Freescale Semiconductor
Application Note

Document Number: AN4864
Rev 1, 04/2014

Generic Timer Module (GTM) Serial Peripheral Interface (SPI) Bus Emulation

by: Inga Harris

Contents

1 Introduction

This Generic Timer Module (GTM) Serial Peripheral Interface (SPI) Bus Emulation application note is intended to provide details of how to emulate an SPI bus master Slave Select (SS), Serial Clock (SCK), Master Output Slave Input (MOSI), and Master Input Slave Output (MISO) signals using the GTM Multi-Channel Sequencer (MCS) submodule, Timer Input Module (TIM) and the ARU connected Timer Output Module (ATOM) submodules. The assembly functions are portable to any product that has a GTM module. Porting the application code which configures the chip and GTM from one chip to another does require minor changes. Example code in this application note is based on the MPC5777M device. This application note should be read in conjunction with application note AN4351,“MPC57xxM Generic Timer Module (GTM) Quick Start Guide” available at freescale.com

2 Overview

The SPI bus is a full duplex synchronous serial data link between a master and slave devices, where the master initiates the data transmission. It is a common communication protocol used in many embedded applications.

An SPI interface is commonly emulated in software where a dedicated hardware peripheral is not available. The solution presented in this application note emulates the interface

outputs and input in the GTM MCS module with only a small amount of software running on the chip core for configuration and MISO reception, which means that the emulated interface does not consume a lot of CPU bandwidth and only consumes two MCS channels, with three ATOM channels, a TIM channel, and a GPIO.

The given example transmits 8 bits of data (synchronous to SCK) and a single SS control line. The example can receive the same message by connecting SCK to a TIM channel, and the MOSI connected to an input port (MISO). The data is stored in the MCS RAM at compile time for simplicity, but it could also be moved to the RAM through DMA or read by the MCS through the PSM submodule in a full application environment. The received data is presented back to the GTM through the PSM FIFO RAM.

The transmission of SS, SCK, and MOSI is controlled by the master. In this example the SPI is operating in mode 0 (CPOL = 0, CPHA = 0).

Figure 1 shows an example transmission from the system where the data is captured on a clock rising edge and the data is propagated on a clock falling edge:

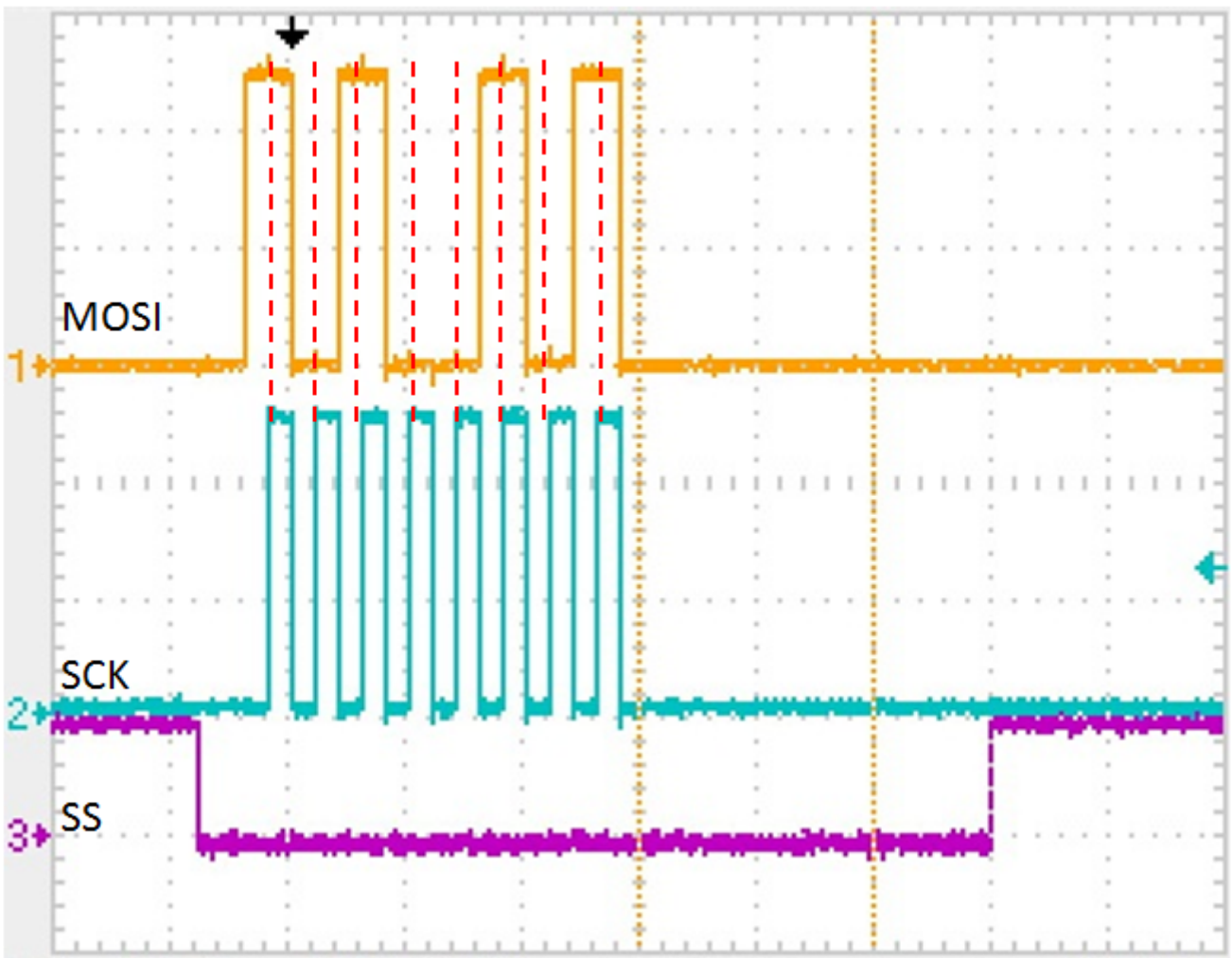

Figure 1. Example output waveform

Table 1. Signal details

SPI Signal	Channel	Color	X Scale	Y Scale	Characteristics
MOSI	1	Yellow	5 µs	2 V	Transmitting 0xA5
SCK	2	Blue	5 µs	2 V	500 KHz
SS	3	Purple	5 µs	5 V	Active low

3 Block diagram

The SPI solution given in this application note uses the following GTM104 submodules:

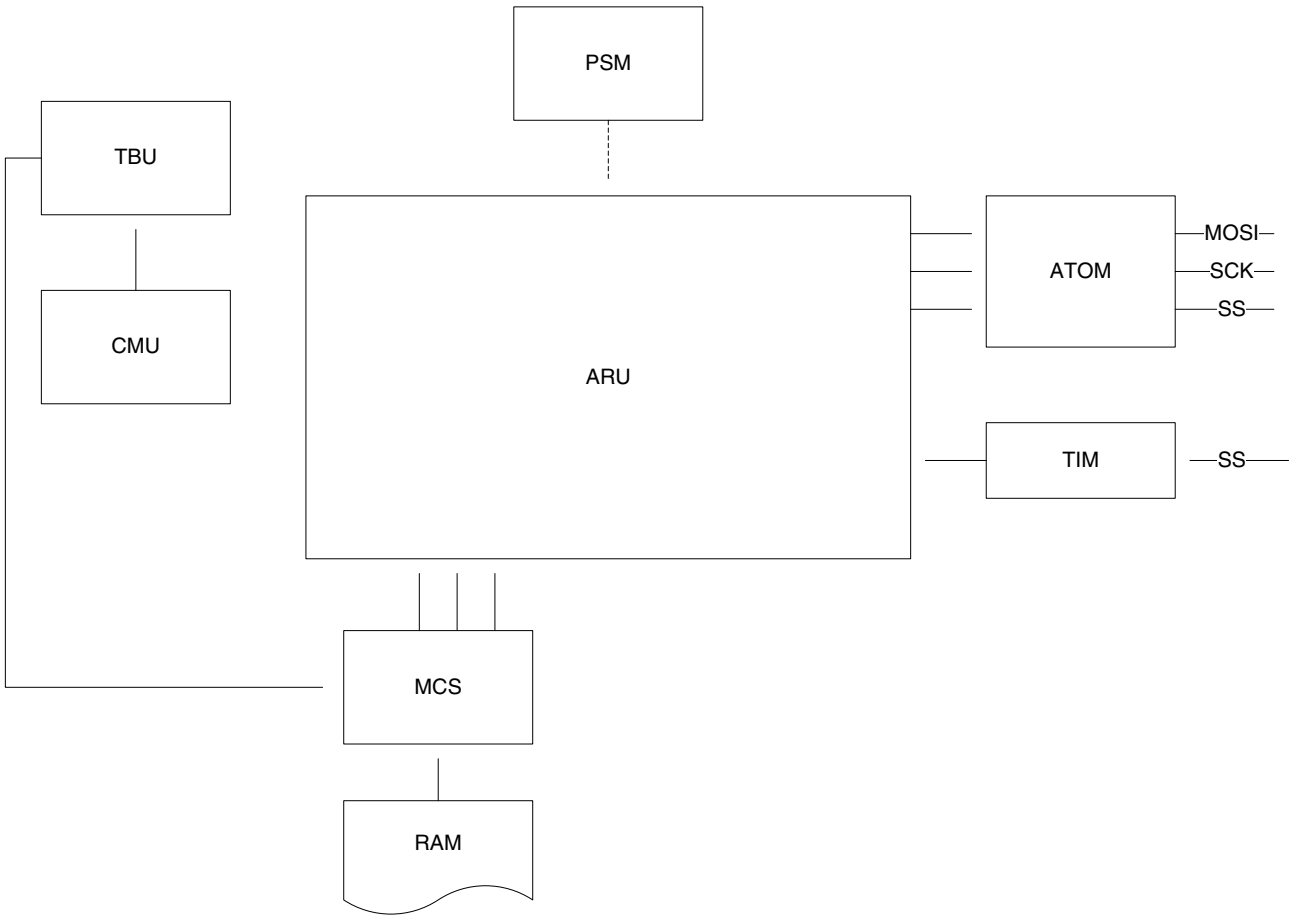

Figure 2. GTM configuration for SPI

Table 2. Submodule functionality

Submodule	Purpose	Use case
Clock Management Unit (CMU)	Generates all of the clocks and counters for the GTM subsystem.	Controls the system clock speed from the chip level clocks.
TimeBase Unit (TBU)	Provides a common timebase that can be used throughout the GTM subsystem.	TBU_TS0 uses CMU_CLK0 as source for the GTM global timebase.
Multi Channel Sequencer (MCS)	A generic data processing module that is connected to the ARU. It allows "programs" to be written to calculate complex output sequences that depend on timebase values.	MCS0 software state machine that controls the data to be driven out of the ATOM channels.
Advanced Routing Unit (ARU)	Provides a mechanism for routing streams of data between data sources	Complex output waveforms for the SS, MOSI, and SCK as instructed by the MCS through the ARU.

Table continues on the next page...

Table 2. Submodule functionality (continued)

Submodule	Purpose	Use case
	and transfer it to a destination. This is the heart of the GTM subsystem.	
ARU connected Timer Output Module (ATOM)	Capable of generating complex output signals through its interconnectivity with the ARU to other modules in the GTM subsystem.	ATOM0 channel 0 (MOSI), channel 1 (SCK) and channel 2 (SS) in SOMS and SOMI modes reading values from MCS0.
Timer Input Module (TIM)	Captures input signals.	Raises an interrupt request on each rising clock edge (SCK) which captures the state of the MISO pin.

Figure 2 also shows the Parameter Storage Module (PSM) which is used to bring in the MISO data to the GTM from the IO processor of the MPC5777M device, as generated in the interrupt request. Another PSM channel could be used to bring in the data to be transmitted by MOSI.

4 Chip level software description

The configuration of the chip modes and clocks, and the GTM at a basic initialization is as described in AN4351 available at freescale.com. The specific configuration of the GTM submodules (TBU, TIM, FIFO, ATOM, and programming of MCS RAM) is shown in Core code to initialize the GTM for SPI.

The MCS array, `MCS0_MEM`, contains both the data and the software for the SPI bus emulation. MCS0 channel 0 and channel 1 are used for the calculations. However, the channels write to three ARU ports for the SS, MOSI, and SCK commands to be consumed by three ATOM channels. To start the SPI bus output after initialization, the MPC5777M core and the MCS do a handshake with the MCS's trigger mechanism as shown below. The MCS's half of the handshake can be seen in MCS software description describing the MCS assembly program operation.

```
/* Start the MCS Program */
GTM_MCS_0.CH0_CTRL.R = 0x00000001;// Enable Channel 0 of MCS0
GTM_MCS_0.CH1_CTRL.R = 0x00000001;// Enable Channel 1 of MCS0

/*Check that the ATOM channels are ready, STRG is set */
while ((GTM_MCS_0.STRG.R & 0x4) == 0);

/*Next Trigger for MCS to signal "Port config finished"  */
GTM_MCS_0.STRG.R = 0x00000001;
```

When this handshake is complete, the MCS is running in an infinite loop.

4.1 ATOM operating as SS

ATOM0 CH2 is the SS output. The Slave Select is an active low output from the master. A low level on SS activates the connected slave.

The ATOM is configured in Signal Output Mode Immediate (SOMI) where an ATOM channel generates an output signal immediately after an update of bit zero of the ATOM[i]_CHn_STAT[ACBI] field when the ARU is enabled.

The MCS channel controls the SS ATOM channel by either sending a High or Low command to the ATOM channel. As the SS is active low the ATOM channel is configured with ATOM[i]_CHn_CTRL[SL] as zero. So that, if the channel is disabled or the output is disabled, the output is set to inverse value of SL, high.

4.2 ATOM operating as MOSI and SCK

Both the MOSI and SCK SPI signals are controlled by ATOM channels running in Signal Output Mode Serial (SOMS) mode.

In this mode the channel acts as a serial output shift register where the content of the CM1 register is shifted to the output whenever the CM1 register is triggered by the configured CMU_CLKn input clock signal. The shift direction is configurable by writing to bit zero of the ATOM[i]_CHn_STAT[ACBI] field when the ARU connection is enabled. The CCU0 runs in counter/compare mode and counts the number of bits that have been shifted. The total number of bits that are to be shifted is defined by the value in the CM0 register.

Table 3. MCS values delivered to ATOM by ARU

SPI signal	Data -> CM1	Shift direction -> ACBI	Shift number -> CM0
SCK	0x005555	Right	Minimum 16
MOSI	For example, 0x0000A5	Right	Minimum 8

4.2.1 Jitter observed on MOSI

The MOSI output is moving by one CMU_CLK6 period relative to the SCK output. This is because the ARU is operating in a round robin path and the order in which the ATOM information is received at channel 0 and channel 1 is not fixed.

Depending upon when the ARU services the data streams on ATOM channels 0 and 1 there will be a worst case scenario of 113 system clock between the channels. As the channels are running at an 80th and a 160th of the system clock frequency this has only a small impact to the data reception at the ATOM channel.

The location of the ARUs 'eyes' is not visible to the GTM, or the chip, so there is no way to fix this delivery order. In SOMS mode the neighboring channels cannot trigger each other.

The clock speeds are slow enough such that the data is still received correctly even with the MOSI jitter with respect to the SCK. Placing the MCS is accelerated scheduling mode helps to minimize the delays between the two channels providing new data to the ARU for delivery to the ATOM channels.

4.3 TIM and input port operating as MISO

The MISO signal is captured on a GPIO input port on each clock rising edge. The clock rising edge is captured by a TIM channel in TIM Input Event Mode (TIEM). The TIM[i]_NEWVALn_IRQ interrupt is enabled and this interrupt captures the input value on the input pin (GPDI[35] in the example code).

```
void IRQ_GTM_TIM0_CH0(void)
{
 extern int MISO, i;

 i++; // increment counter
 MISO = (MISO<<1)+ SIUL2.GPDI[35].B.PDI; // Capture next bit of MISO
 GTM_TIM_0.CH0_IRQ_NOTIFY.R = 0x0000003F; // Clear all interrupts
 if (i==8){ //when 8 bits are received
  GTM_AFD_0.CH[0].BUF_ACC.R = MISO; // place MISO byte in GTM FIFO
  MISO = 0; // clear MISO
  i=0; // reset counter
 }
}
```

The interrupt captures the pin state at the time of the SCK event and appends it on to the previous level from the last clock edge. When eight clocks have been received the interrupt sends the MISO data back to the GTM through the PSM mechanism which can be seen at the address 0xFFD19000.

NOTE

This example does not monitor the SS signal for MISO reception. To add this validity check to the MISO functionality the interrupt could check at pin level too and flag an error, if it were found to be high.

5 MCS software description

The MCS's program and data must be written and pre-compiled before loading in to the MCS RAM block.

As described in the "Example 7: Writing, Compiling, and Programming MCS Code" section of the previously mentioned application note, AN4351 available at freescale.com, the structure of the assembly code includes some definitions, initialization of start addresses for each active channel, and initialization of data and stacks, followed by the subroutines themselves.

Figure 3 and Table 4 describe the general functionality of the MCS assembly code.

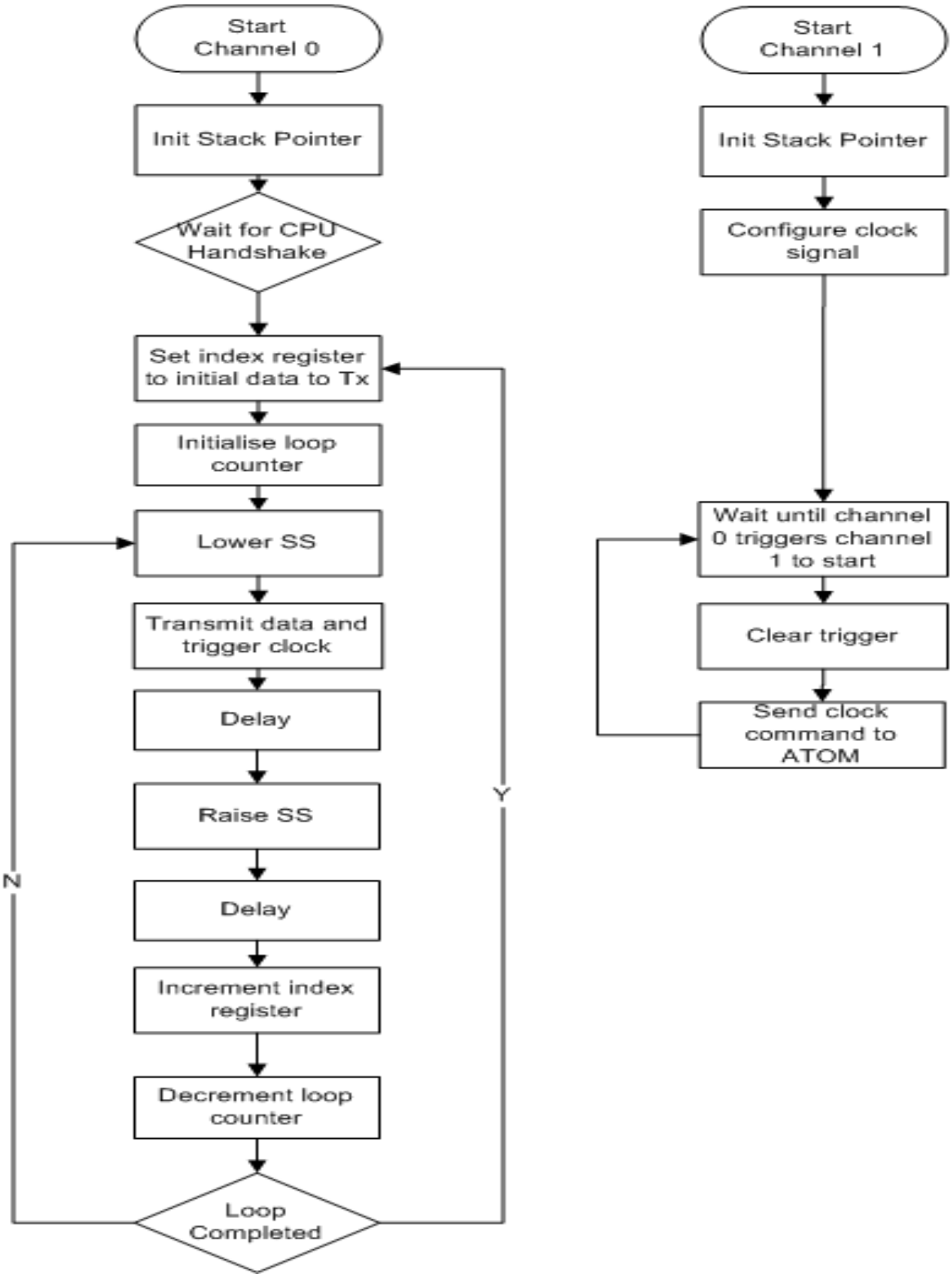

Figure 3. MCS code flow chart

Table 4. SPI bus emulation code blocks description for MCS channel 0

Step	Operation	Description	Code example / Section for further details
1	Start	After reset the MCS channel program counter is at address 0 and must be moved to the start of the code.	JMP tsk0_init
2 (**tsk0_init**)	Init stack pointer	Initialize the stack pointer to the start of the reserved memory space.	MOVL R7 0x000020
3	Wait for CPU handshake	Handshake with the chip core to ensure the system is fully initialized.	Handshake with CPU
4 (**start_tx**)	Load message (index register)	Direct the MCS index register, R6, to the message to be transmitted.	MOVL R6 message_array
5	Initialize loop counter	Set the loop counter such that all the messages in the message array are sent and then repeated.	MRD R1 tsk0_counter
6 (**next_message**)	Lower SS	Transition SS (falling edge) on ATOM0 CH3.	send Start bit
7	Transmit data and trigger clock	Send the 8 bit data of the memory address pointed by index register.	Send message
8	Delay routine	Pause between data transmissions.	MCS delay routine
9	Raise SS	Transition SS (rising edge) on ATOM0 CH3.	send Stop bit
10	Delay routine	Pause between data transmissions.	MCS delay routine
11	Increment index register	Move the index register to the next message to be transmitted.	ADDL R6 0x000004
12	Decrement loop counter	Adjust the loop counter for the completed message.	SUBL R1 0x000001
13	Loop Completed?	Test R1 for zero.	JBC STA Z **next_message**
14	End	In this example the message array repeats from the start.	JMP **start_tx**

Table 5. SPI clock emulation code blocks description for MCS channel 1

Step	Operation	Description	Code example / Section for further details
1 (**tsk1_init**)	Start	After reset the MCS channel program counter is at address 4 and must be moved to the start of the code.	JMP tsk1_init
2	Init stack pointer	Initialize the stack pointer to the start of the reserved memory space.	MOVL R7 0x000024
3	Configure clock signal and send	Set up the clock toggles, shift direct and number of shifts.	Send message

The routines can be configured to use different ARU ports and ATOM channels, to send different commands, and to point to different message address spaces by altering the definitions at the start of the assembly file. This examples uses the configuration listed below.

```
.set ARU_PORT0, 0x0000
.set ARU_PORT1, 0x0001
.set ARU_PORT2, 0x0002

.set PIN_HI, 0x000009
.set PIN_LO, 0x00000A

.set ATOM0_CH0, 0x011F
.set ATOM0_CH1, 0x0120
.set ATOM0_CH2, 0x0121

.set message_array, 0x70
```

There are also variables that are set up in the assembly code and referred to through the routine for the number of messages to be transmitted and the length of the delay between messages as shown below.

```
tsk0_counter: .lit24 68  # number of messages to transmit
tsk0_delay:   .lit24 1   # length of delay between messages
```

The full assembly code in the Hightec™ format is provided in Assembly code for SPI example. To modify the assembly code for the CASPR-MCS assembler, refer to AN4351 available at freescale.com.

5.1 Handshake with CPU

To ensure that both the CPU and the GTM are in the initialized state and ready to start the SPI communication, a handshake routine can be used. Both the CPU and the GTM MCS have access to the STRG and CTRG registers inside the MCS memory map.

The description of the assembly routine that runs inside of the GTM is described below in Table 6.

Table 6. Handshake with CPU

Step	Operation	Description	Code snippet
1	Set the Channel 2 trigger	Set the trigger bit to indicate the routine has started to the CPU	MOVL STRG 0x000004
2	Load R0 with 1	—	MOVL R0 0x000001
3	Wait until bit 0 of the STRG register is same as R0	Wait until the CPU signals back that the handshake was seen	WURM R0 STRG 0x0001
4	Load R3 with the current timebase value	—	MOV R3 TBU_TS0
5	Load R0 with 25,000	—	MOVL R0 0x0061A8
6	Add R0 and R3 in R0	—	ADD R0 R3
7	Wait until TS0 == R0	—	WURM R0 TBU_TS0 0xFFFF
8	Clear the triggers	—	MOVL CTRG 0x000003

The code below is the CPU's side of the handshake code.

```
/*Check that the ATOM channels are ready, STRG is set */
while((GTM_MCS_0.STRG.R & 0x4) == 0);

/*Next Trigger for MCS to signal "Port config finished"  */
GTM_MCS_0.STRG.R = 0x00000001;
```

5.2 Lower SS signal

Before any synchronous message can be transmitted on the SPI bus, the start condition needs to be issued on the bus which is a falling edge on the SS pin to activate the slave.

Table 7. Transition SS for communication to commence

Step	Operation	Description	Parameters	Code snippet
1	Load TBU timestamp	Read the current value of the TBU timestamp in to the MCS register, R3.	n/a	MOV R3 TBU_TS0
2	Load R2 a small value	The SS transition is controlled using the value stored at address 0x68.	n/a	MRD R2 64
3	Add R2 and R3 in R2	Set the match value at 0x68 from the current timestamp.	n/a	ADD R3 R2
4	Configure the ACB for SS	Set on match event (Compare in CCU0 only, use timebase TBU_TS0). The ATOM channel for SS is configures as active low.	ACB = 0x09	MOVL ACB PIN_HI
5	Place the data for the ATOM channel associated with SS into the ARU port	Move R3 to the ARU port.	ARU Read port = 0x0002	AWR R3 R3 ARU_PORT2
6	Return from subprogram	The program counter PC is loaded with current value on the top of the stack.	n/a	RET

5.3 Send message

The messages in this example are stored in the MCS RAM (copied in by the core at the same time as the MCS code is moved in to the MCS RAM), before the SPI routines are started.

Table 8. Send data byte

Step	Operation	Description	Parameters	Code snippet
1	Move index register to R5	Load the data to be transmitted in to R5	n/a	MRDI R5 R6
2	Load R3 with number of shifts the ATOM is to perform	R3 is loaded with number of clock transitions required to clock out the MOSI data.	All 24 bits.	MOVL R3 0x000017

Table continues on the next page...

Table 8. Send data byte (continued)

Step	Operation	Description	Parameters	Code snippet
3	Load ACB with the shift direction	Shift the message left so that the next shift right (step 6) accesses the correct bit each loop rotation.	Shift right (0)	MOVL ACB 0
4	Initiate channel 1 (SCK)	Set bit 1 of STRG register	n/a	MOVL STRG 0x000002
5	Write data and shift value to the ARU port	Move R3 and R5 to the ARU port	ARU read port = 0	AWR R3 R5 ARU_PORT0
6	Return from function	The program counter PC is loaded with the current value of the top of the stack	n/a	RET

Table 9. Send clock

Step	Operation	Description	Parameters	Code snippet
1	Move clock signal to R5	Load the clock signal to be transmitted in to R5	n/a	MOVL R5 0x00AAAA
2	Load R3 with number of shifts the ATOM is to perform	R3 is loaded with number of clock transitions required to clock out the MOSI data.	All 24 bits.	MOVL R3 0x000017
3	Load ACB with the shift direction	Shift the message left so that the next shift right (step 6) accesses the correct bit each loop rotation.	Shift right (0)	MOVL ACB 0
4 (**loop**)	Wait for trigger	Wait for bit 1 of STRG register to be set by channel 0	n/a	MOVL R0 0x000002 WURM R0 STRG 0x00002
5	Clear the trigger	Write 0x2 to the CTRG register	n/a	MOVL CTRG 0x000002
6	Write data and shift value to the ARU port	Move R3 and R5 to the ARU port	ARU read port = 1	AWR R3 R5 ARU_PORT1
7	Return from function	The program counter PC is loaded with the current value of the top of the stack	n/a	JMP loop

NOTE

Steps 5 and 6 in Table 8 and steps 5 to 7 in Table 9 operate in parallel on separate set of local MCS channel registers.

5.4 Raise SS signal

To complete transmission of the message, the stop condition needs to be issued on the bus which is a positive edge on the active low SS pin.

Table 10. Transition SS for communication to conclude

Step	Operation	Description	Parameters	Code snippet
1	Load TBU timestamp	Read the current value of the TBU timestamp in to the MCS register, R3.	n/a	MOV R3 TBU_TS0
3	Add R2 and R3 in R2	Set the match value at 0x68 from the current timestamp.	n/a	ADD R3 R2
4	Configure the ACB for SS	Set on match event (Compare in CCU0 only, use timebase TBU_TS0). The ATOM channel for SS is configures as active low.	ACB = 0x0A	MOVL ACB PIN_LO
5	Place the data for the ATOM channel associated with SS into the ARU port	Move R3 to the ARU port.	ARU Read port = 0x0002	AWR R3 R3 ARU_PORT2
6	Return from subprogram	The program counter PC is loaded with current value on the top of the stack.	n/a	RET

5.5 MCS delay routine

A delay routine is a useful code snippet to have for any software development. In this example, a delay is used to create a time space between messages.

The MCS has direct access to the TBU timestamp counter and also has a "wait until register match" instruction, WURM, which can be used to hold the MCS program counter for a predetermined amount of time or to wait until a trigger event from another channel occurs. WURM suspends the MCS channel until the two registers (with a bit mask) match.

```
WURM A B C

Wait until A = (B & C)
```

A commonly used delay routine often involves a variable "duration" that is decremented in a loop, until it is zero. Within that loop, a known finite time can be included by using a wait operation.

In the example given in this application note, the "duration" variable is stored in the MCS RAM with other data such as the message loop counter at address 0x68 (0x1 in this particular case).

The timebase value is read and the match value is set at 2,500 clocks after "now." If the GTM TBU is running from an 80 MHz clock, the delay is 31.25 µs around each loop.

```
delay:
    MOV     R3    TBU_TS0           # Load timestamp to R3
    MRD     R4    tsk0_delay        # Load loop counter to R4
    ATUL    R4    0x000000          # Is R4 Zero?
    JBS     STA   Z  exit           # If R4 is Zero jump to exit
    MOVL    R0    0x0009C4          # Load R0 with 2,500
  continue: ADD     R3    R0        # Add R0 to the Timebase saved in R3
```

```
    WURM    R3    TBU_TS0  0xFFFF     # Wait until the timebase matches R3
    SUBL    R4    0x000001            # Decrement the loop counter in R4
    JBC     STA   Z  continue         # If not zero continue
  exit: RET                           # Exit subroutine
```

6 Core code to initialize the GTM for SPI

The C code below configures the TBU, the ATOM channels, loads the MCS software and message array into MCS RAM memory, sets up the MISO input pin, TIM channel and FIFO channel, and the handshake with the MCS itself.

The MCS software is compiled in to a binary file and loaded into the MCS RAM as described in AN4351 available at freescale.com. The MCS message array is given as an array of integers and copied into RAM at the specified location so that it is easier to manipulate without the need to reassemble the MCS software each time.

```
// ARU Write Addresses from GTM104 Specification
#define MCS0_WRADDR0 0x077
#define MCS0_WRADDR1 0x078
#define MCS0_WRADDR2 0x079

unsigned int * dest, src;
extern int __MCS0_ADDR; /* Label of location of the raw data set in the linker */

void memcpy_swap_word(unsigned int *, unsigned int *, signed int);

void SPI()
{
int i;
gtm_ptr p;

// Configure TBU
GTM_TBU.CH0_CTRL.R = 0x00000000;// Select CMU_CLK0
GTM_TBU.CHEN.R = 0x00000002;// Switch on TBU0

/*****************************************************************************
* ATOM0_CH2 = SS
* ATOM0_CH1 = SCK
* ATOM0_CH0 = MOSI
* GPIO PC3 = MISO & connect SCK to TIM0_CH0 = PF1
*****************************************************************************/

/* Program MCS. First check whether the RAM RESET is complete.
WAIT until RAM_RST == 0, wait RAM Reset after startup. */
while(GTM_MCS_0.CTRL.R == 0x00010000);

GTM_ATOM_0.CH2_RDADDR.R = MCS0_WRADDR2;// used for reading
GTM_ATOM_0.CH1_RDADDR.R = MCS0_WRADDR1;// used for reading
GTM_ATOM_0.CH0_RDADDR.R = MCS0_WRADDR0;  // used for reading
GTM_ATOM_0.CH2_CTRL.R = 0x00000008;// SOMI, ARU_EN=1, SL=0
GTM_ATOM_0.CH1_CTRL.R = 0x0400500B;// SOMS, CLK5, ARU_EN=1, OSM=1, UPEN_CTRL=1, ACB0=0, SL=0
GTM_ATOM_0.CH0_CTRL.R = 0x0400600B;// SOMS, CLK6, ARU_EN=1, OSM=1, UPEN_CTRL=1, ACB0=0, SL=0

/*ATOM0_CH0-2 switch on*/
GTM_ATOM_0.AGC_OUTEN_CTRL.R = 0x0000002A;
GTM_ATOM_0.AGC_ENDIS_CTRL.R = 0x0000002A;

GTM_ATOM_0.AGC_FUPD_CTRL.R = 0x0000002A;
GTM_ATOM_0.AGC_INT_TRIG.R = 0x00000015;

GTM_ATOM_0.AGC_GLB_CTRL.R = 0x002A0001;// Host Trigger to start ATOM

// load raw bin data in to MCS0 RAM = 0xFFD38000
dest = (int)&MCS0_MEM; /* CPU view of the address of the MCS memory space */
src = (int)&__MCS0_ADDR; /* Label of location of the raw data set in the linker */
memcpy_swap_word(dest, src, 270);
```

Core code to initialize the GTM for SPI

```
p = &MCS0_MEM + 0x1C;
for(i=0;i<=67;i++)
{
/* Copying the content of the array mcs0spi_messages[i] into MCS0 RAM0 */
p[i]=mcs0SPI_messages[i];
}

/* Use PC[3] as an input pin */
SIUL2.MSCR_IO[35].B.SSS = 0;
SIUL2.MSCR_IO[35].B.ODC = 0;
SIUL2.MSCR_IO[35].B.IBE = 1;

/* Configure the TIM for MISO reception */
/* Channel 0 captures the edge event of the clock */
GTM_TIM_0.CH0_CTRL.R = 0x00002c05; //configure TIM0 CH0 to interrupt on every rising edge
while(GTM_TIM_0.CH0_CTRL.R!=0x00002c05); //confirm the configuration is effective
GTM_TIM_0.CH0_IRQ_EN.R = 1; //NEWVAL_IRQ Enabled

/* Configure FIFO0 CH0 */
GTM_FIFO_0.CHANNEL[0].CTRL.R = 0x0000000D; // RAM write unlocked, FIFO flushed and Ring
Buffer Mode
GTM_FIFO_0.CHANNEL[0].IRQ_EN.R = 0x00000002; //Enable Full interrupt

/* Start the MCS Program */
GTM_MCS_0.CH0_CTRL.R = 0x00000001;// Enable Channel 0 of MCS module 0
GTM_MCS_0.CH1_CTRL.R = 0x00000001;// Enable Channel 1 of MCS module 0

/*Check if the Channel program is ready and MCS_STRG is set, then start configure the Ports.
WAIT until MCS0 STRG == 0x00000004
MCS --> ATOM Output finished when MCS0_STRG == h#00000004 */
while((GTM_MCS_0.STRG.R & 0x4) == 0); // MCS0_STRG != 4

/*Next Trigger for MCS to signalize "Port config finished"  */
GTM_MCS_0.STRG.R = 0x00000001; // Port configuration finished, MCS running
/*Now the MCS is running in a infinite loop. */

}/*END of function SPI()*/

void memcpy_swap_word(unsigned int * dst, unsigned int * src, signed int size)
{
while (size-- > 0)
{
*dst++ = SWAPW(*src);
src++;
}
}
```

Below is the SPI message array used in this example.

```
int mcs0spi_messages[68] = {0x0000005a, 0x00000001, 0x00000002, 0x00000004, 0x00000008,
0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000040, 0x00000020, 0x00000010,
0x00000008, 0x00000004, 0x00000002, 0x00000001, 0x00000055, 0x000000aa, 0x000000a5,
0x0000005a, 0x00000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010, 0x00000020,
0x00000040, 0x00000080, 0x00000040, 0x00000020, 0x00000010, 0x00000008, 0x00000004,
0x00000002, 0x00000001, 0x00000055, 0x000000aa, 0x000000a5, 0x0000005a, 0x00000001,
0x00000002, 0x00000004, 0x00000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080,
0x00000040, 0x00000020, 0x00000010, 0x00000008, 0x00000004, 0x00000002, 0x00000001,
0x00000055, 0x000000aa, 0x000000a5, 0x0000005a, 0x00000001, 0x00000002, 0x00000004,
0x00000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000040, 0x00000020};
```

The Hightec assembler generates the binary in the little endian, whereas the MPC57xx is big endian. The endianness can be swapped using the following macro.

```
#define SWAPW(w) \
(((w & 0xff) << 24) | ((w & 0xff00) << 8) \
| ((w & 0xff0000) >> 8) | ((w & 0xff000000) >> 24)) /* change endianness */
```

The MOSI and SCK ATOM ports use a slower clock source which means that the CMU clocks used (CLK_5 and CLK_6) should be set for larger dividers inside the GTM initialization function.

```
GTM_CMU.CLK_CTRL[5].R = 0x4F;    // define CMU_GCLK_EN/80 clock
GTM_CMU.CLK_6_CTRL.R = 0x9F;     // define CMU_GCLK_EN/160 clock
```

7 Assembly code for SPI example

```
#==========================================================================
# Project Name    : AN 4864
# Company         : Freescale
# Author          : Inga Harris
#==========================================================================
.section .mcs.text,"axw",@progbits
.include "mcs.inc"
.set memid, 0
.set memsize, 0x1800


# Define the values of the symbols used
.set ARU_PORT0, 0x0000      # MCS ARU port number 0
.set ARU_PORT1, 0x0001      # MCS ARU port number 1
.set ARU_PORT1, 0x0002      # MCS ARU port number 2
.set PIN_HI, 0x000009       # ACB = 0x09 set high when compare in CCU0 with TBU_TS0
.set PIN_LO, 0x00000A       # ACB = 0x0A clear high when compare in CCU0 with TBU_TS0
.set ATOM0_CH0, 0x011F      # ATOM0_CH0 ARU write address
.set ATOM0_CH1, 0x0120      # ATOM0_CH1 ARU write address
.set ATOM0_CH2, 0x0121      # ATOM0_CH2 ARU write address
.set message_array, 0x70    # offset address of the SPI messages


# initialize reset vectors of different tasks
# -------------------------------------------
.org 0x0
jmp tsk0_init
jmp tsk1_init


# allocate stack frames ( each task has 16 memory locations )
# -----------------------------------------------------------
.org 0x20
tsk0_stack:.lit24 0
tsk1_stack:.lit24 0


# allocate and initialize memory variables
# ----------------------------------------
.org 0x64
tsk0_counter: .lit24 68     # number of messages to transmit
tsk0_delay:   .lit24 1      # length of delay between messages


#*************************************************************
#   tsk0: SPI master
#*************************************************************
.org 0x180
tsk0_init:
movl    R7,    0x000020              # Init stack pointer
movl    STRG,  0x000004              # Set channel 2 trigger
movl    R0,    0x000001              # Load R0 with 1
wurm    R0,    STRG, 0x0001          # Wait until channel 0 trigger is set by core
mov     R3,    TBU_TS0               # Load the current timestamp
movl    R0,    0x0061A8              # Set R0 to 25,000
add     R0,    R3                    # Add R0 and R3
wurm    R0,    TBU_TS0,  0xFFFF      # Wait until the timstamp reaches that value
movl    CTRG,  0x000007              # Clear the triggers
start_tx: movl   R6,   message_array # Initialize index register
mrd     R1,    tsk0_counter          # Initialize loop counter
```

Assembly code for SPI example

```
next_message: call   ss_low          # Lower the SS pin
call   byte_tx                       # Send the data and clock
call   delay                         # Wait tsk0_delay * 2,500 clocks
call   ss_high                       # Raise the SS pin
call   delay                         # Wait tsk0_delay * 2,500 clocks
addl   R6,   0x000004                # Increment index register
subl   R1,   0x000001                # Decrement loop counter
jbc    STA,  Z,  next_message        # Is the loop counter zero? No = next_message
jmp    start_tx                      # Loop ended. Start from beginning


#************************************************************
#   ss_low
#************************************************************
ss_low:
mov    R3,   TBU_TS0                 # Reload the current timestamp
mrd    R2,   64                      # Load the value from tsk0_counter
add    R3,   R2                      # Add tsk0_counter to the timestamp
movl   ACB,  PIN_HI                  # Set ACB value
awr    R3,   R3,   ARU_PORT2         # Send data, shift counter and ACB to ARU
ret                                  # Return from subroutine


#************************************************************
#   byte_tx
#************************************************************
byte_tx:
mrdi   R5,   R6                      # Set data to be sent
movl   R3,   0x000017                # Set number of bits to shift
movl   ACB,  0                       # Set shift direction
movl   STRG, 0x000002                # Set channel 1 trigger to initiate clock
awr    R3,   R5,   ARU_PORT0         # Send data, shift counter and ACB to ARU
ret                                  # Return from subroutine


#************************************************************
#   ss_high
#************************************************************
ss_high:
mov    R3,   TBU_TS0                 # Reload the current timestamp
add    R3,   R2                      # Add tsk0_counter to the timestamp
movl   ACB,  PIN_LO                  # Set ACB value
awr    R3,   R3,   ARU_PORT2         # Send data, shift counter and ACB to ARU
ret


#************************************************************
#   delay
#************************************************************
delay:
mov    R3,   TBU_TS0                 # Load timestamp
mrd    R4,   tsk0_delay              # Load loop counter
atul   R4,   0x000000                # Is it zero?
jbs    STA,  Z,  exit                # If zero exit subroutine
movl   R0,   0x0009c4                # Load R0 with 2,500
continue: add    R3,   R0            # Add 2,500 to timestamp
wurm   R3,   TBU_TS0,  0xFFFF       # Wait until timebase matches R3
subl   R4,   0x000001                # Decrement loop counter
jbc    STA,  Z,  continue            # If not zero jump to continue
exit: ret                            # return from subroutine


#************************************************************
#   tsk1: SCK
#************************************************************
.org 0x250
tsk1_init:
movl   R7,   0x000024                # Init stack pointer
movl   R0,   0x000002                # Load R0 with 2
movl   R5,   0x00AAAA                # Set data to be sent
```

```
movl   R3,   0x000017              # Set number of bits to shift
movl   ACB,  0                     # Set shift direction
loop: wurm   R0,   STRG, 0x00002   # Wait until channel 0 triggers the clock
movl   CTRG, 0x000002              # Clear the trigger
awr    R3,   R5,   ARU_PORT1       # Send data, shift counter and ACB to ARU
jmp    loop                        # Loop ended. Wait for next trigger
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4864
Revision 1, 04/2014