Freescale Semiconductor
Application Note

Document Number: AN5134
Rev. 0, May 2015

Introduction to the Zipwire Interface

Inter-Processor Communication with SIPI/LFAST on the MPC57xx and S32Vxxx families

by: Randy Dees, Hugo Osornio, Steven Becerra, and Ray Marshall

Contents

1 Introduction

The MPC57xx family is the first family of devices that include a new bus for communicating between two devices over a high speed (320 Mb/s) serial interface called Zipwire. It is implemented using a Serial Inter-Processor Interface (SIPI) over an LVDS[1] Fast Asynchronous Serial Transmission Interface (LFAST). The SIPI module controls the higher level protocol of the interface, and the LFAST controls the physical interface.

NOTE

Some devices support two separate interfaces that are similarly based on the LFAST and SIPI modules. The first, discussed in this application note, is the Zipwire interface that consists of the SIPI and LFAST modules and is used for interprocessor communication. Some devices implement a second separate interface that can be used as a high-speed debug interface. This debug Zipwire interface consists of the LFAST, a reduced SIPI module that has a very limited functionality, and a JTAG Master interface module (JTAGM) that allows access of the JTAG debug interface through the LFAST interface.

1. LVDS is Low Voltage Differential Signalling

© 2015 Freescale Semiconductor, Inc.

Zipwire uses a low-speed reference clock that is shared between the clients and uses a single pair of LVDS signals for data transmission and a second pair for reception. The normal communication mode for the Zipwire is 320 Mb/s, however, it starts up at a lower speed until a basic connection is established between the devices.

On the current Freescale microcontrollers (MCU) that support the Zipwire interface, if two MCUs are connected, either MCU can be defined as the master and the other defined as the slave device. This may also depend on the clock generation requirements for the devices.

- The master device is defined as the device that "owns" the link. It acts as the initiator for all link management commands, such as changing the interface speed, sending ping commands, and recovering from errors. The master device receives the reference clock from the slave device.
- The slave device generates the reference clock and provides it to the master. In addition, the slave device responds to all link management commands from the master device.

The following table shows the Freescale devices that implement the Zipwire interface for interprocessor communication. It includes both Power Architecture® based devices and devices based on ARM® cores.

Table 1. Devices that support the Zipwire interprocessor communication interface

Device	Core type
MPC574xP	Power Architecture
MPC574xR	Power Architecture
MPC577xK	Power Architecture
MPC577xC	Power Architecture
MPC5777M	Power Architecture
S32V23x	ARM

The Zipwire interface is compatible with devices available from ST Microelectronics and the High Speed Serial Interface (HSSI) from Infineon Technologies AG.

The Zipwire interface could also be used to connect an MCU with an external smart peripheral.

This application note provides an overview of the Zipwire Interprocessor communication interface, including the hardware interface, the recommended software API[2], and an example of a typical use to transfer information between two devices.

2 Zipwire Interface overview

Some members of the MPC57xx family implement a Zipwire Interface as an Interprocessor Communication interface. Zipwire is a fully operational and layered protocol to exchange data between two devices. Features of the Zipwire interface include:

- Point to point communication
- Simple high speed, full duplex, flexible interface
- Low pin count (5)
- Timeout protection
- Fixed priority
- Cyclic redundancy check for data integrity
- Pipe-lined, multi-channel architecture for overlapping requests (up to two outstanding requests by the initiator)
- Streaming mode
- Multiple loopback modes to check the physical interface
- Automatic ping response generation when in slave mode
- Detects unsupported channel numbers and unsupported payload sizes

2. Application Programming Interface

The actual physical layer is implemented with the LFAST physical communication interface. As implied in the name of the interface, the LFAST physical layer is an asynchronous fast serial interface. The protocol is based on a frame format that includes synchronization information at the beginning of the frame. Data within the frame is synchronous.

The application layer of Zipwire is implemented in the SIPI. The application layer runs on top of the LFAST physical communication interface and has its own protocol. The main purpose of the SIPI is to provide the framework to exchange information and provides the link between memory or processes on one MCU through the LFAST physical communication interface to another MCU or a smart peripheral device. SIPI also adds error detection features such as CRC, acknowledge, and timeout. (The LFAST protocol does not include any error detection/correction scheme by itself). The SIPI layer supports multiple channels and some transfers can be overlapped.

In summary, SIPI and LFAST offer the means to exchange information between processors at rates up to 320 Mb/s. The following figure shows an overview of a typical Zipwire implementation showing the memory interface through the SIPI and LFAST modules to the physical drivers and receivers.

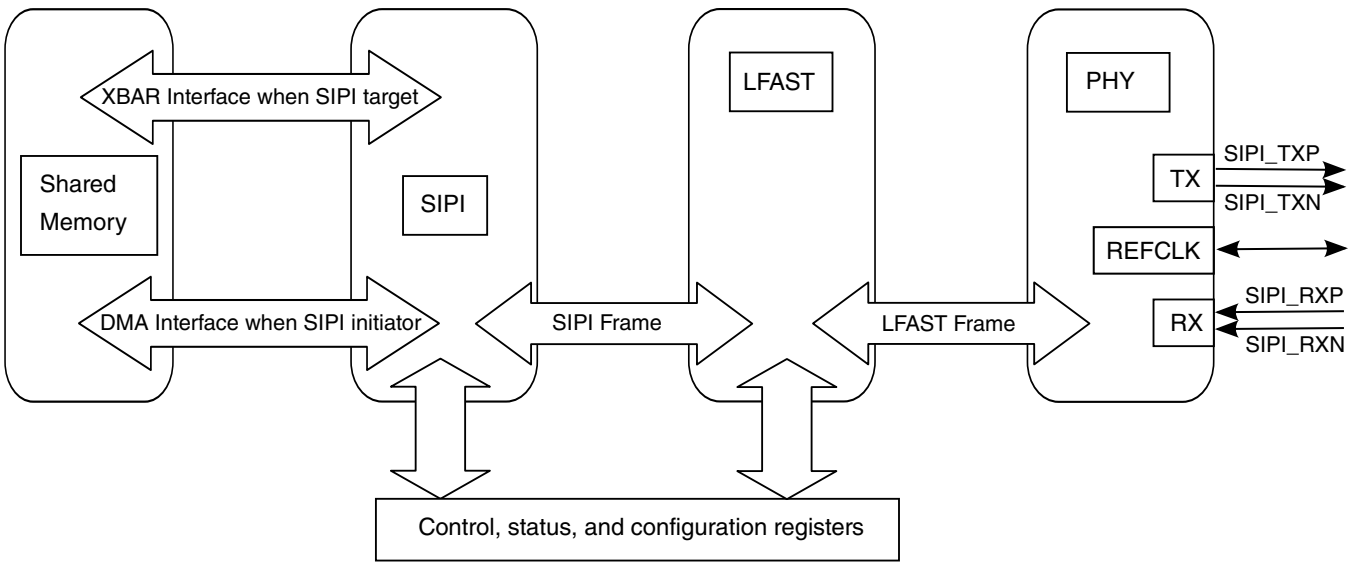

Figure 1. Zipwire overview

The figure below shows the frame format of the LFAST encapsulated SIPI frames/messages. A frame starts with 16 bits for synchronization (0b1010_1000_0100_1011 [0xA84B]) followed by the LFAST header. The payload of the LFAST frame includes the SIPI header and the actual payload (the contents depend on the payload type), followed by a CRC of the SIPI information. The frame ends with a single '1' stop bit. The synchronization pattern allows the receiver to adjust when the bits will be sampled within the remaining bits in the frame for optimum performance. The receiver uses a multiphase clock to determine which phase of its internal clock to use based on decoding the LFAST sync pattern.

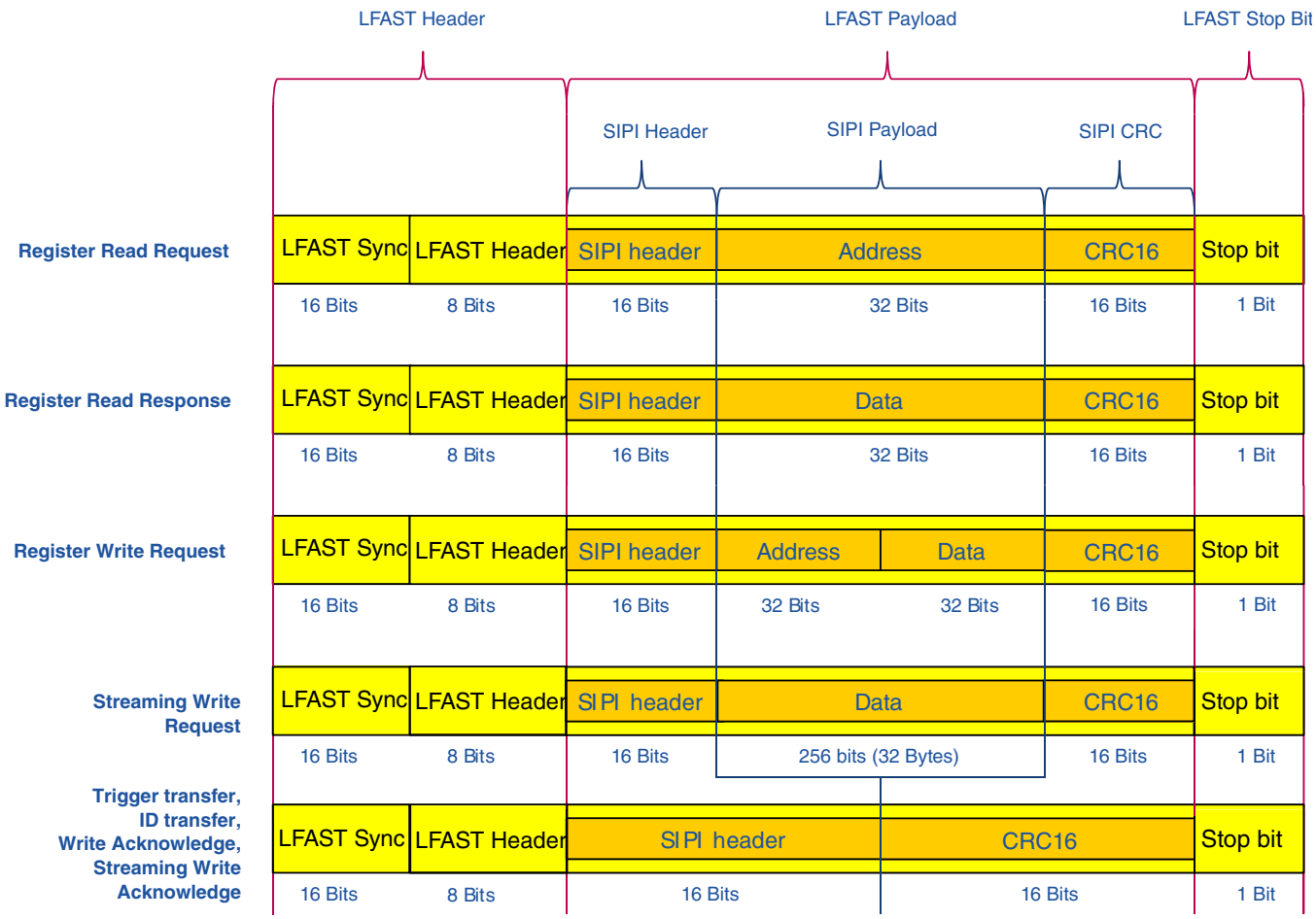

Figure 2. Zipwire message formats

See the Zipwire, SIPI, and LFAST chapters of the device reference manual for more information about the specific values of the LFAST Header, SIPI header, and explanation of the different types of messages.

2.1 Zipwire SIPI LFAST software model

Software should interact with the Zipwire hardware through the standard API. The Zipwire API conforms to the standard Open Systems Interconnection (OSI) model. The following table shows the mapping of the software and hardware to the OSI model. The Zipwire LFAST module handles the basic media layer (packet, frame, and bit formatting), with the SIPI module handling most of the Host layers, except for the network process to application layer (transport, session, and presentation layers). Using the Zipwire API allows the user software to interface in a standard manner to the lower levels of the model.

Table 2. Zipwire SIPI LFAST software model

OSI Model				Protocol Name	
Layer	Data Unit	Layer	Function		
Host Layers	Data	7. Application	Network process to application	SIPI Software	Zipwire
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data	SIPI Hardware	

Table continues on the next page...

Table 2. Zipwire SIPI LFAST software model (continued)

OSI Model				Protocol Name	
Layer	Data Unit	Layer	Function		
		5. Session	Inter-host communication, managing sessions between applications		
	Segment	4. Transport	End-to-end connections, reliability and flow control		
Media Layers	Packet	3. Network	Path determination and logical addressing	LFAST	
	Frame	2. Data Link	Physical addressing		
	Bit	1. Physical	Media, signal and binary transmission		

2.2 Typical Zipwire example overview

The simplest example of a Zipwire interface is two MPC5777M microcontrollers communicating to each other (See the figure below). Device 1 wants to write a memory location of Device 2.

1. The SIPI module, acting as the initiator, of the first MCU will use the DMA to acquire data to be sent from memory. This is done via software initialization.
2. Once the data registers are full, the initiator SIPI will automatically construct a frame indicating the operation to perform, the address to be written, the data to write, and a CRC. This is handled by the SIPI module in hardware.
3. This SIPI frame will then be sent by the hardware o the LFAST module and the module will embed the SIPI frame within a greater LFAST compliant frame that will be sent through the physical interface.
4. The LFAST frame will be received by the second MCU, acting as a target node. The LFAST will then decompose the frame back into a SIPI understandable frame and send it to the SIPI. This is handled in hardware.
5. The SIPI target will verify that the message is correct and use the XBAR master interface to modify the data at the requested address. Once completed, the SIPI target will send back a write acknowledge message to the initiator. This is all handled by the Zipwire hardware.
6. Once the acknowledge has been received by the initiator, the DMA on the first device can start another operation until the desired amount of data has been written to the target MCU.

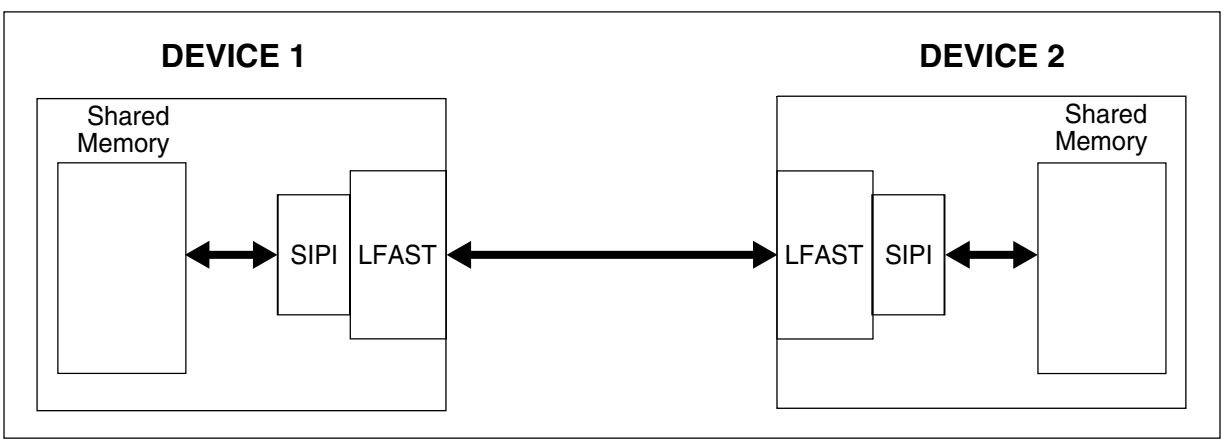

Figure 3. Typical Zipwire application

3 Zipwire examples

This section describes some basic examples of using the Zipwire driver. Simple functions were written to show the basic operations of the Zipwire interface, including determining the remote device connected and performing simple operations to the remote device over the Zipwire link.

Each of the examples is set up as a separate example and each contains both the Master and the Slave software. Each includes calls to the Zipwire API routines. The Zipwire API is included in this document as an appendix.

3.1 Function file locations

This application note includes a zip file that contains all of the code discussed in this application note. The software was compiled with the Green Hills Software Multi compiler, version 2014.1.2, but later versions should also work.

The software project included in this application note is divided into two sections. There is the main example code and there is a separate subdirectory of the code required for the API interface. The following table lists all functions in this example application note and the file name in which they are implemented. There are three functions that must be modified with MCU specific information.

Table 3. Example function file locations

File Name	Description	Function
crt0.s	Device low level 'C' code initialization	—[1]
LFAST.c	Device specific initialization	LFAST_Configure(unsigned char master)[1]
main.c	Main example code	user_testcase(void)
		main(void)
mcu_init_flash.c	MCU specific initialization (in 'C')	MC_MODE_INIT(void)[1]
SIPI.c	Example functions used in the example code that call the Zipwire API functions	unsigned char sipi_app_note_read(void)
		unsigned char sipi_app_note_write(void)
		unsigned char sipi_app_note_multiple_read_no_dma(void)
		unsigned char sipi_app_note_multiple_write_no_dma(void)
		unsigned char sipi_app_note_multiple_write_dma(void)
		unsigned char sipi_app_note_stream_write(void)
		unsigned char sipi_app_note_ID(void)
		unsigned char sipi_app_note_event(void)

1. Values used in this function are device specific

In addition to the example code, there are additional header files that are included in this example project. These are shown in the following table.

Table 4. Zipwire example header files

File Name	Description
SIPI_HSSL_Header_v4.h	This header file provides all of the prototype functions for the Zipwire driver. This file can be included in the target software to include the Zipwire API functions.
SIPI_API.h	This header file contains all of the necessary definitions for the Zipwire driver internal use.

The files and functions for the Zipwire API itself, are included in the appendix of this application note. The file SIPI_HSSL_Header_v4.h, along with the Zipwire API object files should be included in the project to include the Zipwire API routines.

3.2 Zipwire demo overview

The Zipwire demo program is an example of using the Zipwire driver and is contained in the file main.c. This program initializes the MCU and then calls the example functions. The flow chart of the demo is shown in Figure 4.

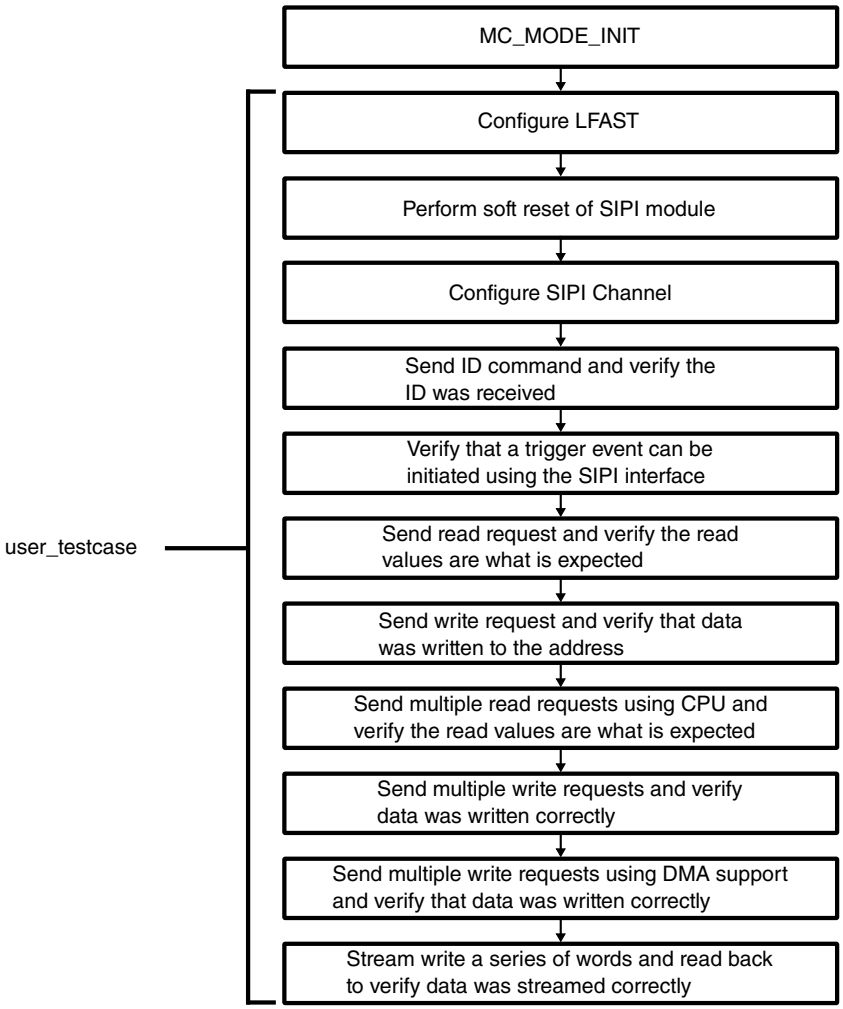

Figure 4. Zipwire Demo Code Flow Chart

The demo consists of several functions that are called from the demo. These functions are listed in Table 5.

Table 5. Demo function overview

Function	Description
sipi_app_note_read	Exercises the SIPI_read() and SIPI_read_channel_data functions to send a read request and verifies that the read values are the expected values.
sipi_app_note_write	Exercises the SIPI_write(), SIPI_read() and SIPI_read_channel_data functions to send a write request and then verify that the address was actually written.
sipi_app_note_ID	Exercises the SIPI_ID() and SIPI_read_channel_data functions to send the ID command and read the CADR to verify that the ID was received.
sipi_app_note_event	Exercises SIPI_Trigger() to verify that a trigger event can be initiated using the SIPI interface.
sipi_app_note_multiple_read_no_dma	Exercises SIPI_multiple_read() without using DMA support, each array will be read using CPU and the read contents will be verified.

Table continues on the next page...

Table 5. Demo function overview (continued)

Function	Description
sipi_app_note_multiple_write_no_dma	Exercises SIPI_multiple_write() and SIPI_multiple_read() to send multiple write requests, once completed will read the written content to verify that the data was written correctly.
sipi_app_note_multiple_write_dma	Exercises SIPI_multiple_write() using DMA support and SIPI_multiple_read() to send multiple write requests, once completed will read the written content to verify that the data was written correctly.
sipi_app_note_stream_write	Exercises SIPI_stream_transfer() and SIPI_multiple_read() to stream write a series of words and then read them back to verify the data.

3.2.1 Function index

The following table shows the functions used in the SIPI application note demo.

Table 6. Quick function reference

Type	Name	Arguments
void	LFAST_Configure	unsigned char master
void	MC_MODE_INIT	void
unsigned char	sipi_app_note_ID	void
unsigned char	sipi_app_note_event	void
unsigned char	sipi_app_note_multiple_read_no_dma	void
unsigned char	sipi_app_note_multiple_write_dma	void
unsigned char	sipi_app_note_multiple_write_no_dma	void
unsigned char	sipi_app_note_read	void
unsigned char	sipi_app_note_stream_write	void
unsigned char	sipi_app_note_write	void
void	user_testcase	void

3.2.2 Function MC_MODE_INIT

This function initializes the MPC5777M processor. It sets up the phase lock loop to set the device operating frequency, the peripheral clock frequencies, and enables all of the peripheral modules in the different "RUN" modes. It then performs a mode change to enable the clocks and enable all cores of the device.

The function must be tailored for the target system and environment, including the device crystal frequency.

NOTE

The Zipwire clock is set up in the file LFAST.c.

Prototype: `void MC_MODE_INIT(void);`

3.2.3 Function user_testcase

The user testcase is the actual example code and calls most of the other functions of the Zipwire example code. The flow of this section is shown in Figure 4.

Prototype: `void user_testcase(void);`

3.2.4 Function LFAST_Configure

Configures the LFAST as either a Master or a Slave, sets the clocks and bus speeds, and configures the pins to set the LFAST link.

Prototype: `void LFAST_Configure(unsigned char master);`

Table 7. LFAST_Configure Arguments

Type	Name	Direction	Description
unsigned char	master	**input**	defines whether to configure the LFAST node as master or slave.

3.2.5 Function sipi_app_note_ID

Exercises the SIPI_ID() and SIPI_read_channel_data functions to send the ID command and read the CDR to verify that the ID was received.

Prototype: `unsigned char sipi_app_note_ID(void);`

Return:
- 0 = Successfully Set Up
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error
- 10 = ID Not received properly

3.2.6 Function sipi_app_note_read

Exercises the SIPI_read() and SIPI_read_channel_data() functions to send a read request and verifies that the read values are the expected values.

Prototype: `unsigned char sipi_app_note_read(void);`

Return:
- 0 = Successful
- 1 = Invalid Width
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error
- 10 = Wrong read value

3.2.7 Function sipi_app_note_write

Exercises the SIPI_write(), SIPI_read() and SIPI_read_channel_data functions to send a write request and then verify that the address was actually written.

Prototype: `unsigned char sipi_app_note_write(void);`

Return:
- 0 = Successfully Set Up
- 1 = Invalid Data Size
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error / Wrong Acknowledge
- 10 = Wrong Read Value

3.2.8 Function sipi_app_note_multiple_read_no_dma

Exercises SIPI_multiple_read() without using DMA support. Each array will be read using CPU and the read contents will be verified.

Prototype: `unsigned char sipi_app_note_multiple_read_no_dma(void);`

Return:
- 0 = Successfully Set Up
- 1 = Incorrect Channel
- 2 = Channel Busy
- 4 = Timeout Error
- 10 = Data not read properly

3.2.9 Function sipi_app_note_multiple_write_no_dma

Exercises SIPI_multiple_write() and SIPI_multiple_read() to send multiple write requests, once completed will read the written content to verify that the data was written correctly.

Prototype: `unsigned char sipi_app_note_multiple_write_no_dma(void);`

Return:
- 0 = Successfully Set Up
- 1 = Incorrect Channel
- 2 = Channel Busy
- 4 = Timeout Error
- 10 = Data not read properly

3.2.10 Function sipi_app_note_multiple_write_dma

Exercises SIPI_multiple_write() using DMA support and SIPI_multiple_read() to send multiple write requests. Once completed, will read the written content to verify that the data was written correctly.

Prototype: `unsigned char sipi_app_note_multiple_write_dma(void);`

Return:
- 0 = Successfully Set Up
- 1 = Incorrect Channel

- 2 = Channel Busy
- 4 = Timeout Error
- 10 = Data not read properly

3.2.11 Function sipi_app_note_stream_write

Exercises SIPI_stream_transfer() and SIPI_multiple_read() to stream write a series of words and then read them back to verify the data.

Prototype: `unsigned char sipi_app_note_stream_write(void);`

Return:
- 0 = Successfully Set Up
- 1 = Incorrect Channel
- 2 = Channel Busy
- 4 = Timeout Error
- 10 = Data not read properly

3.2.12 Function sipi_app_note_event

Exercises SIPI_Trigger() to verify that a trigger event can be initiated using the SIPI interface. Target device waits until the trigger event is received.

Prototype: `unsigned char sipi_app_note_event(void);`

Return:
- 0 = Successfully Set Up
- 1 = Incorrect Channel
- 2 = Channel Busy

3.3 LFAST clock settings

The LFAST interface requires a reference clock and uses an internal Phase Lock Loop (PLL) to derive the timing used for transmitting and receiving the LFAST signals. The clock is generated by the Zipwire slave device, and the Zipwire master device uses the external clock as the reference for its PLL. The possible frequencies are shown in the table below.

Table 8. Zipwire clocking options

Reference clock frequency	Low speed operating frequency	High speed operating frequency	LFAST PLL multiplier
10 MHz	5 MHz	320 MHz	32
20 MHz	5 MHz	320 MHz	16

Software must enable the clock signal in low speed mode initially. Once a link is established at low speed, the Zipwire interface can be put into high speed mode.

In the current MCUs, there are three sources for the clock used by the Zipwire interface, the MCU oscillator, the output of the system phase-lock loop (PLL), or the external LFAST reference clock that is provided by the slave Zipewire device. The clock frequency is typically 20 MHz. The Zipwire interface contains a dedicated PLL[3] that uses the reference clock to generate the 320 MHz required for operation of the LFST and SIPI modules.

3.4 Zipwire pins

The Zipwire interface consists of five signals: a pair of LVDS transmit pins, a pair of receive LVDS pins, and a clock. The clock is unidirectional and is defined to be an output on the Slave and an input on the Master node. The following table shows the Zipwire pins.

NOTE

The signal names are named slightly different between the different devices. Both names indicate the Zipwire signal.

Table 9. Zipwire Signals

Zipwire signal	Full Name	Direction	Description
SIPI_TXN/ LFAST_TXN	SIPI Transmit LVDS Negative terminal	Output	The negative signal of the Zipwire transmit interface.
SIPI_TXP/ LFAST_TXP	SIPI Transmit LVDS Positive terminal	Output	The positive signal of the Zipwire transmit interface.
SIPI_RXN/ LFAST_RXN	SIPI Receive LVDS Negative terminal	Input	The negative signal of the Zipwire receive interface.
SIPI_RXP/ LFAST_RXP	SIPI Receive LVDS Positive terminal	Input	The positive signal of the Zipwire receive interface.
LFAST_REFCLK/REF_CLK	LFAST interface system clock	Input/Output	The input or output reference clock for the Zipwire interface.

Different devices instantiate the Zipwire signals on different pins of the device. The following table shows the pins used for the Zipwire interface.

Table 10. Zipwire pins

Zipwire pin	MPC574xP[1]		MPC577xC		MPC577xK		MPC5777M		S32V23x	
	Port Pin	MSCR	Port Pin	MSCR	Port Pin	MSCR	Port Pin	MSCR	Port Pin	MSCR
SIPI_TXN	PI[5]	133 (SSS=3)	PLLCFG0	208 (SSS=4)	—[2]	—	PD[6]	54 (SSS=1)	—[3]	—
SIPI_TXP	PC[12]	44 (SSS=3)	PLLCFG1	209 (SSS=4)	—[2]	—	PA[14] (SSS=1)	14 (SSS=1)	—[3]	—
SIPI_RXN	PI[6]	134 (SSS=3)	WKPCFG	213 (SSS=4)	—[2]	—	PF[13]	93	—[4]	—
SIPI_RXP	PG[7]	103 (SSS=3)	BOOTCFG0	211 (SSS=4)	—[2]	—	PD[7]	55	—[4]	—
LFAST_REFCLK	PI[7]	135 (SSS=1)	PLLCFG2	210 (SSS=4)	135	PI[7] (SSS=1)	PF[14]	94 (SSS=1)	PC8 (MODE_MUX=2)	40/550

1. The Zipwire interface is only available in the 257 MAPBGA package.
2. The transmit and receive pins on this device are dedicated LVDS functions and require no configuration.
3. Function selected by SRC_SOC_GPR3[4]
4. This is a dedicated LVDS signal for the LFAST interface.

3. In devices that include both an Interprocessor communication Zipwire interface and a debug Zipwire interface, the PLL may be shared between both interfaces.

Table 11 describes the values to be programmed into the Multiplexed Signal Configuration register to setup up the Zipwire pins for the MPC5777M. Other devices in Table 10 may require a different initialization.

Table 11. MPC5777M Multiplexed Signal Configuration register (MSCR) values

Zipwire signal	Register Setting	Direction	Bit settings	Description
SIPI_TXN	0x0500_0000	Output	ODC = 0b101	Output Drive Control is LFAST LVDS
SIPI_TXP	0x0500_0000	Output	ODC = 0b101	Output Drive Control is LFAST LVDS
SIPI_RXN	0x0020_0001	Output	ILS = 0b10	Input Level Selection is LVDS
			SSS = 0x01	Source Signal Source is SIPI_RXN
SIPI_RXP	0x0020_0001	Input	ILS = 0b10	Input Level Selection is LVDS
			SSS = 0x01	Source Signal Source is SIPI_RXP
LFAST_REFCLK (Master)	0x0008_0001	Input	WPUE = 1	Weak pull up is enabled
			SSS = 0x1	Source Signal Source is LFAST_REFCLK
LFAST_REFCLK (Slave)	0x2280_0000	Output	OERC = 0b010	Output Edge Rate is Strong Drive (50 ohm/5 ns)
			ODC = 0b010	Output drive control is push-pull
			SMC = 0b1	Pin is not disabled in Safe Mode Control
			SSS = 0x1	Source Signal Source is LFAST_REFCLK

3.5 Example Configuration

The Zipwire.h file is used to define if the software should configure the Zipwire node as a master or slave interface. CONFIGURED_AS_MASTER must be set to select either master or slave mode depending on the compile requirements of the software. To use this example, the example software must be compiled both as a master and as a slave and programed into two separate MCUs. These MCUs should have the Zipwire interfaces connected to each other, connecting the master's transmit pins (N and P) to the Slave's receive pins (N and P). The Slave's transmit pins (N and P) must be connected to the master's receive pins (N and P). The clock pin of the master and slave must also be connected to each other.

NOTE

Depending on the board layout and requriements, a termination resistor may be needed on the REFCLK signal. See Zipwire hardware and layout.

This example is written to support different implementations of the Zipwire interface, although all current implementations have the same number of channels. In addition, all current MCU implementations support being set as a master or a slave.

3.5.1 Define CONFIGURED_AS_MASTER

Definition:`#define CONFIGURED_AS_MASTER 0`

Possible values::

- 0 = Slave
- 1 = Master

4 LFAST configuration

The LFAST module allows for many options to be programmed into its configuration registers. However, most of these options should not be used by customers. This section describes the recommended configuration that should be used for the LFAST Master interface that is implemented in this example, specifically, in the LFAST_Configure (LFAST.c).

Table 12. LFAST Master mode configuration

Step	Description	RM Operation	Registers[1]	Bits
0	Initialize device pins for the Zipwire interface	Set up the Positive and Negative Transmit pins in the System Integration unit Lite 2 (SIUL2) Multiplexed Signal Control Registers (MSCR)	See Table 11	
		Set up the Positive and Negative Receive pins in the SIUL2 MSCR		
1	Set the LFAST wakeup delay and rate change delay for the Line Driver (LD).	After reset the SLCR and RCDCR are programmed according to the LVDS requirements of the device.	LFAST Rate Change Delay Control Register (RCDCR)	Data Rate Controller Count Value (DRCNT) = 0xF
			LFAST Wakeup Delay Control register (SLCR)	High Speed Sleep Mode Exit Time (HSCNT) = 0x12 (18 cycles)
				Low Speed Sleep Mode Exit Time (LSCNT) = 0x1
				Wake Up time for the LD (HWKCNT) = 0x54
				Wake Up time for the LD (LWKCNT) = 0x2
2	Set the LFAST operating speeds	The PLLCR is programmed with configuration parameters for the PLL	LFAST PLL Control Register (PLLCR)	PLL Loop Optimization (LPCFG) = 3 (2x IBASE current
				Division Factor for the PLL Reference Clock (PREDIV) = 0 (Direct Clock)
				Feedback Division factor for PLL Reference Clock (FBDIV) = 15
				Test mode programmability (IPTMODE) = 0b000 Normal functional mode. Test modes should not be used by customers.
				SW signal to turn off the PLL (SWPOFF) = 0b0. The PLL should not be disabled for normal operation.
				Software signal to turn on the PLL (SWPON) =

Table continues on the next page...

Table 12. LFAST Master mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
				0b0 = Do not turn on the PLL.
				Invert reference clock edge to the PFD (REVINV) = 0b0 = The reference clock should not be inverted.
				PLL Lock Ready Count Width (PLCLKCW) = 0b0 = 1040 cycles[2]
				Enable Fraction mode in feedback divider (FDINEN) = 0b0 - do not enable
3	Set up the SIUL2 MSCR [3]	Initialize the Zip wire clock pin as an input with CMOS levels to receive the clock from the Slave	SIUL2 Multiplexed Signal Control Register (MSCR)[4]	Master: MSCR = 0x0038_0001
4	Program the LVDS Control Register	The LCR is programed with configuration parameters of the LVDS	LCR = 0x0000_502C (default value)	SWWKLD, SWSLPLD, SWWKLR, SWSSLPLR, SWOFFLD, SWONLD, SWOFFLR, SWONLR = 0b0 - do not put the line driver or line receiver to sleep or disabled state
				LVDS Line Receiver off state (LVRXOFF) = 0b0 - low when the LFAST Receiver is in shutdown mode
				LVTXOE = 0b1 - output buffer enabled
				TXCMUX = 0b0 - do not put phase clock on the transmit pin
				LVRFEN = 0b1 - enable the LVDS reference
				LVLPEN = 0b0 - enable normal mode (not loopback mode)
				LVRXOP = 0b101 - enable receive termination and configure for maximum data rate
				LVTXOP = 0b1 - enable LFAST mode
				LVCKSS = 0b0 - use normal data mode
				LVCKP=0b0 - use the direct PLL clock

Table continues on the next page...

Table 12. LFAST Master mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
5	Enable LFAST and select Master or Slave operation.	Write MCR[MSEN] = 1. Then select LFAST modes by configuring MCR[CTSEN], MCR[TXSLPEN]and MCR[DATAEN]	Mode Configuration Register (MCR)	Master: MCR[MSEN] = 0b1
		Enable Data Frame Transmission.		MCR[DATAEN] = 0b1 - enable data frame transmission and reception
6	Use 20 MHz clock input with divide by 4 for PLL clock	Select the fraction of sysclk in Low Speed Select mode	Mode Configuration Register (MCR)	MCR[LSSEL] = 0b1 - divide by 4
7	Enable LFAST Transmitter Line Driver and Line Receiver	Write MCR[DRFEN] = 1 to enable the LFAST	Mode Configuration Register (MCR)	LFAST Driver/Receiver Enable (DRFEN) = 0b1 - Enable
8	Enable the Transmit and Receive circuits	Write MCR[RXEN] = 1 and MCR[TXEN] = 1 to negate the Line Driver (LD) powerdown, Line Receiver (LR) disable, and LR powerdown signals	Mode Configuration Register (MCR)	LVDS Transmit Enable (TXEN) = 0b1, LVDS Receiver Enable (RXEN = 0b1.
Repeat the following steps (9-16) until the slave acknowledges whether the Ping frame was successfully received or not from the Slave (failed)				
9	Send request to enable the slave Transmit interface	Write ICR[ICLCPLD] = 31h to enable the Slaves Tx Interface	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x31 - Enable Slave transmitter
10	Initiate a ICLC frame request and set to only allow ICLC frames to be sent	Write ICR[SNDICLC] = 1 and ICR[ICLCSEQ] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
				ICLC Enabled (ICLCSEQ) = 0b1= Enable the sending of only ICLC frames
11	Enable transmit arbiter	Write MCR[TXARBD] = 0	Mode Configuration Register (MCR)	Transmit Arbiter Disable (TXARBD) = 0x0 = Enable Tx arbiter and framer
12	Wait for ICLC frame to be transmitted	The ICLC transmission is confirmed by verifying one of the following: • ICR[SNDICLC] = 0 • TISR[TXICLCF] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b0 Frame transmitted
			Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
13	Clear TXICLCF	Write TISR[TXICLCF] = 1	Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1

Table continues on the next page...

Table 12. LFAST Master mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
14	Request a ping from the Slave node	Write ICR[ICLCPLD] = 00h to check the LFAST slaves status	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x00 Ping request from Master to Slave
15	Send the ping from slave request frame	Write ICR[SNDICLC] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
16	Confirm slave status	The LFAST slave status is confirmed by occurrence of one of the following: • LFAST slave is enabled if RIISR[ICPSF] = 1. Proceed to step 14 • LFAST slave is disabled if RIISR[ICPFF] = 1. The LFAST master must wait and restart from Step 7	Receive Interface Control Logic Channel (ICLC) Interrupt Status Register (RIISR)	Ping Frame Response successful (ICPSF) =0b1 (Continue initialization) or Ping Response Failed (ICPFF) = 0b1 (If fail, restart loop)
End of Wait for successful Slave communication loop				
17	Clear the Ping Frame Request Successful flag	Clear ICPSF	Receive Interface Control Logic Channel (ICLC) Interrupt Status Register (RIISR)	Ping Frame Response successful (ICPSF) = 1 = clear the response status
Speed Mode Change				
18	Begin change to high speed mode	Write PLLCR[SWPON] = 1 to enable the LFAST masters PLL	LFAST PLL Control Register (PLLCR)	Software signal to turn on the PLL (SWPON) = 0b1
19	Wait for PLL to relock	Wait for PLL disable signal to be negated and wait for PLL lock by confirming: • PLLLSR[PLLDIS] = 0 • then PLLLSR[PLDCR] = 1	LFAST PLL and LVDS Status Register (LFAST_PLLLSR)	PLL disable Status (PLLDIS) = 0 = PLL disable signal is negated.
				PLL Lock Delay Counter Ready (PLDCR) = 1 = PLL Lock delay counter is decremented to 0
20	Start PLL frame	Write ICLC start PLL frame, ICR[ICLCPLD] = 02h	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x02 = Start PLL in preparation for High Speed mode
21	Initiate transfer of ICLC frame	Write ICR[SNDICLC] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
22	Confirm ICLC transmission	The ICLC transmission is confirmed by occurrence of one of the following: • ICR[SNDICLC] = 0 • TISR[TXICLCF] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b0 Frame transmitted
			Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1

Table continues on the next page...

Table 12. LFAST Master mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
23	Clear TXICLCF	Write TISR[TXICLCF] = 1	Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
24	Change the LFAST masters RX interface speed	Both of the following operations are performed to change the LFAST masters Rx interface speed: **Note:** (The slaves Transmit interface speed mode should be changed first) • Write ICR[ICLCPLD] = 80h to select Rx data fast frame • Write ICR[SNDICLC] = 1	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x10 = Select Receive Data Fast frame
				ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
25	Confirm ICLC transmission	The ICLC transmission is confirmed by the occurrence of one of the following: • ICR[SNDICLC] = 0 • TISR[TXICLCF] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b0 Frame transmitted
			Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
26	Clear TXICLCF	Write TISR[TXICLCF] = 1	Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
27	Enable transmit fast frames	To change LFAST master's Transmit interface speed both of the following operations are performed: **Note:** (The slaves Rx interface speed should be changed first.) • Write ICR[ICLCPLD] = 10h for Tx data fast frame • Write ICR[SNDICLC] = 1	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x10 = Select Transmit Data Fast frame
				ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
28	Confirm ICLC transmission	The ICLC transmission is confirmed by the occurrence of one of the following: • ICR[SNDICLC] = 0 • TISR[TXICLCF] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b0 Frame transmitted
			Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
29	Clear TXICLCF	Write TISR[TXICLCF] = 1	Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
30	Clear ICLCSEQ	Write ICR[ICLCSEQ] = 0 to be able to set the TDR bit	ICLC Control Register (ICR)	ICLC enabled (ICLCSEQ) = 0
31	Change LFAST master Rx interface speed	Write SCR[TDR] = 1 to change the LFAST masters Tx interface speed.	Speed Control Register (SCR)	Transmit Data Rate (TDR) = 0b1 = High speed (320 Mb/s)
32	Change LFAST master Rx	Write SCR[RDR] = 1 to change the LFAST masters Rx interface speed	Speed Control Register (SCR)	Receive Data Rate (TDR) = 0b1= High speed (320 Mb/s)

Table continues on the next page...

Table 12. LFAST Master mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
	interface speed			
33	Confirm speed change	Wait until GSR reflects speed change by the following • GSR[LDSM] = 1 • GSR[DRSM] = 1	Global Status Register (GSR)	Transmit Interface Data Rate Status (LDSM) = 1 = Data rate of HIGH speed mode
				Receive Interface Data Rate Status (DRSM) = 1 = Data rate of HIGH speed mode
34	Enable ICLCSEQ	Write ICR[ICLCSEQ] = 1	ICLC Control Register (ICR)	ICLC enabled (ICLCSEQ) = 1
35	Send ping to confirm high speed operation	Write ICR[ICLCPLD] = 00h to confirm the change in speed of the LFAST slave. This frame should be written after waiting for the expected delay in the start of the PLL and speed mode change delay at the LFAST slave.	ICLC Control Register (ICR)	ICLC Payload (ICLCPLD) = 0x00 Ping request from Master to Slave
36		Write ICR[SNDICLC] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b1 = initiate the transfer of an ICLC frame
37	Confirm ICLC transmission	The ICLC transmission is confirmed by the occurrence of one of the following: • ICR[SNDICLC] = 0 • TISR[TXICLCF] = 1	ICLC Control Register (ICR)	ICLC Frame request (SNDICLC) = 0b0 Frame transmitted
			Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
38	Clear TXICLCF	Write TISR[TXICLCF] = 1	Transmit Interrupt Status Register (TISR)	Transmit ICLC Frame transmitted Interrupt (TXICLCF) = 0b1
39	Disable Arbiter and Framer	Write MCR[TXARBD] = 1 after some delay to ensure the ICLC frame is sent but no other frame is sent to arbitration	Mode Configuration Register (MCR)	Transmit Arbiter Disable (TXARBD) = 0x1 = Disable all other transmit frame requests
40	Confirm speed mode	The LFAST slaves speed mode is confirmed by the occurrence of the following: • If RIISR[ICPSF] = 1. The LFAST slave is in High Speed mode • If RIISR[ICPFF] = 1. The LFAST slave is in Low Speed mode	Receive Interface Control Logic Channel (ICLC) Interrupt Status Register (RIISR)	Ping Frame Response successful (ICPSF) =ob1
				Ping Frame Response Failed (ICPFF) = 0b1
41	Enable Arbiter and Framer	If RIISR[ICPSF] = 1, then all frame arbitration is enabled by both of the following operations: • Write ICR[ICLCSEQ] = 0 • Write MCR[TXARBD] = 0	ICLC Control Register (ICR)	ICLC Enabled (ICLCSEQ) = 0b0 = Enable Single ICLC frames only
			Mode Configuration Register (MCR)	Transmit Arbiter Disable (TXARBD) = 0x0 = enable transmit arbiter and framer.

1. Unless otherwise noted, all registers are in the LFAST module.

2. It is possible that this value could be reduced, however, it is dependent on the actual crystal and system frequencies, as well as board design parameters such as PLL stability, power supply stability, board layout, and other operating conditions.
3. The SIUL2 module is labeled as SIU in some devices.
4. The exact MSCR register and value will depend on the device type and whether the device is being programmed for Master or Slave operation.

When operating in slave mode, the Zipwire interface requires much less initialization. The following table shows the configuration of the MCU for the LFAST Slave interface.

Table 13. LFAST Slave mode configuration

Step	Description	RM Operation	Registers[1]	Bits
0	Initialize device pins for the Zipwire interface	Set up the Positive and Negative Transmit pins in the System Integration Unit Lite 2 (SIUL2) Multiplexed Signal Control Registers (MSCR)	See Table 11	
		Set up the Positive and Negative Receive pins in the SIUL2 MSCR		
1	Set the LFAST wakeup delay and rate change delay for the Line Driver (LD).	After reset the SLCR and RCDCR are programmed according to the LVDS requirements of the device.	LFAST Rate Change Delay Control Register (RCDCR)	Data Rate Controller Count Value (DRCNT) = 0xF
			LFAST Wakeup Delay Control register (SLCR)	High Speed Sleep Mode Exit Time (HSCNT) = 0x12 (18 cycles)
				Low Speed Sleep Mode Exit Time (LSCNT) = 0x1
				Wake Up time for the LD (HWKCNT) = 0x54
				Wake Up time for the LD (LWKCNT) = 0x2
2	Set the LFAST operating speeds	The PLLCR is programmed with configuration parameters for the PLL	LFAST PLL Control Register (PLLCR)	PLL Loop Optimization (LPCFG) = 3 (2x IBASE current
				Division Factor for the PLL Reference Clock (PREDIV) = 0 (Direct Clock)
				Feedback Division factor for PLL Reference Clock (FBDIV) = 15
				Test mode programmability (IPTMODE) = 0b000 Normal functional mode. Test modes should not be used by customers.
				SW signal to turn off the PLL (SWPOFF) = 0b0. The PLL should not be disabled for normal operation.
				Software signal to turn on the PLL (SWPON) = 0b0 = Do not turn on the PLL

Table continues on the next page...

Table 13. LFAST Slave mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
				Invert reference clock edge to the PFD (REVINV) = 0b0 = The reference clock should not be inverted
				PLL Lock Ready Count Width (PLCLKCW) = 0b0 = 1040 cycles
				Enable Fraction mode in feedback divider (FDINEN) = 0b0 - do not enable
3	Set up the SIUL2 MSCR [2]	Initialize the Zipwire clock pin as an output to send the clock to the Master	SIUL2 Multiplexed Signal Control Register (MSCR)[3]	Slave: MSCR = 0x2280_0001
4	Program the LVDS Control Register	The LCR is programed with configuration parameters of the LVDS	LCR = 0x0000_502C (default value)	SWWKLD, SWSLPLD, SWWKLR, SWSSLPLR, SWOFFLD, SWONLD, SWOFFLR, SWONLR = 0b0 - do not put the line driver or line receiver to sleep or disabled state
				LVDS Line Receiver off state (LVRXOFF) = 0b0 - low when the LFAST Receiver is in shutdown mode
				LVTXOE = 0b1 - output buffer enabled
				TXCMUX = 0b0 - do not put phase clock on the transmit pin
				LVRFEN = 0b1 - enable the LVDS reference
				LVLPEN = 0b0 - enable normal mode (not loopback mode)
				LVRXOP = 0b101 - enable receive termination and configure for maximum data rate
				LVTXOP = 0b1 - enable LFAST mode.
				LVCKSS = 0b0 - use normal data mode
				LVCKP=0b0 - use the direct PLL clock

Table continues on the next page...

Table 13. LFAST Slave mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
5	Enable LFAST and select Master or Slave operation	Write MCR[MSEN] = 1. Then select LFAST modes by configuring MCR[CTSEN], MCR[TXSLPEN]and MCR[DATAEN]	Mode Configuration Register (MCR)	Slave: MCR[MSEN] = 0b0
		Enable Data Frame Transmission.		MCR[DATAEN] = 0b1 - enable data frame transmission and reception
6	Use 20 MHz clock input with divide by 4 for PLL clock	Select the fraction of sysclk in Low Speed Select mode	Mode Configuration Register (MCR)	MCR[LSSEL] = 0b1 - divide by 4
7	Enable LFAST Transmitter Line Driver and Line Receiver	Write MCR[DRFEN] = 1 to enable the LFAST	Mode Configuration Register (MCR)	LFAST Driver/Receiver Enable (DRFEN) = 0b1 - Enable
8	Enable LFAST Receiver	The LR is enabled by writing MCR[RXEN] = 1. This also disables the Rx LVDS Line Receiver	Mode Configuration Register (MCR)	LFAST Receiver Enable (RXEN) = 0b1 = Receiver Interface is Enabled
9	Configure to send ping automatically	Configure automatic ping and set slave to receive ICLC frame at appropriate speed by: • writing PICR[PNGAUTO] = 1 • writing SCR[DRMD] = 1.	Ping Control Register(PICR)	Ping Response Frame Request (PNGAUTO) = 0b1 = Ping response frame transmission request is queued
			Speed Control Register (SCR)	Data Rate Controller mode (DRMD) = 0b1 = In LFAST Slave, the reception of ICLC frame for rate change sets appropriate speed mode.
10	Confirm LD enable signal is received	The LD enable signal will be received by the master sending an ICLC of 0x31. Reception can be confirmed by either: • RIISR[ICTEF] = 1 • MCR[TXEN] = 1	Rx ICLC Interrupt Status Register (RIISR)	ICLC frame for LFAST Slaves Tx Interface Enable received (TCTEF) = 0b1 = Interrupt event has occurred
			Mode Configuration Register (MCR)	LFAST Transmitter Enable (TXEN) = 0b1 = LFAST transmitter Interface is enabled. New requests are accepted
11	Send ping frame	After a write to MCR[TXARBD] = 0, a ping frame will be sent on one of the following conditions: PICR[PNGAUTO]=1 and 0x00 is received in ICLC	Mode Configuration Register (MCR)	Tx Arbiter Disable (TXARBD) = 0b0 = Enable Tx arbiter and framer
Speed Mode Change				

Table continues on the next page...

Table 13. LFAST Slave mode configuration (continued)

Step	Description	RM Operation	Registers[1]	Bits
12	Confirm ping frame reception	Ping frame reception is confirmed if RIISR[ICPRF] = 1.	Rx ICLC Interrupt Status Register (RIISR)	ICLC Ping Frame Request Received (ICPRF) = 0b1 = Interrupt event has occurred
13	Confirm PLL ON is received	ICLC frame for PLL ON has been received when RIISR[ICPONF] = 1		ICLC from for PLL ON received (ICPONF) = 0b1 = Interrupt event has occurred
14	Wait for PLL lock	Wait for PLL to lock by confirming both: • PLLLSR[PLLDIS] = 0 • PLLLSR[PLDCR] = 1	PLL and LVDS Status Register (PLLSR)	PLL disable Status (PLLDIS) = 0b0 = PLL disable signal is negated
				PLL Lock Delay Counter Ready (PLDCR) = 0b1 = PLL Lock delay counter is decremented to 0
15	Confirm Tx interface speed change	The speed of the Tx interface is changed on one of the following conditions when SCR[DRMD] = 1 and an ICLC frame with payload 80h is received.: • H/W writes RIISR[ICTFF] = 1 • H/W writes SCR[TDR] = 1.	Rx ICLC Interrupt Status Register (RIISR)	ICLC frame for LFAST Slaves Tx Interface fast mode switch received (ICTFF) = 0b1 = Interrupt event has occurred
			Speed Control Register (SCR)	Transmit Data Rate (TDR) = 0b1 = Data rate of Tx block is 312/320 Mb/s
16	Confirm Rx interface speed change	The speed of the Rx interface is changed on one of the following conditions when SCR[DRMD] = 1 and an ICLC frame with payload 10h is received.: • H/W writes RIISR[ICRFF] = 1. • H/W write SCR[RDR] = 1.	Rx ICLC Interrupt Status Register (RIISR)	ICLC frame for LFAST Slaves Rx Interface fast mode switch received (ICRFF) = 1 = Interrupt event has occurred
			Speed Control Register (SCR)	Receiver Data Rate (TDR) = 0b1 = Data rate of Rx block is 312/320 Mb/s

1. Unless otherwise noted, all registers are in the LFAST module.
2. The SIUL2 module is labeled as SIU in some devices.
3. The exact MSCR register and value will depend on the device type and whether the device is being programmed for Master or Slave operation.

5 Zipwire hardware and layout

The Zipwire interface is intended to be used to communicate between two nodes implemented on a single board. The interface uses a "low speed" reference clock that is shared between the two nodes. A single-ended 10 to 26[4] MHz reference clock is used to generate the Zipwire high speed operation of approximately 320 MHz. A termination resistor is required at the receiving end of the clock for best performance of the interface. The value of the resistor depends on the board layout and impedance.

The data signals use a low voltage differential signaling (LVDS) that is internally terminated on the MCU.

The following diagram shows the connection between two devices.

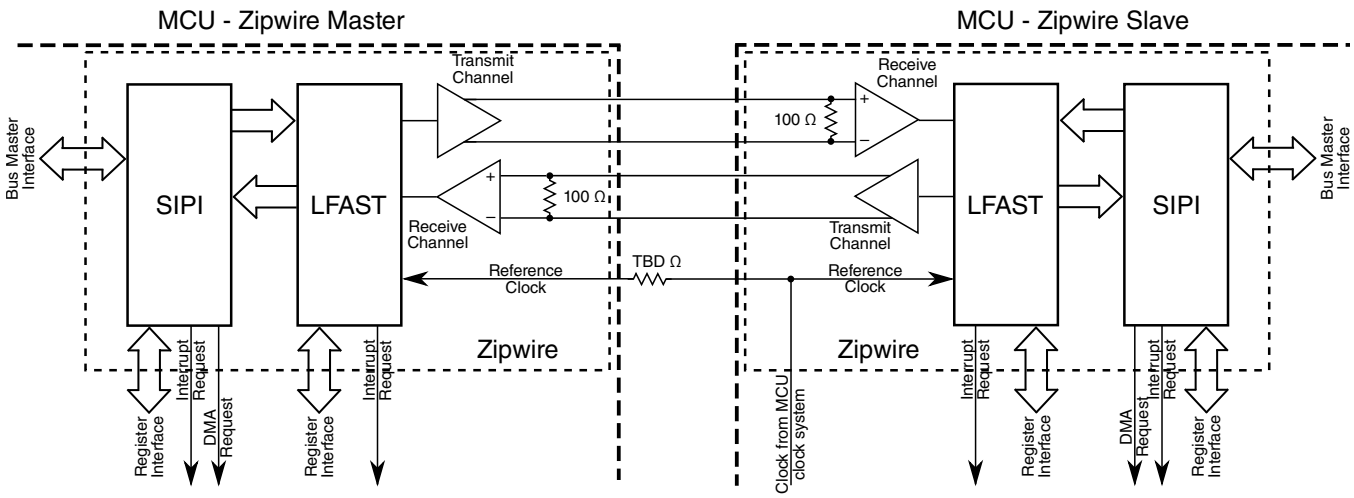

Figure 5. Typical Zipwire hardware interface

The Zipwire interface is a high-speed interface, therefore care should be taken in laying out the signals on a printed circuit board. The following guidelines are suggested.

- A controlled impedance PCB is required for the LVDS signals.
- The differential LVDS + and – pair should be routed parallel and close to each other. The length of the + and - pairs should be matched to less than 0.05 inches of difference.
- The LVDS transmit pairs should be of the same approximate length (within 0.1 inches). The receive pins should also be the approximate length (within 0.1 inches), but are not required to be the same length as the transmit signals.
- The differential pair should be routed with a maximum of two vias. Ideally, the differential pair should be routed without vias on a single plane of the board preferably on the top or bottom plane of the board. However, due to pin escape issues with the placement of the high speed signals on the surface mounted devices, routing on a single layer is not possible.
- Keep necking of the signal to less than 0.01 inch to avoid discontinuities. Some necking is usually required in escaping the signals for the BGA or LQFP signal feeds to other layers on the board.
- The differential pair must be routed on a layer that is one dielectric away from ground.
- A connector is not recommended for the Zipwire interface, but if a connector is used, a high speed connector system, such as the Samtec ERF8 0.8 mm Edge Rate Rugged High Speed Socket, should be used with twin-ax cabling. The odd side of the connector should be placed parallel and nearest to the MCU package on the board to allow direct connection to the package signals. See Zipwire connector.

Appendix A Zipwire driver

This section describes the Freescale Zipwire driver that is used in the Zipwire example code.

The Zipwire driver requirements and APIs are described in the API Reference section. The Zipwire driver implements software to control the SIPI and LFAST modules of the MCU.

4. 20 MHz is the most commonly used frequency, 10 MHz can also be used. 26 MHz is not recommended.

A.1 Overview

The Zipwire Inter-processor Communication Interface is a combination of the Serial Interprocessor Interface module and the LVDS Fast Asynchronous Serial Transmission interface. This provides a standard interface for communicating at a high speed between two microcontrollers or between a microcontroller and a smart peripheral. This driver provides a standard Application Programming Interface to use the Zipwire interface in user written software.

A.2 About this Appendix

This Technical Reference Appendix employs the following typographical conventions:

- **boldface** type: Bold is used for important terms, notes and warnings.
- `courier` font: Courier typeface is used for code snippets in the text. Note that C language modifiers such “const” or “volatile” are sometimes omitted to improve readability of the presented code.

Notes and warnings are shown below:

Note

This is a note.

A.3 Zipwire Driver API

The Zipwire driver API implements the basic functions required to use the Zipwire interface as detailed in this document.

A.3.1 Driver Design Summary

The Zipwire driver API provides services for the following features:

- LOW LEVEL
  - LFAST Initialization for High Speed mode and Low Speed mode (20 MHz and 10 MHz)
  - SIUL2 configuration for Zipwire pins and clock signal for both, initiator and target mode.
  - Clock Initialization for AUXCLOCK 1 (LFAST exclusive clock)
- API LEVEL
  - SIPI initiator and Target Initializations
  - SIPI Mode Changes
  - SIPI Channel Initialization.
  - Read Transfer Operation
  - Write Transfer Operation (DMA Supported)
  - Stream Transfer Operation
  - Event Trigger Operation
  - ID Operation

All of the Zipwire API functions are included in a single 'C' source code file.

Table A-1. Zipwire function file locations

File Name	Description	Function
SIPI_HSSL_API.c	Zipwire API functions	uint8_t SIPI_read(DATA_TEMPLATE_t data_address, CHANNEL_t channel, uint8_t injected_error)
		uint32_t SIPI_read_channel_data(CHANNEL_t channel)
		uint8_t SIPI_multiple_read(DATA_TEMPLATE_t * read_array, uint16_t array_length, CHANNEL_t channel, uint8_t injected_error, uint32_t * read_temp)
		uint8_t SIPI_write(DATA_TEMPLATE_t write_data, CHANNEL_t channel, uint8_t injected_error)
		uint8_t SIPI_multiple_write(DATA_TEMPLATE_t write_array[], uint16_t array_length, CHANNEL_t channel, uint8_t injected_error, uint8_t DMA_Enable, uint32_t * dma_array)
		uint8_t SIPI_ID(uint32_t * id_array, CHANNEL_t channel)
		uint8_t SIPI_init_INITIATOR(uint16_t Clock_Prescale)
		uint8_t SIPI_init_TARGET(uint32_t max_count, uint32_t reload_address, uint8_t Add_Inc)
		uint8_t SIPI_init_channel(CHANNEL_t channel, uint8_t mode, uint8_t error_int_enable, uint8_t data_int_enable)
		uint8_t SIPI_Trigger(CHANNEL_t channel)
		uint8_t SIPI_get_initiator_event(uint8_t channel_number)
		uint8_t SIPI_reset()
		uint8_t SIPI_module_mode(uint8_t Mode)

In addition to the API code, there are additional header files that are included in the Zipwire API. These are shown in the following table.

Table A-2. Zipwire API header files

File Name	Description
SIPI_HSSL_Header_v4.h	This header file provides all of the prototype functions for the Zipwire driver. This file can be included in the target software to include the Zipwire API functions.
SIPI_API.h	This header file contains all of the necessary definitions for the Zipwire driver internal use.

A.3.2 API Reference

This section contains description of the Zipwire driver API.

A.3.2.1 Function Index

Table A-3. Quick Function Reference

Type	Name	Arguments
uint8_t	SIPI_ID	uint32_t * id_array CHANNEL_t channel
uint8_t	SIPI_Trigger	CHANNEL_t channel
uint32_t	SIPI_get_initiator_event[1]	uint8_t channel_number
uint8_t	SIPI_init_INITIATOR	uint16_t Clock_Prescale
uint8_t	SIPI_init_TARGET	uint32_t max_count uint32_t reload_address uint8_t Add_Inc
uint8_t	SIPI_init_channel	CHANNEL_t channel uint8_t mode uint8_t error_int_enable uint8_t data_int_enable
uint8_t	SIPI_module_mode	uint8_t Mode
uint8_t	SIPI_multiple_read	DATA_TEMPLATE_t read_array[] uint16_t array_length CHANNEL_t channel uint8_t injected_error uint32_t * read_temp
uint8_t	SIPI_multiple_write	DATA_TEMPLATE_t write_data CHANNEL_t channel uint8_t injected_error
uint8_t	SIPI_read	DATA_TEMPLATE_t data_address CHANNEL_t channel uint8_t injected_error
uint32_t	SIPI_read_channel_data	CHANNEL_t channel
uint8_t	SIPI_reset	void
uint8_t	SIPI_stream_transfer	uint32_t * temp_data_stream uint8_t initiator uint8_t length
uint8_t	SIPI_write	DATA_TEMPLATE_t write_data CHANNEL_t channel uint8_t injected_error

1. SIPI_get_initiator_event not implemented

A.3.2.2 Function SIPI_reset

Performs soft reset of module. Clears all status and error registers, returning the module to 'Disabled'. Any transfers in progress when reset is called will immediately end. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_reset(void);`

Return:

- 0 = Successfully Reset on Module

A.3.2.3 Function SIPI_init_TARGET

Initializes Target side of SIPI module, setting SIPI_MCR[TEN], SIPI_MAXCR and SIPI_ARR. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_init_TARGET(uint32_t max_count, uint32_t reload_address, uint8_t Add_Inc);`

Table A-4. SIPI_init_TARGET Arguments

Type	Name	Direction	Description
uint32_t	max_count	**input**	Maximum address count value of target node
uint32_t	reload_address	**input**	Reload value for the address counter at target node
uint8_t	Add_Inc	**input**	Integer representation of Address Increment/Decrement bits. Can be 0,1,2 or 3 for 'No change, Increment Address by 4, Decrement Address by 4, or Not Used respectively.

Return:

- 0 = Successfully Set Up Target Node
- 1 = Address Increment Error
- 2 = Max Count Address Conflicts with Address Count

A.3.2.4 Function SIPI_init_channel

Initializes SIPI Channels. Sets up SIPI_CIRn registers. Will also need to set up interrupts/events to handle received packets appropriately. Clears all errors and events associated with the channel. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_init_channel(CHANNEL_t channel, uint8_t mode, uint8_t error_int_enable, uint8_t data_int_enable);`

Table A-5. SIPI_init_channel Arguments

Type	Name	Direction	Description
`CHANNEL_t`	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.
uint8_t	mode	**input**	Sets the channel mode to be used. Set 0 - Run, 1 - Disabled, 2 - clear error, 3 - stop request;
uint8_t	error_int_enable	**input**	Set to 1 to enable error interrupts on the channel
uint8_t	data_int_enable	**input**	Set to 1 to enable data interrupts on the channel

Return:

- 0 = Successfully Set Up Channel
- 1 = Incorrect Channel Mode
- 2 = Incorrect Channel

A.3.2.5 Function SIPI_module_mode

Puts the SIPI module into the required mode. Must be used to place module into 'INIT' mode before calling the SIPI_init functions. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_module_mode(uint8_t Mode);`

Table A-6. SIPI_module_mode Arguments

Type	Name	Direction	Description
uint8_t	Mode	**input**	Integer representation of required mode. 0 = Disabled, 1 = Enabled/Init, 2 = Enabled/Run

Return:

- 0 = Successfully Set Up Module Mode
- 1 = Invalid Mode Selected

A.3.2.6 Function SIPI_init_INITIATOR

Initializes Initiator side of SIPI module, setting SIPIMCR with clk prescale, AID and MOEN. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_init_INITIATOR(uint16_t Clock_Prescale);`

Table A-7. SIPI_init_INITIATOR Arguments

Type	Name	Direction	Description
uint16_t	Clock_Prescale	**input**	Integer representation of Prescale for Timeout Clock. Default is 64. Can be 64, 128, 256, 512 or 1024.

Return:

- 0 = Successfully Set Up Initiator Node
- 1 = Incorrect Clock Prescale

A.3.2.7 Function SIPI_get_initiator_event

Returns 32 bit register showing event status for the channel. Should be polled with mask within calling function to determine if transactions have completed successfully.

Prototype: `uint32_t SIPI_get_initiator_event(uint8_t channel_number);`

Table A-8. SIPI_get_initiator_event Arguments

Type	Name	Direction	Description
uint8_t	channel_number	**input**	SIPI Channel to use.

Return:

- 0 = Incorrect Channel
- SW Channel Status Register

A.3.2.8 Function SIPI_ID

Sends ID Request Frame to target. Stores received command in the address passed. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_ID(uint32_t *id_array, CHANNEL_t channel);`

Table A-9. SIPI_ID Arguments

Type	Name	Direction	Description
uint32_t *	id_array	**input**	used as a dummy data to set CAR to initiate the transfer
`CHANNEL_t`	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.

Return:

- 0 = Successfully Received Acknowledge and ID
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error

A.3.2.9 Function SIPI_read_channel_data

Reads channel data when a successful read reply / ID reply is received.

NOTE

The function will always return a 32 bit value. If 8 or 16 bit data is read, it will be replicated as described in the RM. A relevant casting / mask operation may be required for 8 and 16 bit read replies. Need to check for command completion and channel errors before calling this function.

Prototype: `uint32_t SIPI_read_channel_data(CHANNEL_t channel);`

Table A-10. SIPI_read_channel_data Arguments

Type	Name	Direction	Description
`CHANNEL_t`	channel	**input**	SIPI Channel the received data used.

Return:

- 0 = Invalid Channel
- 32-bit value contained in Channel data register.

A.3.2.10 Function SIPI_read

Performs a single read transfer. Returns '0' if successful, error code otherwise. Stores read value in `DATA_TEMPLATE_t` passed.

Prototype: `uint8_t SIPI_read(DATA_TEMPLATE_t data_address, CHANNEL_t channel, uint8_t injected_error);`

Table A-11. SIPI_read Arguments

Type	Name	Direction	Description
DATA_TEMPLATE_t	data_address	**input**	DATA_TEMPLATE_t structure which includes read Address and data size.
CHANNEL_t	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.
uint8_t	injected_error	**input**	injected error (if required) - Not currently implemented

Return:

- 0 = Successful
- 1 = Invalid Width
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error

A.3.2.11 Function SIPI_multiple_read

Performs a direct read transfer. Returns '0' if successful, error code otherwise. Stores read values in struct pointed to. Should call SIPI_read() to process each read in array.

Prototype: `uint8_t SIPI_multiple_read(DATA_TEMPLATE_t *read_array, uint16_t array_length, CHANNEL_t channel, uint8_t injected_error, uint32_t *read_temp);`

Table A-12. SIPI_read_channel_data Arguments

Type	Name	Direction	Description
DATA_TEMPLATE_t	read_array	**input**	Pointer to DATA_TEMPLATE_t structure which includes read address.
unit16_t	array_length	**input**	Amount of data elements in array to be sent.
CHANNEL_t	channel	**input**	SIPI Channel to use. Should be passed as one element of CHANNEL array.
uint8_t	injected_error	**input**	injected error (if required).
uint32_t	read_temp	**output**	Provides a pointer to a data structure that will store all read data.

Return:

- 0 = Invalid Channel
- 32-bit value contained in Channel data register.

A.3.2.12 Function SIPI_write

Performs a direct write transfer. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_write(DATA_TEMPLATE_t write_data, CHANNEL_t channel, uint8_t injected_error);`

Table A-13. SIPI_write Arguments

Type	Name	Direction	Description
DATA_TEMPLATE_t	write_data	**input**	DATA_TEMPLATE_t structure which includes write Address and Data to be written
CHANNEL_t	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.
uint8_t	injected_error	**input**	injected error (if required)

Return:
- 0 = Successfully Set Up
- 1 = Invalid Data Size
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error / Wrong Acknowledge

A.3.2.13 Function SIPI_multiple_write

Performs multiple transfers. Returns '0' if successful, error code otherwise. SIPI_write should be called to process each separate write in array, and poll for the SIPI_write_ack function to complete before moving to next message.

Prototype: `uint8_t SIPI_multiple_write(DATA_TEMPLATE_t *write_array, uint16_t array_length, CHANNEL_t channel, uint8_t injected_error, uint8_t DMA_Enable, uint32_t *dma_array);`

Table A-14. SIPI_multiple_write Arguments

Type	Name	Direction	Description
DATA_TEMPLATE_t *	write_array	**input**	DATA_TEMPLATE_t structure which includes array containing write Address and Data to be written for each array record.
uint16_t	array_length	**input**	Amount of data elements in array to be written.
CHANNEL_t	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.
uint8_t	injected_error	**input**	injected error (if required)
uint8_t	DMA_Enable	**input**	Selects whether DMA should be used for transfer or software. Software will form blocking function which will run until all writes complete.
uint32_t *	dma_array	**input**	receives a pointer to an array of integers that contains DMA friendly structure

Return:
- 0 = Successfully Set Up
- 1 = Invalid Data
- 2 = Channel Busy
- 3 = Invalid Channel
- 4 = Timeout Error / Wrong Acknowledge

A.3.2.14 Function SIPI_Trigger

Sends Trigger Request Frame to target. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_Trigger(CHANNEL_t channel);`

Table A-15. SIPI_Trigger Arguments

Type	Name	Direction	Description
`CHANNEL_t`	channel	**input**	SIPI Channel to use. Should be passed as 1 element of CHANNEL array.

Return:

- 0 = Successfully Sent Trigger Command
- 1 = Incorrect Channel
- 2 = Channel Busy

A.3.2.15 Function SIPI_stream_transfer

Performs a streaming write transfer. Returns '0' if successful, error code otherwise.

Prototype: `uint8_t SIPI_stream_transfer(uint32_t *temp_data_stream, uint8_t initiator, uint8_t length);`

Table A-16. SIPI_stream_transfer Arguments

Type	Name	Direction	Description
uint32_t *	temp_data_stream	**input**	Pointer to address containing start of data to be streamed.
uint8_t	initiator	**input**	Decides which configuration will be taken, Initiator or Target.
uint8_t	length	**input**	amount of bytes to be sent

Return:

- 0 = Stream Successful
- 1 = Invalid Length
- 2 = Acknowledge Error

A.3.3 Structure definitions

This section describes structures that are used by the Zipwire driver.

A.3.3.1 Structure CHANNEL_t

CHANNEL structure should be utilized as a 4 element array in the application (for SIPI channels 1-4).

Declaration

```
typedef struct
{
  uint8_t Number,
```

```
  uint8_t Timeout
} CHANNEL_t;
```

Table A-17. Structure CHANNEL_t member description

Member	Description
Number	Defines the Zipwire Channel Number that will be used.
Timeout	Can be used to hold if a specific channel request has timed out.

A.3.3.2 Structure DATA_TEMPLATE_t

DATA_TEMPLATE structure should contain write Address and Data pointers for SIPI_write command,read Address pointer and Data size for SIPI_read.

Declaration

```
typedef struct
{
  uint32_t Data,
  uint32_t Address,
  uint16_t Size
} DATA_TEMPLATE_t;
```

Table A-18. Structure DATA_TEMPLATE_t member description

Member	Description
Data	Data that will be transferred
Address	Holds the address where the data will be written or read from on target.
Size	Specifies the size of the write and read values.

Appendix B Zipwire connector

In most cases, the Zipwire interface will be implemented between two devices on the same printed circuit board (PCB). For evaluation purposes, a connector may be desired. The tables below show the recommended connectors and the recommended pin out for a Zipwire interface. The recommended connector is a 10-pin (5 position) from Samtec. Cable locks on each end of the connector are also grounded.

CAUTION

Zipwire is intended to be used between two devices on the same printed circuit board. This connector and cable is only intended for evaluation of products and not for actual customer implementations.

The differential signals should ideally be implemented as twin-axial cables if a cable is used. When the devices are located on the same board, the signals should be routed as matched impedance pairs.

Table B-1. Recommended Connector (Samtec part numbers)

Target connector	ERF8-005-05.0-L-DV-L-TR
12 inch cross-over cable[1]	HDR-169378-xx[2]

1. Approximate length.

2. xx is the revision number

Table B-2. MPC57xx SIPI connector

Position	Signal	Direction	Pin number	Pin number	Direction	Signal
			GND			
1	SIPI_TXP	In	1	2	GND	Ground reference
2	SIPI_TXN	In	3	4	GND	Ground reference
3	Ground	GND	5	6	In/Out	REFCLK
4	SIPI_RXN[1]	Out	7	8	GND	Ground reference
5	SIPI_RXP	Out	9	10	GND	Ground reference
			GND			

1. Initial definition of this connector (prior to November 2011) had the SIPI_RXP and SIPI_RXN reversed.

The table below shows the descriptions of the SIPI/LFAST signals.

Table B-3. SIPI signal descriptions

Signal	Direction (as viewed by the MCU)	Description
SIPI_TXP	Input	Zipwire LFAST Transmit Positive terminal
SIPI_TXN	Input	Zipwire LFAST Transmit Negative terminal
SIPI_RXP	Output	Zipwire LFAST Receive Positive terminal
SIPI_RXN	Output	Zipwire LFAST Receive Negative terminal
REFCLK	Input or output	LFAST reference clock. This should be either 10 or 20 MHz. The clock is always generated by the slave LFAST device and is generated from the MCU PLL0:PHI clock. The master SIPI device uses the DRCLK as its reference.

A cross-over cable will be available from Samtec/Freescale to connect two Zipwire interfaces together that are located on separate boards. The initial cable available will be approximately 12 inches in overall length.

The pin out of the cross-over cable is shown in the following table.

NOTE

One device must be set to be the master and the other device must be set as a slave to avoid contention on the DRCLK signal.

Table B-4. Cross-over cable connections

Master Signal	Connector M Pin		Connector S Pin	Slave signal
SIPI_TXP	1	——>	9	SIPI_RXP
SIPI_TXN	3	——>	7	SIPI_RXN
REFCLK	6	<——	6	REFCLK
SIPI_RXN	7	<——	3	SIPI_TXN
SIPI_RXP	9	<——	1	SIPI_RXP
GND	2, 4, 5, 8, and 10	<——>	2, 4, 5, 8, and 10	GND

The figure below shows a drawing of a cable that is available from Freescale engineering.

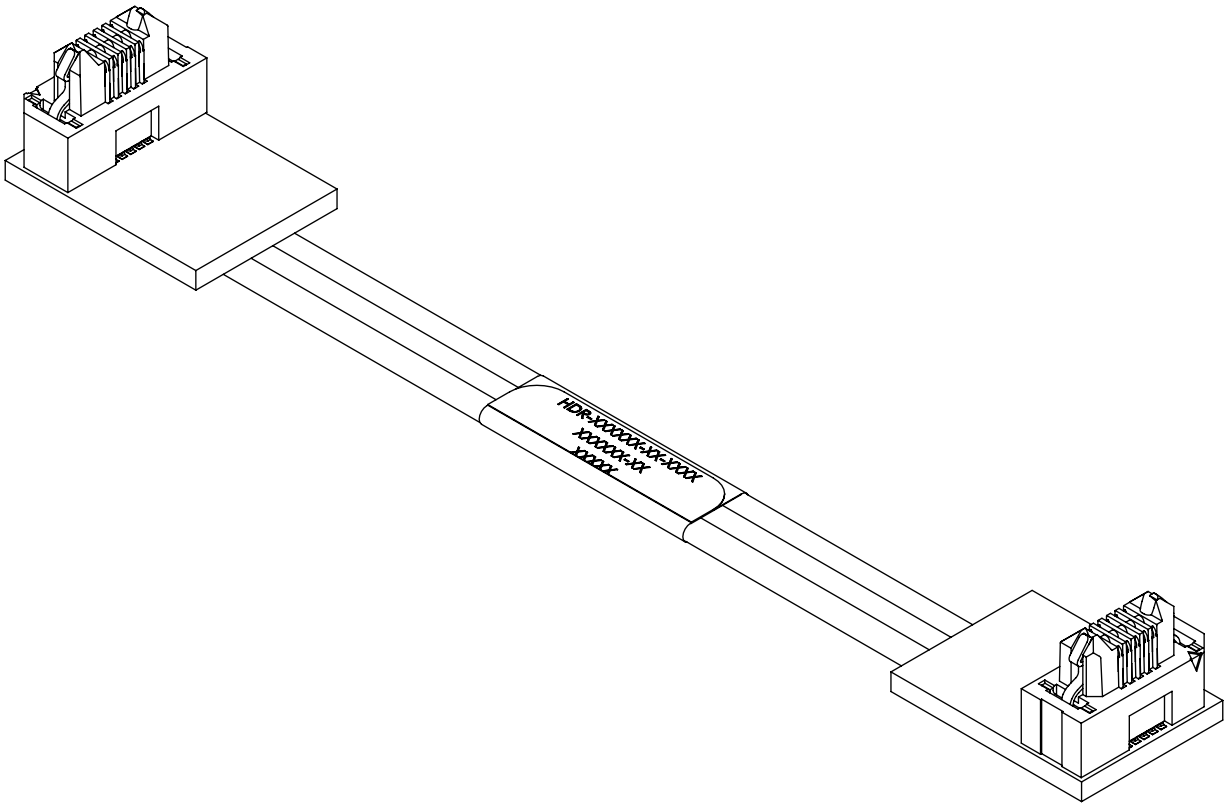

Figure B-1. HDR-169378-01 cable drawing

Appendix C References

Additional information can be found in the documentation listed below.

Table C-1. References

Document	Title	Availability
AN4566	MPC5746M Hardware Design	freescale.com
AN4812	Initializing the MPC5777M Clock Generation Module and Progressive Clock Switching Feature	freescale.com
MPC574xPRM, MPC5777CRM, MPC577xKRM, MPC5777MRM, S32V234RM	Device Reference Manual	

Appendix D Revision history

Table D-1. Revision history

Revision	Description	Date
1	Initial customer release	May 2015

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, SafeAssure, and SafeAssure logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. ARM and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. mbed is a trademark of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other product or service names are the property of their respective owners.

© 2015 Freescale Semiconductor, Inc.

Document Number AN5134
Revision 0, May 2015