi sample pjt>”*.

- For example for C6424, its *"<root>\pspdrivers\system\c6424\bios\evm6424\spi\build\c6424_evm_spi_st_sample.pjt"*.
- Compile this project using Project->Build
- Note: Following Components needs to be linked for successful build and functionality of the application.
  - SPI
  - 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. *Configuration Parameters*

This section describes how MSP430 to be configured. Refer section **"3.1.1 Initialization details"** for configuring TCI file.

- **SPI Configuration Parameters**

SPI_devParam used during SPI driver registration with BIOS using TCI files. SPI_devParam is configured as follows.

```
SPI_devParams.mode                  = PSP_OPMODE_POLLED;
SPI_devParams.moduleInputClkFreq = PSP_SPI_MODULE_CLK_FREQ;
SPI_devParams.outputClkFreq         = SPI_BUS_FREQ;
SPI_devParams.loopBack              = FALSE;
SPI_devParams.edmaHandle            = NULL;

SPI_devParams.spiHWCfgData          = spiHWData;

spiHWData.clkInternal             = TRUE;
spiHWData.delay.c2EDelay          = 0;
spiHWData.delay.c2TDelay          = 0;
spiHWData.delay.t2CDelay          = 0 ;
spiHWData.delay.t2EDelay          = 0;
spiHWData.enableHighZ             = FALSE;
spiHWData.intrLevel               = FALSE;
spiHWData.pinOpModes              = PSP_SPI_OPMODE_SPISCS_4PIN;
spiHWData.powerDown               = FALSE;
spiHWData.masterOrSlave           = PSP_SPI_TYPE_MASTER;
spiHWData.waitDelay               = FALSE;
spiHWData.csDefault               = CS_DEFAULT_SPI;

for(i = 0 ; i < 4 ; i++)
{
        spiHWData.configDatafmt[i].charLength    = SPI_DATA_CHAR_LENGTH;
        spiHWData.configDatafmt[i].clkHigh       = FALSE;
        spiHWData.configDatafmt[i].lsbFirst      = FALSE;
        spiHWData.configDatafmt[i].oddParity     = FALSE;
        spiHWData.configDatafmt[i].parityEnable  = FALSE;
        spiHWData.configDatafmt[i].phaseIn       = TRUE;
        spiHWData.configDatafmt[i].waitEnable    = FALSE;
        spiHWData.configDatafmt[i].wDelay        = 0;
    }

    spiHWData.configDatafmt[1].charLength   = 16;
```

❖ SPI Data Param Configuration Parameters

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

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

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

    buf.spiTrans.outBuffer  = wBbuffer;
    buf.spiTrans.inBuffer  = rBuffer;
    buf.spiTrans.bufLen   = 3u;
    buf.spiTrans.flags      = PSP_SPI_CSHOLD;
    buf.spiTrans.param     = NULL;
    buf.spiTrans.chipSelectNo = 2;
    buf.spiTrans.dataFormat = PSP_SPI_FMTSEL_0;
    buf.timeout = 1000;
      size                  = 1u;
```

❖ Driver for Channel creation

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

Currently there are no channel specific parameters.