Freescale Semiconductor
Application Note

Document Number: AN4238
Rev. 0, Jan 2011

Using the DSPI Automatic Serial PWM (ASP) Generation Feature

Hardware support for daisy-chaining external lamp drivers

by: Stefan Lüllmann
Field Application Engineer
EMEA
Germany

Contents

1 Introduction

The aim of this document is to give the reader an overview of the new serialization feature of the DSPI module first implemented on Freescale's MPC564xB device. It is presumed that the reader has a basic understanding of the DSPI functionality in general, because this document will focus only on the details of this specific enhancement.

The automatic serial pulse width modulation (ASP) generation feature can be used to facilitate the connection of up to 4 × 8-bit external lamp drivers in daisy chain connection (or up to 8 × 8-bit drivers in case two DSPI modules are connected in series). This document will explain the reasons for the implementation of this feature, and show how it can be used to minimize software involvement and save pins for this typical body application scenario. The example code provided will supplement the description and give the reader a demo implementation for the different modes to facilitate the start.

After reading this document, the reader should have an understanding of how the ASP feature can be used to reduce the software overhead and save pins, and have an overview of the modules and registers involved.

As serialization/de-serialization in general is not new and was already implemented on MPC55xx devices, the ASP feature can be understood as an extension to the already existing DSPI module. Hence, this application note builds on the basic functionality already described in Freescale application notes AN2865 and AN2867 (available at www.freescale.com) and

freescale semiconductor

will focus only on the specific use case of pulse width modulation (PWM) generation through SPI for externally daisy chained devices. The standard DSPI modes and functionality will not be discussed.

1.1 What ASP is and why it is needed

In the body space PWM generation for lamp control is a key requirement. Basically, there are three different ways to create PWM on a microcontroller:

- By hardware through dedicated pins (in other words, using a timer module)
- By software through GPIOs
- Serialized through SPI, to save pins

Although the MPC560xB/C devices offer up to 64 hardware PWM channels with support for autonomous ADC read-back, it is typical for body applications to be cost-driven and have a limited number of I/O pins on the device. Therefore, a mixture of SPI serialization and hardware PWM generation is often implemented in the ECU to save pins (and ultimately cost) by either saving external multiplexers or being able to stay longer on a lower pin-count package.

For the SPI serialization solution, external lamp drivers are frequently connected in a daisy-chain structure. The length of the daisy chain may vary, but typically two daisy chains for left and right lighting are common. Depending on the SPI interface bit-width of the external drivers, a resulting 16- to 32-bit SPI frame is not unusual.

However, taking a deeper look into the PWM requirements, a pure software solution can become complex and demands a high amount of system resources. Some of the details and drawbacks are discussed in the following two example implementations.

A fourth option might be the use of some high-end lamp drivers to generate the PWM internally. But as the body application domain is still cost-driven, Freescale offers this new feature as a hardware-supported cost-saving option, which also offers a competitive advantage for low-end applications.

1.2 Example standard requirements

The two example software implementations discussed in the next two sections are based upon these assumptions.

The use case assumes three external lamp drivers connected to the MCU in a daisy-chain configuration. Each external driver has an 8-bit SPI interface. This results in a 24-bit-wide SPI frame to be sent.

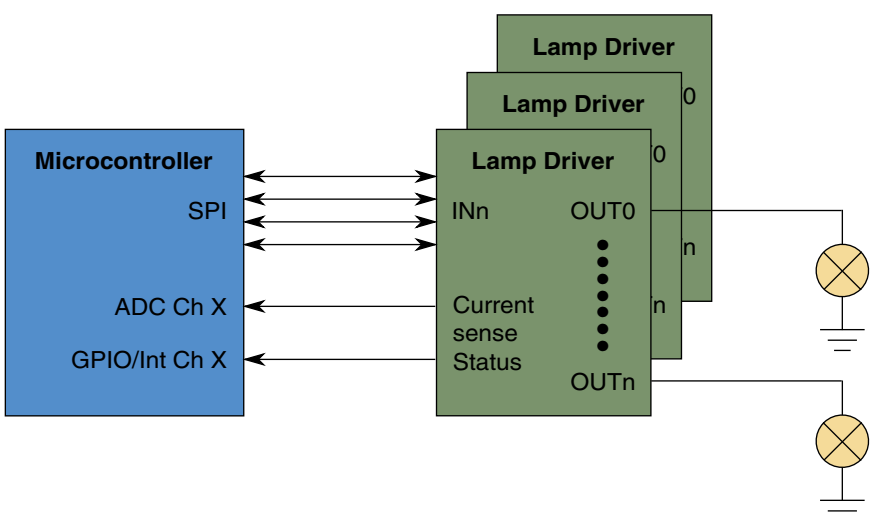


Figure 1. Daisy-chain configuration

For driving bulbs in this setup a typical requirement is to have a 100 Hz PWM with a 0.5% resolution. In other words:

period = 10 ms; resolution = 10 ms × 0.5% = 50 µs

For single core MCUs without DMA this would result in a high interrupt load, because every 50us a change of state, and therefore an interrupt, would be necessary. However, as most of the devices in the MPC560xB/C family have a DMA controller,[1] parts of the software solution can be transferred to it to reduce the interrupt load. But there is still a certain penalty for doing that, which is explained in the next section.

1.3 24-bit software SPI frame example 1

Figure 2 shows the first possibility for driving a 24-bit software-generated SPI frame. The DSPIx.PUSHR register for sending SPI frames is divided into a 16-bit command word in the upper halfword of the register and a 16-bit data portion in the lower halfword of the register. Every 8- or 16-bit write to this register transfers the complete 32-bit DSPIx.PUSHR to the TX FIFO. As the 0.5% resolution requirement demands an update of this register every 50 µs, the usage of the DMA to copy the data to be transferred from a pre-calculated table in RAM to the push register is optimal to reduce the CPU load. To achieve the 50 µs trigger time a periodic interrupt timer (PIT) is used to start the DMA transfer. Because of the fact that the DMA works optimally with data sets of constant size, and to keep the software simple, it is assumed here that the data is stored in memory as a complete set of 32-bit command plus data word. This way it is possible to copy it with only one DMA channel in a single 32-bit write. Although this method only uses one DMA channel it results in a non-optimal use of RAM memory as the same command word is stored multiple times and 8-bits of user data is stored as a 16-bit halfword in RAM. In this example with a table of 300 entries this would result in a RAM overhead of

(300 × 16-bit command word) + (300 × 8-bit unused data) = 7200 bit = 900 byte

If you consider that typically two SPI chains are used for left and right lighting this would add up to ~1.8 KB.

In addtion, one DMA channel would be used for the transmit side and one for the receive side for each SPI chain. For two SPI chains this would add up to four DMA channels. This already consumes up to 25% of the on-chip DMA resources (16 channels for MPC5607B/C) just for this feature.

Obviously, the use of system resources increases with the table length and the number of SPI chains.

1. The MPC5602B/C, MPC5603B/C and MPC5604B/C devices do not have a DMA controller.

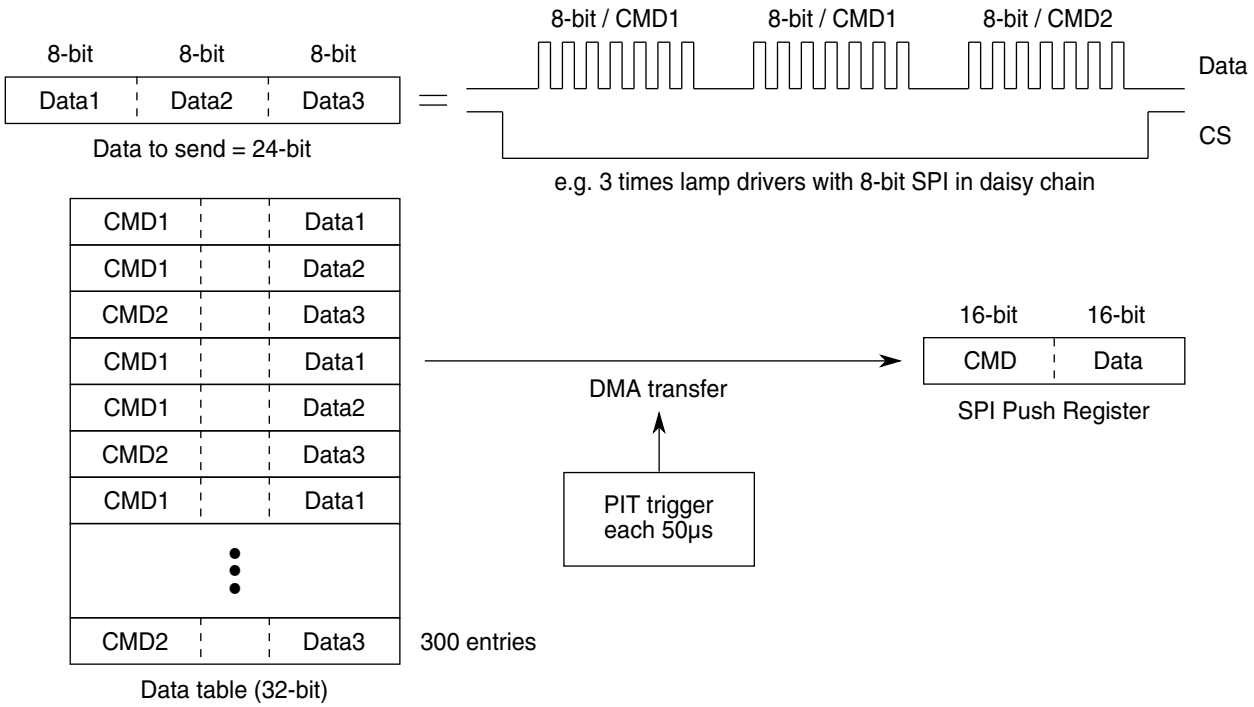


Figure 2. Direct DMA copy to PUSH register

1.4 24-bit software SPI frame example 2

Using the same assumptions as in the previous example, a second possibility for driving the external lamp drivers with a 24-bit software generated SPI frame would be to establish a buffer in RAM for assembling the data for the DSPIx_PUSHR register prior to copying it. This will reduce the amount of data that has to be stored in the RAM table to only user-relevant data. However, the smaller RAM overhead is achieved at the expense of requiring two DMA channels for one transmit. The first DMA channel is again triggered by the PIT each 50 µs. Once the buffer is filled with data, the first DMA channel uses channel linking to trigger the second channel to store the data in the DSPIx.PUSHR register. The RAM overhead for this solution is only the RAM buffer with 3 × 32 bits = 96 bits.

In regards to system resources used, two DMA channels would be used for the transmit side and one for the receive side for each SPI chain. For two SPI chains this would add up to six DMA channels. This already consumes up to 37.5% of the on-chip DMA resources just for this feature.

Obviously, the use of system resources increases with the table length and the number of SPI chains. Additionally, inhomogeneous data (e.g. 8-bit, 8-bit, 16-bit) will increase the RAM waste.

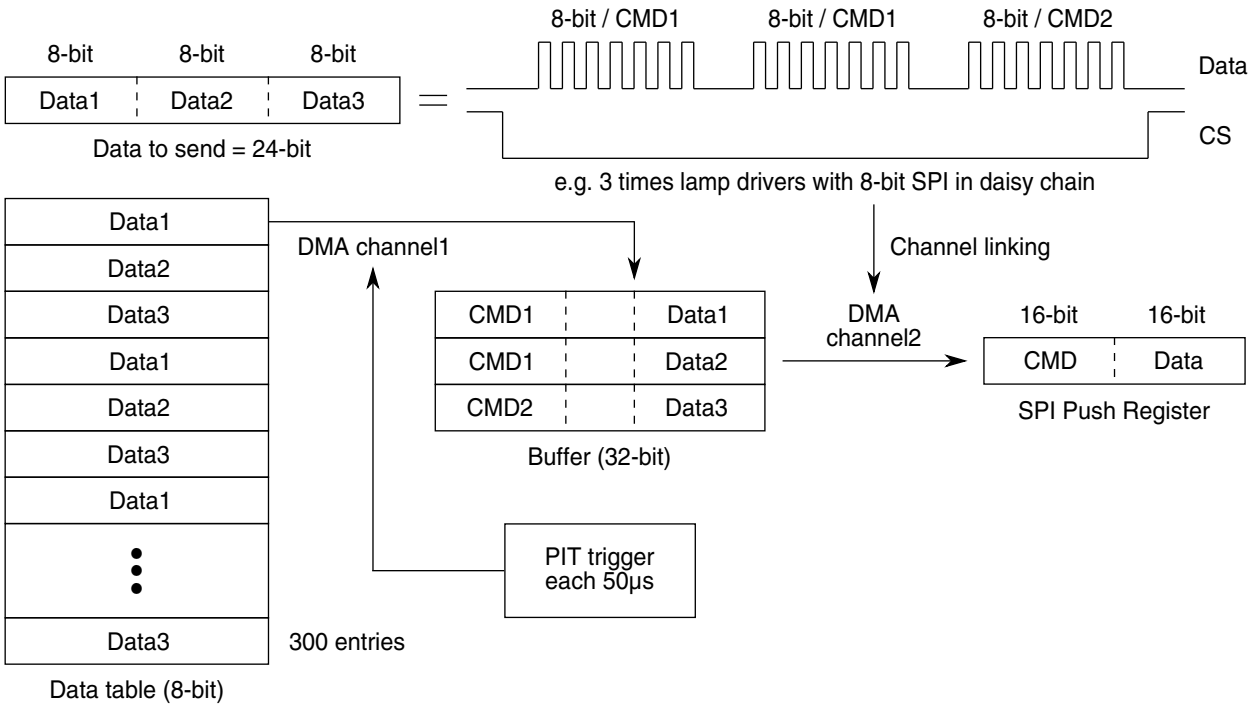


Figure 3. Indirect DMA copy to PUSH register

1.5 Conclusion

The above two examples demonstrate two possibilities for implementing a software solution for SPI-driven PWM for external daisy-chained lamp drivers, as used in today's standard MPC560xB/C devices. But each solution has its advantages and disadvantages in using system resources. Solution one has a higher RAM usage, which dramatically increases with the table length and the number of SPI chains, but uses fewer system resources, in this case DMA channels. Solution two uses significantly less RAM, but needs two to three DMA channels per SPI chain.

By including hardware support for this kind of functionality, the new Freescale devices will offer the same capabilities with considerably less use of system resources. In addition, it will also significantly reduce the software complexity and make maintenance easier. However, backwards compatability is maintained, so the software approach discussed above could still be used.

Another benefit of the new feature is the ability to automatically trigger ADC conversions through the cross-triggering unit (CTU) while doing SPI PWM generation. When using the classical software PWM approach, the eMIOS timer module is not used — therefore the hardware link of the timer module to the ADC through the CTU is not applicable. With the new feature, the input for the SPI frame is now coming from the eMIOS timer module channels. It is possible to use the CTU triggering capability to start hardware-triggered ADC conversion. Therefore, the missing hardware connection is reestablished. The EMIOS can now be used to trigger an ADC conversion through the hardware CTU link, and by using the OPWMT mode the ADC trigger can be configured at an arbitrary position in the PWM period. The generated PWM signals are then output by serializing them through the DSPI pins. Therefore, pins are saved for normal GPIO function.

Please find a schematic overview in Figure 4. The details of this feature are discussed in the next section.

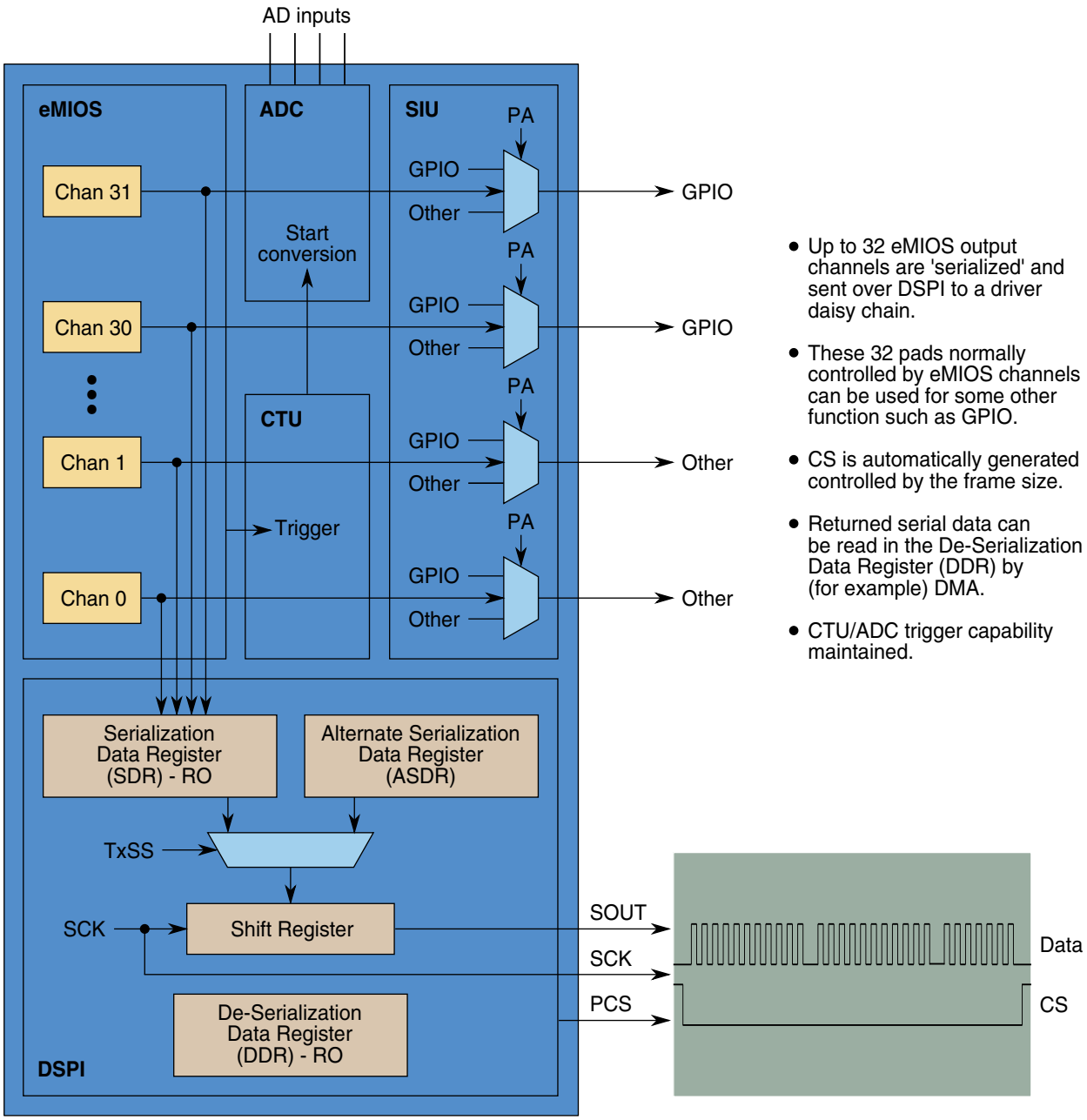


Figure 4. System-level view of ASP

2 Implementation overview

For implementing this specific new enhancement two new modes, DSI and CSI, have been added to the DSPI module. This mode does not exist on previous MPC560xB/C family devices, where only the SPI mode was implemented. However, on MPC55xx devices these modes already exist, so the risk of adding them on the MPC564xB was low.

Please find an overview of these three modes below.

- Serial peripheral interface (SPI) configuration — the DSPI operates as an SPI with support for queues

- Deserial serial interface (DSI) configuration — the DSPI serializes eMIOS output channels and deserializes the received data by placing the data in the DSPIx.DDR register
- Combined serial interface (CSI) configuration — the DSPI operates in both SPI and DSI configurations, interleaving DSI frames with SPI frames, giving priority to SPI frames

In addition to the two new modes and some internal circuitry a set of new registers was added in the DSPI module itself as well as in the SIUL module. Please find an overview below.

Table 1. New DSPI registers

Address	Description
0x00BC	DSPI DSI Configuration Register (DSPI_DSICR)
0x00C0	DSPI DSI Serialization Data Register (DSPI_SDR)
0x00C4	DSPI DSI Alternate Serialization Data Register (DSPI_ASDR)
0x00C8	DSPI DSI Transmit Comparison Register (DSPI_COMPR)
0x00CC	DSPI DSI Deserialization Data Register (DSPI_DDR)
0x00D0	DSPI DSI TSB Configuration Register 1 (DSPI_DSICR1)
0x00D4	DSPI DSI Serialization Source Select Register (DSPI_SSR)
0x00D8–0x00E4	Reserved
0x00E8	DSPI DSI Deserialized Data Interrupt Mask Register (DSPI_DIMR)
0x00EC	DSPI DSI Deserialized Data Polarity Interrupt Register (DSPI_DPIR)

Table 2. New SIUL registers

Address	Description
0x1100–0x113F	Parallel Input Select Register (PISR0—PISR15)
0x1140–0x11FF	Reserved
0x1200	DSPI Input Select Register (DISR)
0x1204–0x3FF	Reserved

More details of how the serialization/deserialization and chaining of DSPI modules works on the predecessor MPC55xx family can be found in Freescale application note AN2867, "Using the DSPI Module on the MPC5500 Family," available at www.freescale.com.

For more detailed information on the DSPI module itself as implemented on MPC564xB, please refer to Freescale document MPC5646BCRM, *MPC5646B/C Reference Manual*.

2.1 DSPI serialization

With the new ASP feature implemented on the MPC564xB it is now possible to create a 4-bit to 32-bit SPI frame with automatic chip select generation in hardware on one DSPI module, with minimal software intervention. By having the input sources highly configurable it is possible to have control and data bits in the same frame. This allows very high flexibility of the data to be sent.

In turn this flexibility allows control of a wide range of SPI-based external devices, requiring different frame configurations. Control bits may come from the DSPIx.ASDR register, set by the CPU, while data bits can be generated by SoC peripheral modules such as PWM timers.

However, this new feature is only implemented on four of the eight DSPI modules, DSPI0 to DSPI3.

To reduce the amount of data sent over the DSPI and therefore reduce the emissions on the PCB (presuming that the two daisy chains represent long traces on the board), the DSPI has four different conditions that can initiate a transfer:

- Continuous — data transmitted continuously
- Change in data — any change in data to be transmitted causes a new SPI frame to be transmitted
- Edge-sensitive hardware trigger — triggered by a slave DSPI connected via chaining

Table 3. DSI data transfer initiation control

DSPI_DSICR bits		Transfer initiation control
TRRE	CID	
0	0	Continuous
0	1	Change in data
1	0	Triggered
1	1	Triggered or change in data

For creating the frame to be transmitted, the data can be generated either by software (DSPIx.ASDR) or by the eMIOS timer hardware (DSPIx.SDR), or can be an arbitrary mixture of both. The source selection between these two registers can be configured by the DSPIx.DSICR.TXSS bit. If DSPIx.DSICR.TXSS = 1 then the complete frame is taken from the DSPIx.ASDR register; otherwise it is from the DSPIx.SDR register. In this case each bit of the frame can be taken from either the DSPIx.ASDR or the DSPIx.SDR register. This selection is determined by the DSPIx.SSR register on a bit-by-bit basis. If DSPIx.SSRx = 1 then the data is from the DSPIx.ADSR register; otherwise it is from the parallel input pins. The selection of which parallel input pin is routed to which bit position inside the DSPIx.SDR is explained in DSPI eMIOS selection for SDR register.

A copy of the last transmitted DSI frame is stored in the DSPIx.COMPR register. This register provides added visibility for debugging and serves as a reference for transfer initiation control. Figure 5 shows the dataflow described in this section.

Because the primary DSPI configuration registers support only 16-bit frame sizes, the DSPIx.DSICR.FMSZ[4] bit has been added to support frame sizes of 32 bits. When this bit is set, then sixteen will be added to the frame size defined by the DSPIx.CTAR.FMSZ field.

By serially chaining two of the DSPI blocks on the SoC, the frame size can even be extended to 64 bits, as described in DSPI serial chaining.

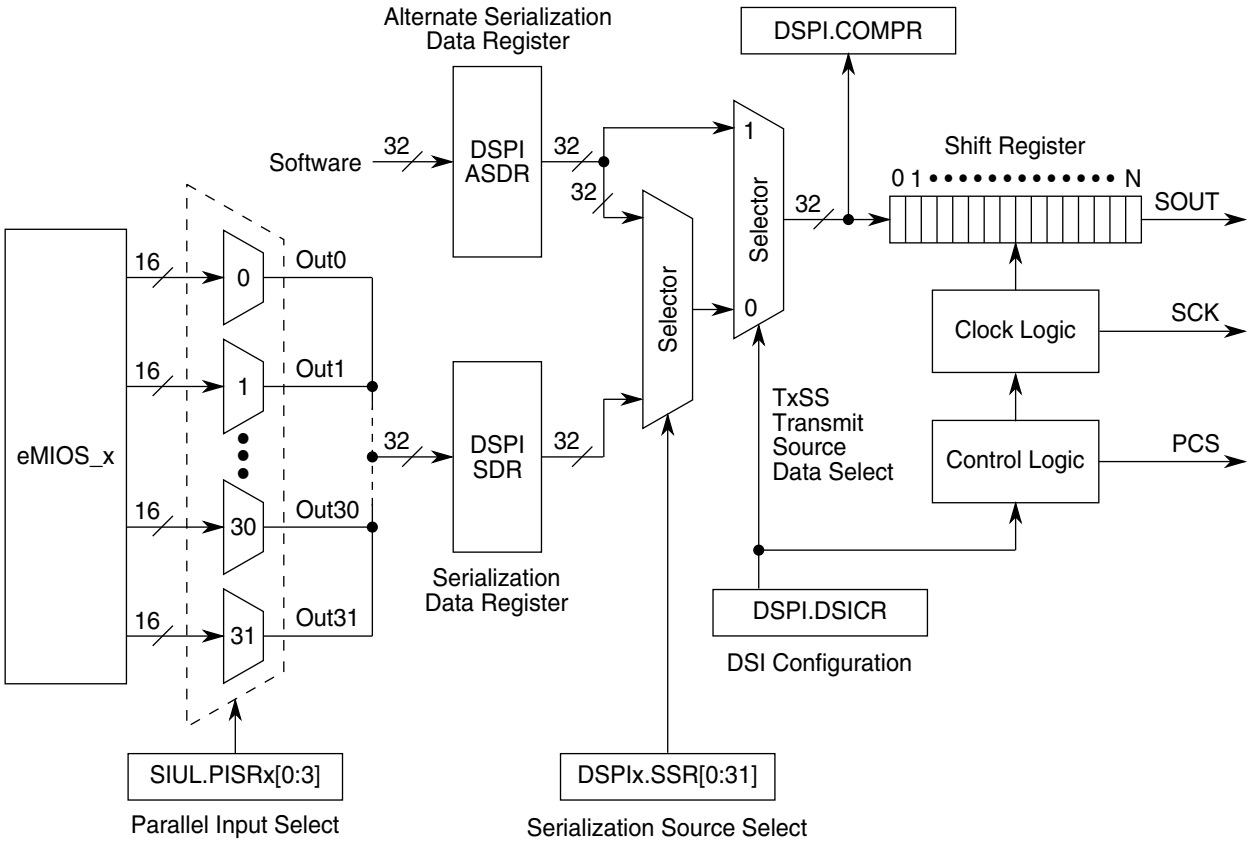


Figure 5. DSPI ASP serialization block level view

2.2 DSPI eMIOS selection for SDR register

As described previously in this application note, the new DSPI allows selecting data that is to be serialized from the eMIOS timer channels on a bit-by-bit basis, through the SIUL.PISRx registers. Altogether there are sixteen PISR registers in the SIUL which are mapped as discussed here.

On the MPC564xB there are four DSPI modules that implement the new functionality (DSPI0 to DSPI3) and two eMIOS timer blocks (eMIOS0 and eMIOS1). eMIOS0 outputs connect to DSPI0 and DSPI2, and eMIOS1 outputs connect to DSPI1 and DSPI3. This enables 64-bit frames to be built up by two of the eMIOS modules. How this is done is explained later in this document.

Each DSPIx.SDR register has thirty-two parallel inputs from the corresponding eMIOS module. The driver for each of these inputs can be one of sixteen eMIOS timer channels. The selection is done by the SIUL.PISRx registers. This results in four PISR registers per DSPI module as there are four bits needed for each multiplexer (IPS field). The IPS field is a 4-bit integer number that represents numbers from –8 to 7 (default=0). The input pin selected is defined by the sum of the IPS*n* field number and the IPS*n* field value. An example of how the selection works is shown in Table 4.

Table 4. DSPI pin selection example

Value	Mux0 (IPS0)	Mux16 (IPS16)	Mux31 (IPS31)
0x0	0	16	31
0x1	1	17	0

Table continues on the next page...

Table 4. DSPI pin selection example (continued)

Value	Mux0 (IPS0)	Mux16 (IPS16)	Mux31 (IPS31)
0x2	2	18	1
0x3	3	19	2
0x4	4	20	3
0x5	5	21	4
0x6	6	22	5
0x7	7	23	6
0xF (–1)	31	15	30
0xE (–2)	30	14	29
0xD (–3)	29	13	28
0xC (–4)	28	12	27
0xB (–5)	27	11	26
0xA (–6)	26	10	25
0x9 (–7)	25	9	24
0x8 (–8)	24	8	23

The DSPIx.SDR register holds the latest parallel input signal values, which are sampled at every rising edge of the DSPI system clock input. That is the input clock into the DSPI module to generate, for example, the baud rate.

Table 5. DSPI DSI Parallel Input Select Register 0 (DSPI_PISR0)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	IPS7				IPS6				IPS5				IPS4			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	IPS3				IPS2				IPS1				IPS0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.3 DSPI deserialization

On the receive side the data is deserialized automatically. When all bits of a DSI frame have been shifted in, they are stored in the deserialization data register (DSPIx.DDR). The DSPIx.DDR register is memory-mapped and allows host software to read the data directly. The received data is bit-wise compared to the value of the DSI Deserialized Data Polarity Interrupt Register (DSPIx.DPIR), bit-wise AND'ed with DSI Deserialized Interrupt Mask Register (DSPIx.DIMR), and the results OR'ed to produce DSPIx.DDIF flag in the DSPIx.SR register. This, in turn, can cause a DDI interrupt request if the DDIF_RE bit of DSPIx.RSER register is set and/or can stop DSI frame transmissions if the DMS bit of the DSPIx.DSICR register is set.

This process improves the diagnostic capabilities. It is now possible to generate an interrupt or flag just on certain bits and polarities of the received data. Figure 6 shows an overview of the data flow described in this chapter.

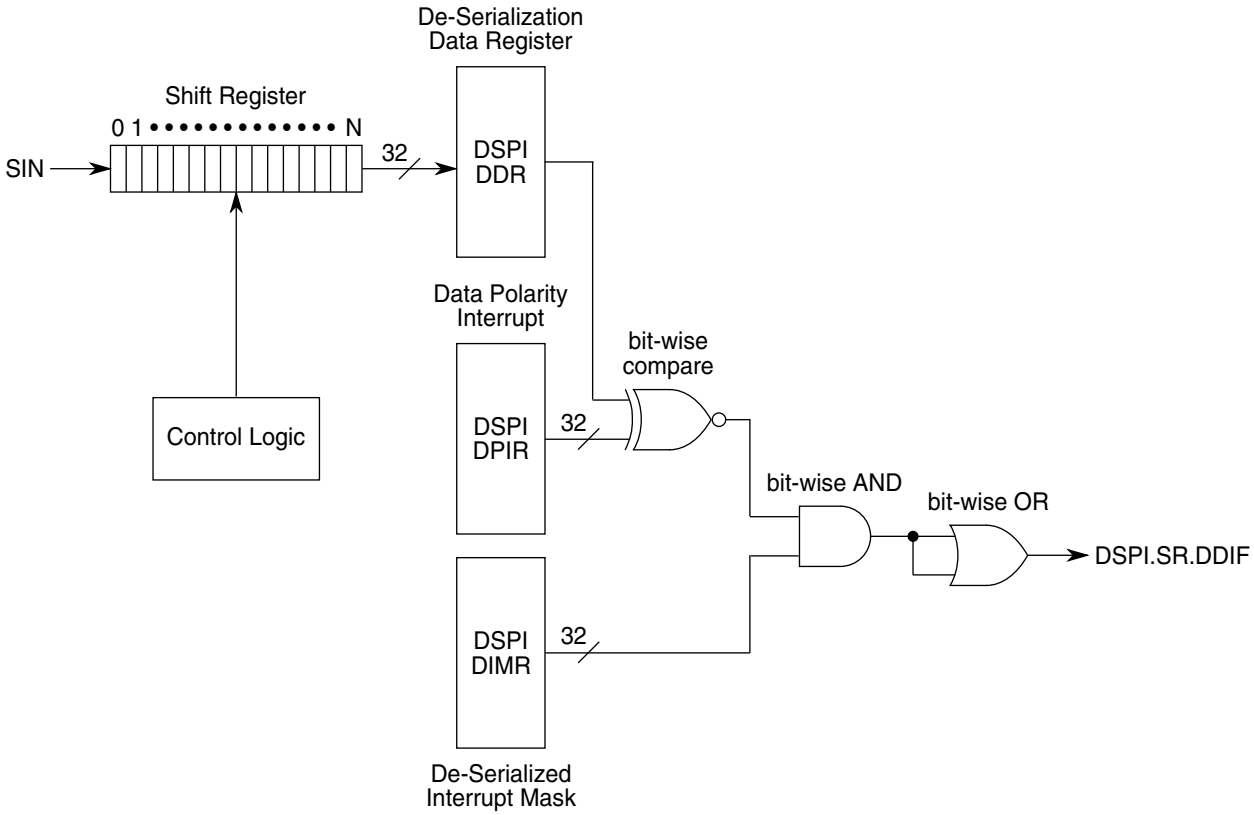


Figure 6. DSPI ASP deserialization block level view

2.4 DSPI serial chaining

To achieve frame sizes beyond the 32-bit boundary, two DSPI modules can be concatenated using the so-called multiple transfer operation (MTO). In this mode one DSPI module is operating as a master and the other as the slave. By using the DSPI input select register (SIUL.DISR) the connection of the modules can be configured. In the example that is shown in Figure 7 and is used in the demo code, the data output (SOUT) of the master is connected to the data input (SIN) of the slave. The SOUT of a slave is connected to an external pin, which connects to the input of an external SPI device. The slave DSPI and external SPI device use the master peripheral chip select (CSx) and clock (SCK). The trigger input of the master allows a slave DSPI to trigger a transfer when the data change occurs in the slave DSPI and the slave DSPI is operating in change in data mode. The trigger input of the master is connected to MTRIG output of the slave. The concatenated frames can be 8 to 64 bits long.

Only the concatenation of DSPI0 + DSPI1 or DSPI2 + DSPI3 is supported on MPC564xB.

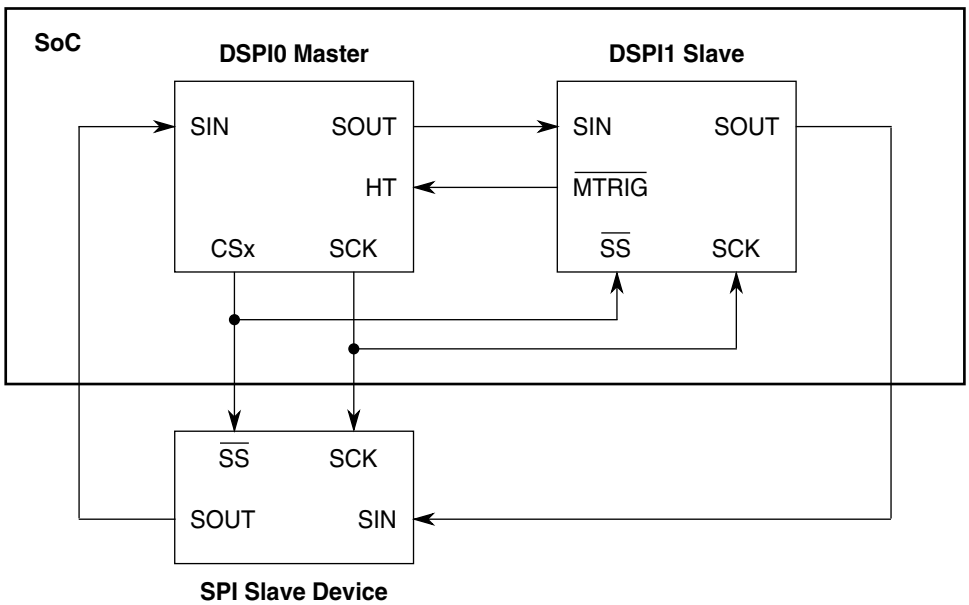


Figure 7. Example of serial chaining of DSPIs

2.5 DSPI CSI mode

The CSI mode is the combination of the classical SPI mode and the DSI mode for serialization. The CSI mode allows SPI and DSI frames to be interleaved on a frame-by-frame basis. SPI frame data is taken from the TX FIFO, and DSI frame data is taken from the DSPIx.SDR or DSPIx.ASDR registers, depending on the configuration. In this mode SPI frames in the TX FIFO have priority over DSI frames and will be transmitted as long as the TX FIFO is not empty. Then the DSI transfer will resume. When SPI frames are written into the TX FIFO they are transmitted at the next frame boundary. This mode can be configured by setting the DSPIx.MCR.DCONF[0:1] field to 0b10.

The received data is stored in either the DSPIx.DDR register or the RX FIFO, depending on the transfer type.

To differentiate between SPI and DSI frames the user can configure two CTARs and two different CS.

3 Conclusion

The discussion in What ASP is and why it is needed shows the challenges of purely software-generated PWM through SPI. By including hardware support for this kind of functionality the new Freescale devices will offer the same capabilities as before but with significantly less use of system resources and with re-established hardware triggering mechanisms between the timers and the ADC(s). In addition, it will also significantly reduce the software complexity and make maintenance easier while maintaining a high level of flexibility to construct the data frame. However, the system still stays backward-compatible within the family, as the SPI mechanisms did not change.

As an overview, the features of the new automatic serial PWM (ASP) generation are summarized below:

- Serialized data sources:
  - Up to 64 eMIOS output channels with selectable bit positions (DSPIx.SDR)
  - Memory-mapped register in the DSPI (DSPIx.ASDR)
  - Source select register to construct final data word (DSPIx.SSR)

- Deserialized data destinations:

  - Memory-mapped register in the DSPI (DSPIx.DDR)
  - Interrupt mask and bitwise polarity selection for triggering an interrupt from the received data (DSPIx.DIMR, DSPIx.DPIR)

- Transfer initiation conditions:
  - Continuous
  - Change in data
  - Edge-sensitive hardware

- Serial chaining of DSPI0 and DSPI1 or DSPI2 and DSPI3 to support 64-bit frames
- Support of mixing control messages and serialized data with CSI mode

4 ASP example code description

To ease the introduction for the user, an example project is delivered with this application note. This code should be a general example of how to use this feature and how to set up modes and registers described above. It is not intended to showcase a complete application but should be understood as a starting point to facilitate beginning a similar project. Furthermore, the user should be aware that this is example code only and not intended for production software whatsoever.

This example project was tested on the MPC56xx Freescale evaluation motherboard with an MPC564xB 208LQFP daughter card. The example project contains several use cases starting with the easiest way to create a 32-bit SPI frame and ending with the DSI mode, including CTU-triggered ADC conversions. The project is using the GHS compiler 5.1.7 and Lauterbach debugger. The software concept of the single use cases is described in this section on a case-by-case basis.

4.1 32-bit serialized frame in DSI mode

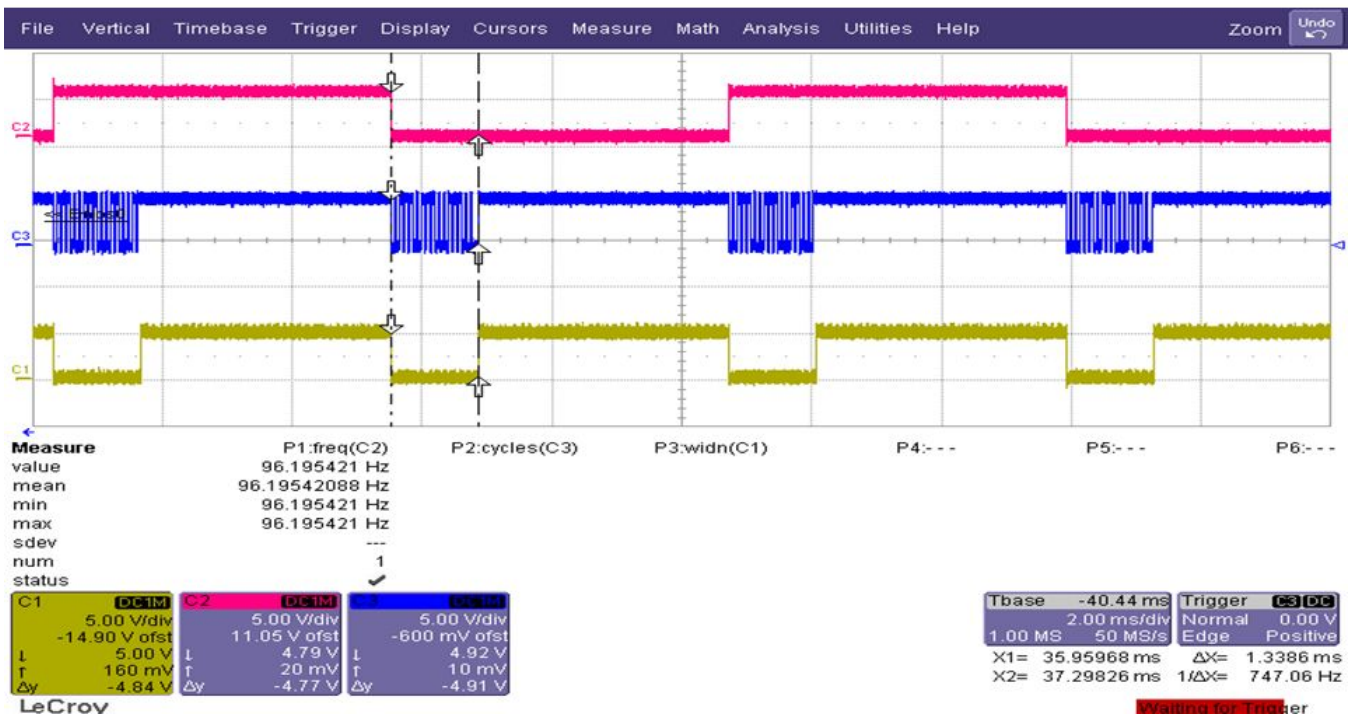
Project Name: DSPI_SDR_ASDR32

This example generates a 32-bit DSI frame on DSPI0. Bits 30 to 0 are taken from the DSPIx.ASDR register and are set constant to 0x55555555. Only bit position 31 is coming from eMIOS0 channel0, which is configured in PWM mode running at ~100Hz.

- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

Testcase – DSI mode:

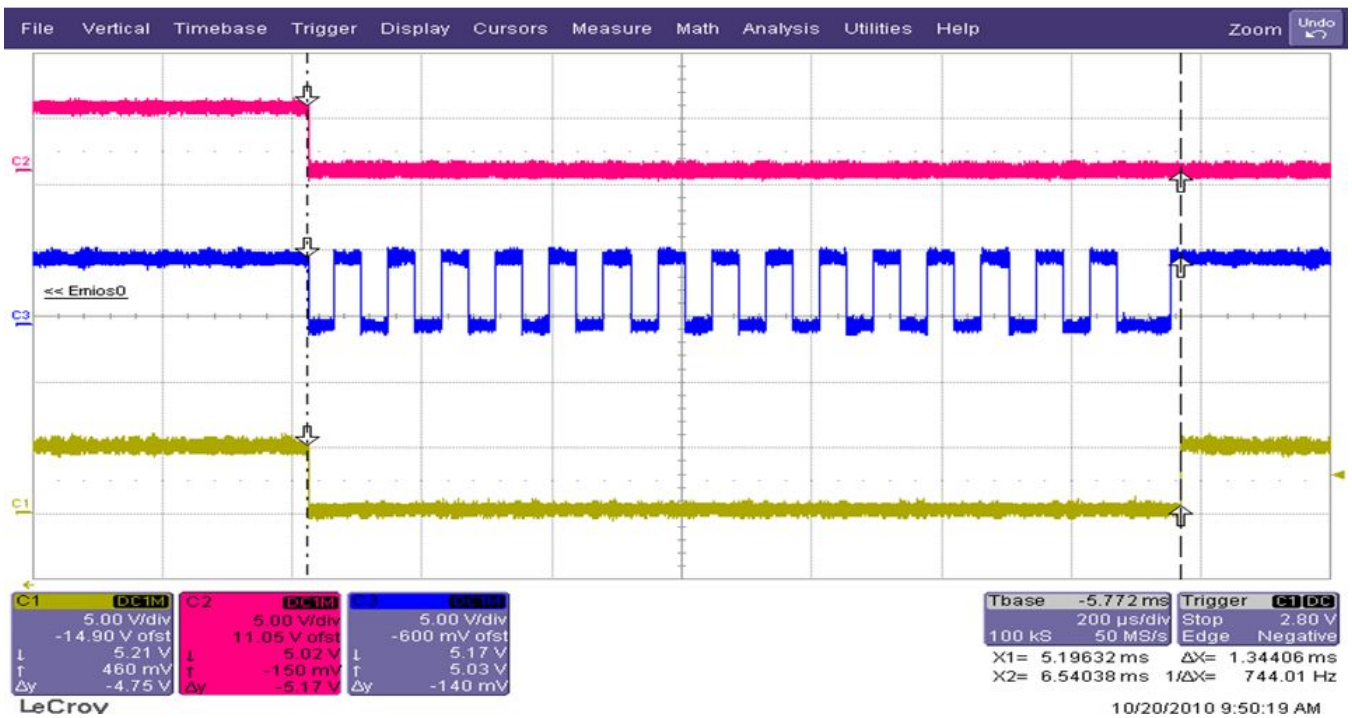
32Bit DSPI frame taken from ASDR except Bit31, which is coming from EMIOS0 channel0



- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

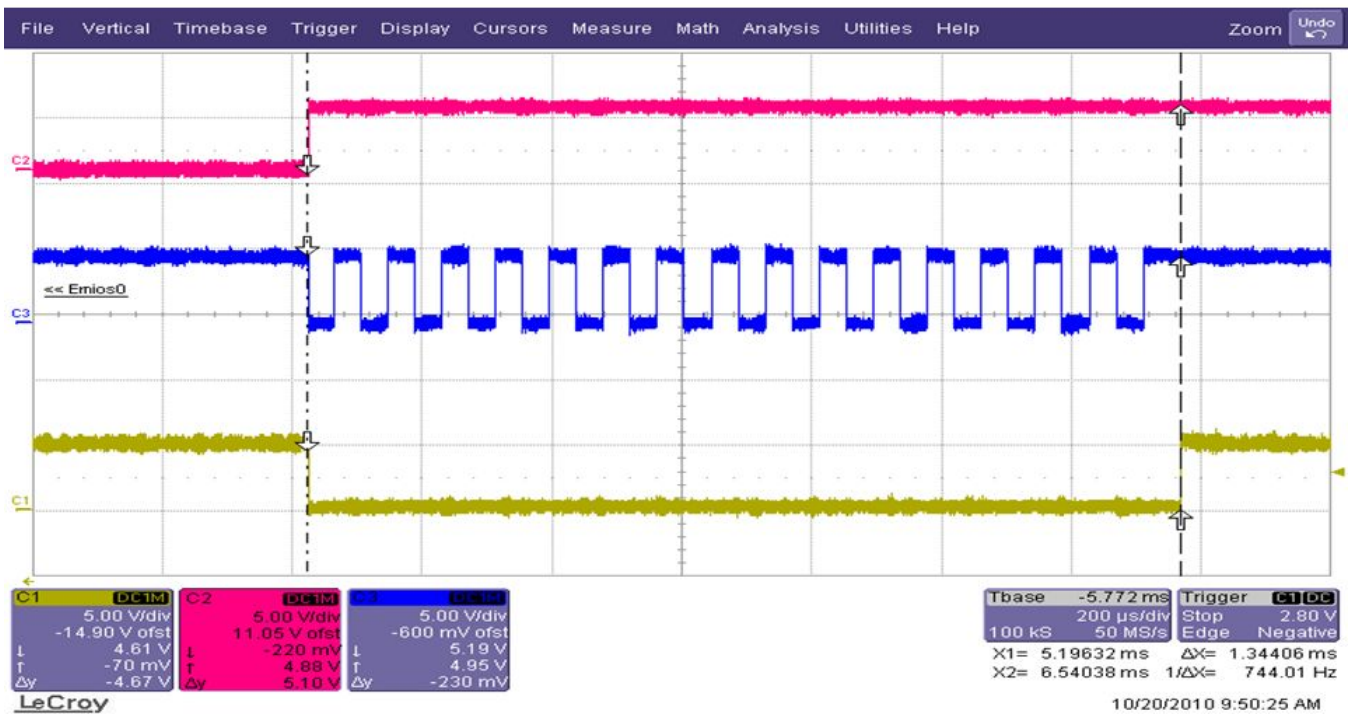
Testcase – DSI mode:

Zoom from previous. ASDR = 0x55555555 and EMIOS channel0 = 0



- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

Testcase – DSI mode:
Zoom from previous. ASDR = 0x55555555 and EMIOS channel0 = 1



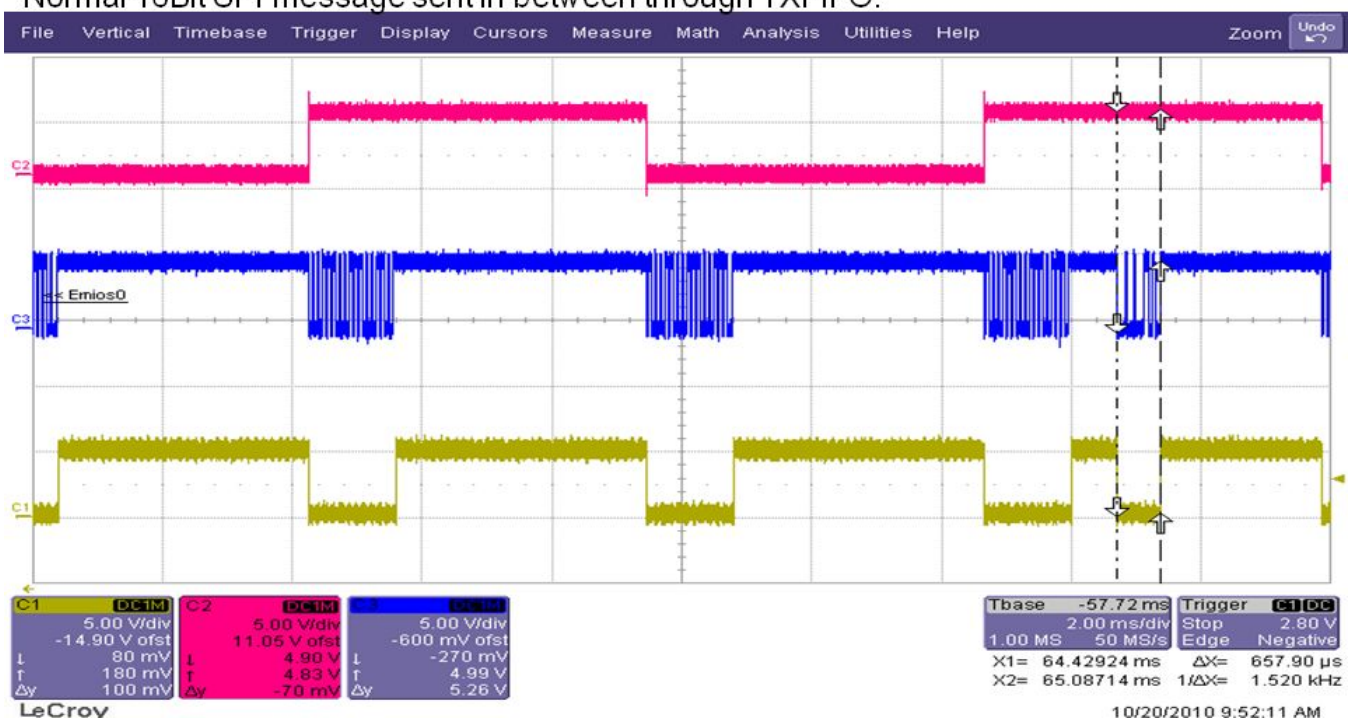
4.2 DSPI CSI mode

Project Name: DSPI_SDR_ASDR32_CSI

This example is taking the setup described in 32-bit serialized frame in DSI mode and sending CSI frames in between.

- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

Testcase – CSI mode:
32Bit DSPI frame taken from ASDR except Bit31, which is coming from EMIOS0 channel0
Normal 16Bit SPI message sent in between through TXFIFO.



Test case — CSI mode:

32-bit DSPI frame taken from ASDR except bit 31, which is coming from EMIOS0 channel0

Normal 16-bit SPI message sent in between through TXFIFO.

4.3 DSI mode with hardware-triggered ADC conversion

Project Name: DSPI_SDR_ASDR32_CTU

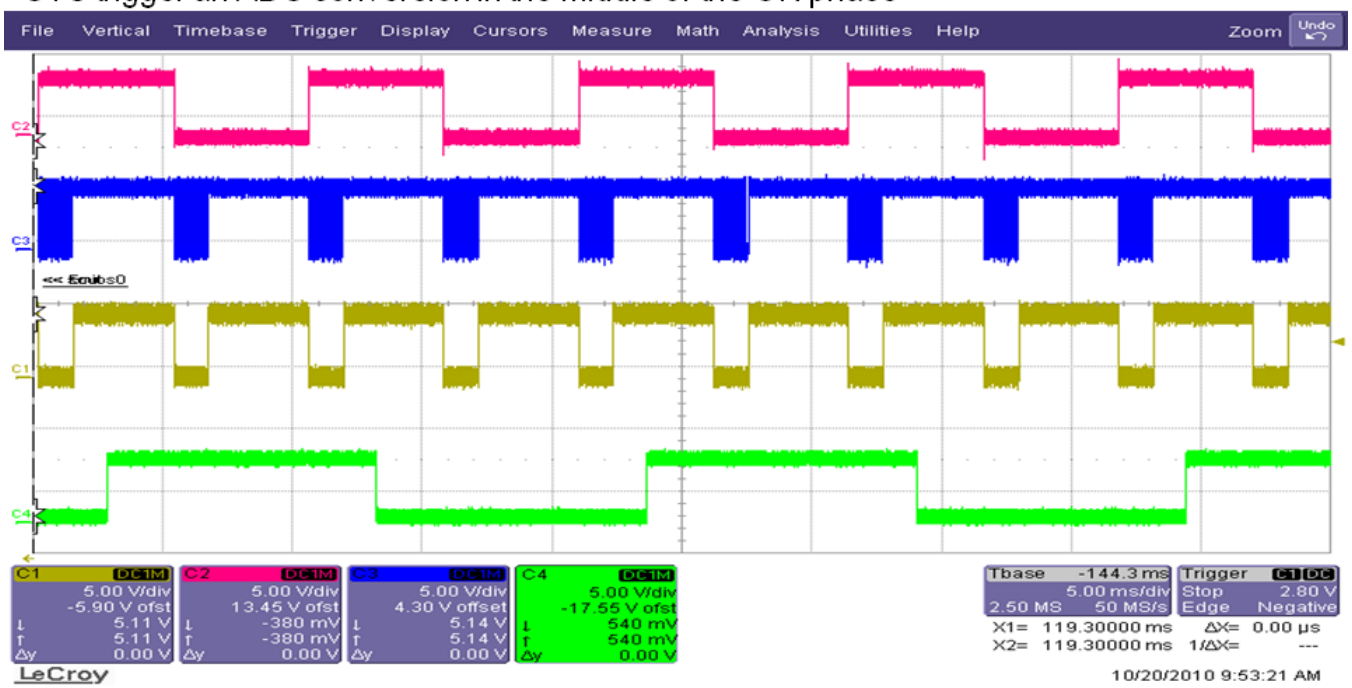
This example is taking the setup described in 32-bit serialized frame in DSI mode but now generates the PWM signal by using the eMIOS in the OPWMT mode. The hardware trigger for the ADC conversion is set to 50% of the on time. The trigger point can be made visible externally by toggling a pin within the ISR that is initiated after the ADC conversion has finished.

- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- GREEN = CTU Trigger (pin toggle in ADC end of conversion ISR)
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

Testcase – DSI mode:

32Bit DSPI frame taken from ASDR except Bit31, which is coming from EMIOS0 channel0

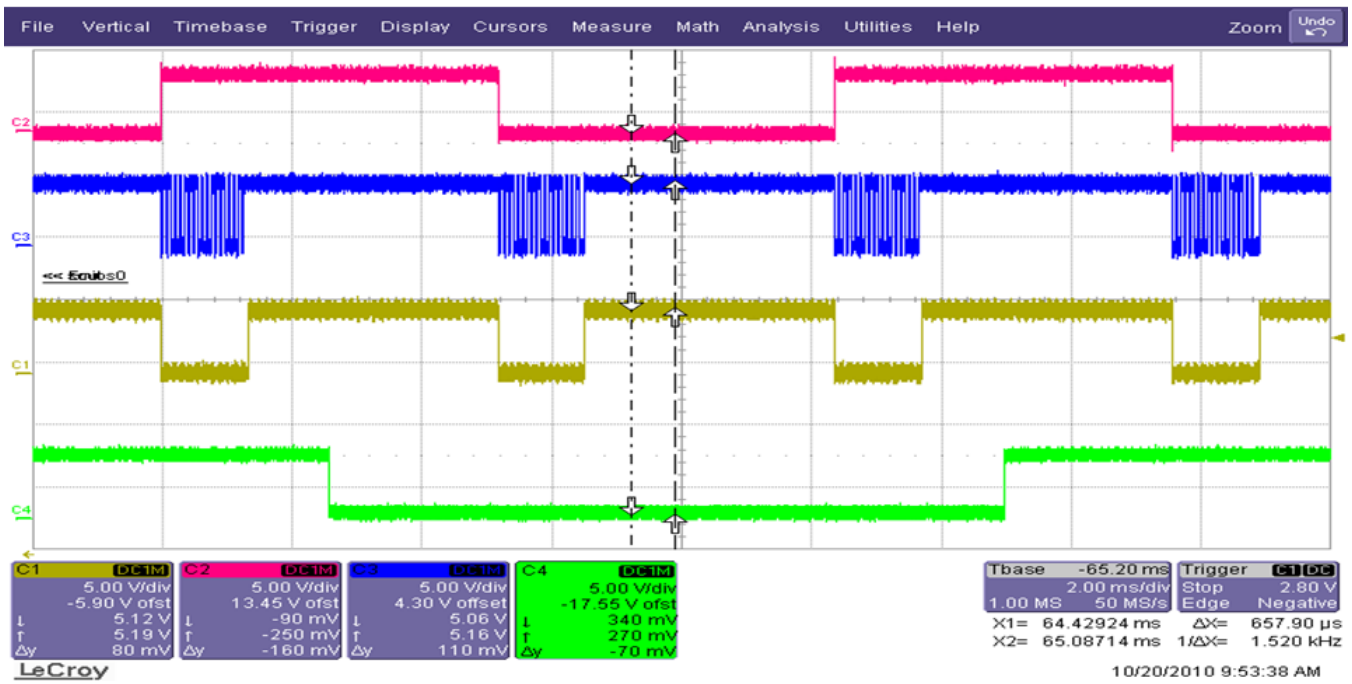
CTU trigger an ADC conversion in the middle of the ON phase



Zoom from previous to see the CTU trigger point.

- RED = EMIOS0 channel0 PWM @ 100Hz (deviation is coming from 16MHz IRC)
- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- GREEN = CTU Trigger (pin toggle in ADC end of conversion ISR)
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 32Bit frame it will take ~1,28ms (deviation from 16Mhz IRC)

Testcase – DSI mode:
Zoom from previous to clearly see the CTU trigger point.



4.4 Multiple transfer operation — serial chaining

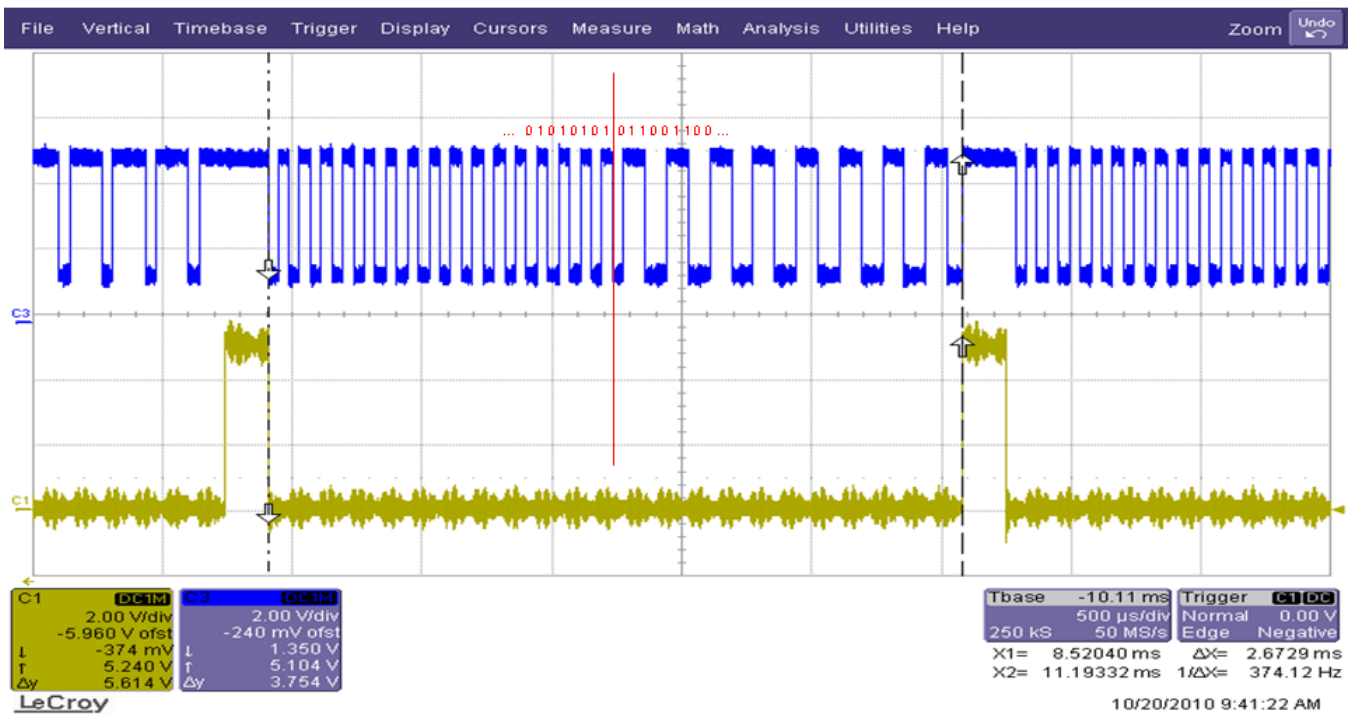
Project Name: DSPI_ ASDR_serial_chaining64

This example is generating a 64-bit DSI frame where DSPI0 and DSPI1 are serially chained. DSPI0 is the master and DSPI1 is the slave. The 64-bit data frame is created by writing 0x66666666 to the DSPI0.ASDR register and 0x55555555 to the DSPI1.ASDR register in one cycle, then 0xBBBBBBBB to the DSPI0.ASDR register and 0xAAAAAAAA to the DSPI1.ASDR register. The transfer is initiated by a change in data, so each time the register content changes the next 64-bit DSI frame is sent out.

- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 64Bit frame it will take ~2,56ms (deviation from 16Mhz IRC)

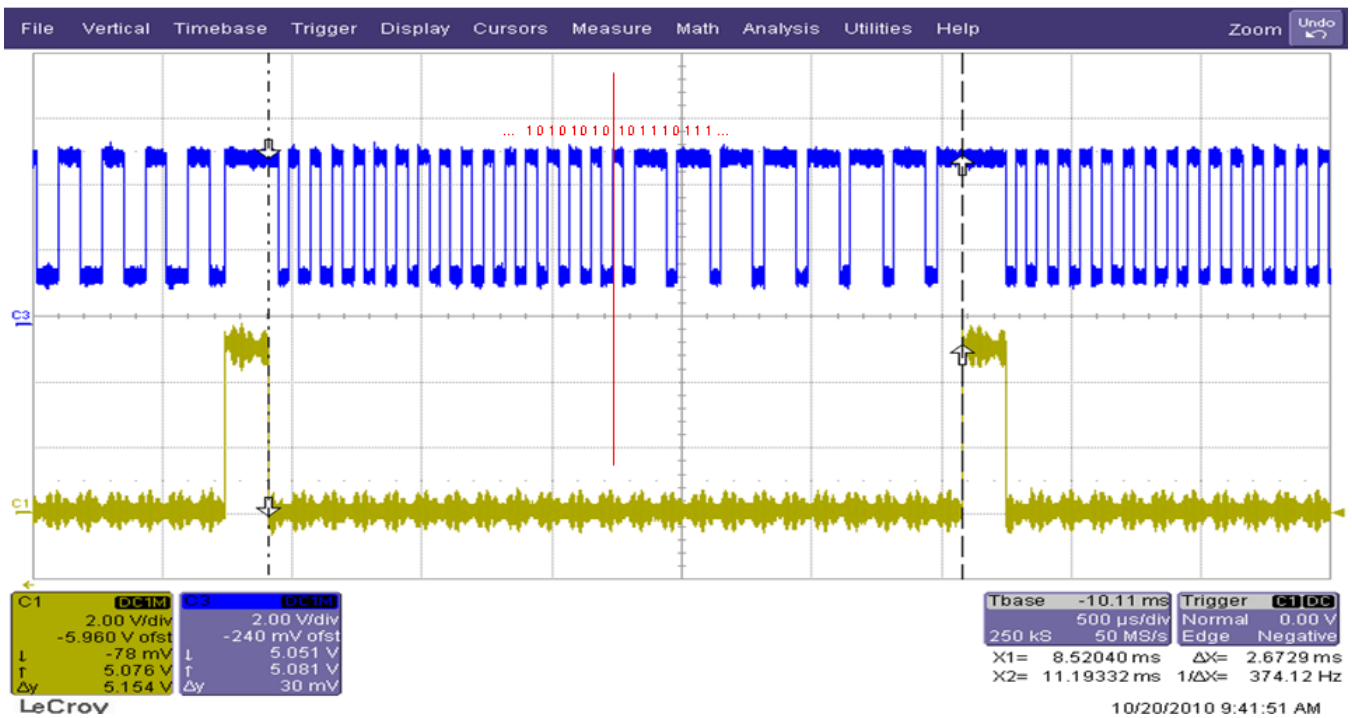
Testcase – serial chaining:

DSPI0 (Master) and DSPI1 serially chained to create a 64Bit frame from their ASDR register
DSPI1.ASDR=0x55555555 + DSPI0.ASDR=0x66666666



- BLUE = DSPI SOUT
- YELLOW = DSPI CS
- DSPI is running @ baud rate = 25kHz → 40us per Bit → a 64Bit frame it will take ~2,56ms (deviation from 16Mhz IRC)

Testcase – serial chaining:
DSPI0 (Master) and DSPI1 serially chained to create a 64Bit frame from their ASDR register
DSPI1.ASDR=0xAAAAAAAA + DSPI0.ASDR=0xBBBBBBBB



5 References

For the most up-to-date versions of Freescale documents, please go to freescale.com.

- AN2867, "Using the DSPI Module on the MPC5500 Family"
- AN2865, "MPC5500 & MPC5600 Simple Cookbook"
- MPC564xBCRM, *MPC564xB/C Reference Manual*

6 Glossary

ADC — Analog-to-digital converter

ASP — Automatic serial PWM generation

CPU — Central processing unit

CTU — Cross triggering unit

DMA — Direct memory access

DSPI — Deserial serial peripheral interface

eMIOS — Enhanced modular I/O subsystem

FIFO — First in / first out

GHS — Green Hills Software

MCU — Microcontroller unit

MTO — Multiple transfer operation

OPWMT — Output pulse-width-modulation–triggered

PIT — Periodic interrupt timer

PWM — Pulse width modulation

RAM — Random access memory

RX — Receive

SIUL — System integration unit lite

TX — Transmit

Appendix A Example code

A.1 32-bit serialized frame in DSI mode

A.1.1 Main.c

```
/*****************************************************************************/
/* FILE NAME: DSPI_SDR_ASDR32.c            COPYRIGHT (c) Freescale 2010      */
/*                                                  All Rights Reserved      */
/* DESCRIPTION:                                                              */
/* 32bit serialized SPI frame in DSI mode. Data is taken from the            */
/* DSPIMasterTransmitBuffer, but kept constant in this example               */
/* The device is running on 16MHz IRC and eMIOS0 channel0 is routed to       */
/* DSPI bit position 31.                                                     */
/* A 32bit SPI frame is always sent at a emios0.channel0 data change         */
/* It is assumed that DSPI_0 output is connectd to DSPI_0 input on the EVB*/
/*                                                                           */
/* SETUP:                                                                    */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach              */
/*   */
/*****************************************************************************/

#include "MPC5646x.h"

/*===========================================================================*/
/*Global variables         */
/*===========================================================================*/

const uint32_t  DSPIMasterTransmitBuffer[2] = {0x55555555,0x55555555};
uint32_t DSPI_frame, temp;
uint32_t TransmissionOKCounter = 0, TransmissionErrorCounter = 0, DDIFFlagCounter = 0;


int main(int argc, char *argv[])
{
```

```
int i = 0, k = 0;

 DISABLE_WATCHDOG(); // watchdog is enabled by default
MODE_INIT();   // enable all peripherals in all modes with all peripheral clocks

    SIU.PCR[0].R = 0x0604;  //MPC56xxB: Config pad as EMIOS0.channel0 output

    // initialize the EMIOS.channel0 for PWM generation in OPWMT mode 100Hz
    Mcu_Emios_0_Init();

    // initialize DSPI
    // DSPI's are part of peripheral set 2, which can run up to 64MHz max
    // but is running at 16MHz system clock in this case
    initDSPI_0_serial();



while (1)
  {
  //Configure ASDR registers of DSPI A
  DSPI_0.ASDR.R = DSPIMasterTransmitBuffer[i];
  i++;

  // Wait the transmission to be completed
  while(DSPI_0.SR.B.TCF != 0x1);
  // Clear the TCF
  DSPI_0.SR.R = 0x80000000;

  for (k=0;k<1000;k++); // w

  DSPI_frame = DSPI_0.DDR.R; // read out received data

  if (DSPI_frame == DSPI_0.COMPR.R) // compare against data sent
    TransmissionOKCounter++;
  else
    // Error Counter may be one as the transmission starts directly
    TransmissionErrorCounter++;

  // DDIFFlagCounter should toggle every second transmission on a one, so
TransmissionOKCounter/2 = DDIFFlagCounter
  if (DSPI_0.SR.B.DDIF) {DDIFFlagCounter++;DSPI_0.SR.B.DDIF=1;}

  if (i==2) {i=0;} // go to beginning of TransmitBuffer if the end is reached
  }

}
```

A.1.2 function.c

```
/*************************************************************************/
/* FILE NAME: functions.c                  COPYRIGHT (c) Freescale 2010   */
/*                                              All Rights Reserved       */
/* DESCRIPTION:                                                          */
/* This is a basic collection of useful functions                        */
/*                                                                       */
/* SETUP:                                                                */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach          */
/*  */
/*=======================================================================*/

#include "MPC5646x.h"
#include <ppc_ghs.h>

/*=======================================================================*/
/* DEFINES    */
/*=======================================================================*/
```

```
#define DEBOUNCEDELAYTIME 0xFFFF

#define DRUN_MODE 0x03
#define RUN0_MODE 0x04
#define STOP_MODE 0x0A
#define STANDBY_MODE 0x0D


// eMIOS PWM channel assignment
#define PWM1  0

// PWM 100Hz period
// PWM Duty cycle 0.5% step
#define DUTY_CYCLE_STEP_100   200
// PWM Start delay 0.5% step
#define START_DELAY_TIME_STEP_100   200
// PWM ADC trigger point 0.5% step
#define TRIGGER_ADC_STEP_100  200
// PWM period counter roll over value
#define COUNTER_PERIOD_100    40000

/*=========================================================================*/
/* VARIABLES  */
/*=========================================================================*/
long count;

// eMIOS channel control structure
typedef struct
{
vuint8_t dutyCycle;       // PWM Duty cycle (from 0 to 200 - 0.5% step)
vuint8_t startDelayTime;  // PWM shift to base (from 0 to 200 - 0.5% step)
vuint8_t triggerAdc;      // ADC trigger point (from 0 to 200 - 0.5% step)
}chPar;

// Channels parameters array
chPar ch[];

/*=========================================================================*/
/* FUNCTIONS  */
/*=========================================================================*/

 void initDSPI_0_serial(void) {

   //Program to enter into master DSI mode
  //-- Configure DSPIA as master
  DSPI_0.MCR.R = 0x90010001;
  // configure CTAR0
  // with 16Mhz input clock, BR=128, PBR=5, DBR=0
  // SCK baud rate = (fsys/PBR) * (1+DBR/BR) = 25kHz --> 40us
  // for 32Bit frame it will take ~1,28ms
  DSPI_0.CTAR[0].R = 0x780A7727;

  //Test: DSPI A transmission is started when ADSR is loaded
  //DSPI A - DSI: FMSZ[4], use DSPI_ASDR
  //transmit when CID, assert PCS0
  DSPI_0.DSICR.R = 0x40010001;
  // transmit continuously
  //DSPI_0.DSICR.R = 0x40000001;

  // enable DSPIs
  DSPI_0.MCR.B.HALT = 0x0;


  // DSPI_0 pin settings
  SIU.PCR[13].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SOUT output */
  SIU.PCR[12].R = 0x0103;        /* MPC56xxB: Config pad as DSPI_0 SIN input */
  SIU.PCR[14].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SCK output */
  SIU.PCR[15].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 PCS0 output */
```

```
  // DSPI serialized source select register
  DSPI_0.SSR.R = 0xFFFFFFFE; // take all bits except bit31 from the ASDR register

  // play with the DDIF interrupt flag
  // DSPI data Interrupt Mask register
  DSPI_0.DIMR.R = 0x00000001; // enable only bit31 of DSPI_0

  // DSPI deserialized data polarity interrupt register
  DSPI_0.DPIR.R = 0x00000001; // enable only bit31 of DSPI_0 triggers the flag when it is 1

}


void Mcu_Emios_0_Channel_Config(vuint8_t i)
{
vuint32_t chDutyCycle, chStartDelay, chTriggerAdc, counterPeriod;
vuint16_t chDutyCycleStep, chStartDelayStep, chTriggerAdcStep;


// Load recalculation constants
chDutyCycleStep = START_DELAY_TIME_STEP_100;
chStartDelayStep = DUTY_CYCLE_STEP_100;
chTriggerAdcStep = TRIGGER_ADC_STEP_100;
counterPeriod = COUNTER_PERIOD_100;

    // Channel Start delay time
    // Recalculate PWM leading edge Start delay time
chStartDelay = (vuint32_t)(ch[i].startDelayTime * chStartDelayStep);
// Load PWM leading edge start delay time to A1
EMIOS_0.CH[i].CADR.R = chStartDelay;


    // Channel PWM Duty Cycle
// Recalculate Duty cycle
chDutyCycle = (vuint32_t)(ch[i].dutyCycle * chDutyCycleStep);
    // Load duty cycle to channel B2 register (transfered to B1 on counter A1 match)
if (chDutyCycle == counterPeriod)
{
// PWM Duty cycle 100%
EMIOS_0.CH[i].CBDR.R = chDutyCycle;
}
else if ((chDutyCycle + chStartDelay) > counterPeriod)
{
// PWM trailing edge after counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay - counterPeriod;
}
else
{
// PWM trailing edge before counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay;
}


    // Channel ADC trigger point
// Recalculate Trigger ADC
chTriggerAdc = (vuint32_t)(ch[i].triggerAdc * chTriggerAdcStep);
// Load channel trigger ADC to A2 register
if ((chTriggerAdc + chStartDelay) > counterPeriod)
{
// Channel ADC trigger occurs after counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay - counterPeriod;
}
else
{
// Channel ADC trigger occurs before counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay;
}
}
```

```
void Mcu_Emios_0_Channel_Init(void)
{
// Channel PWM1
// Leading edge at begin of counting cycle
ch[PWM1].startDelayTime = 0;
// Duty cycle 50%
ch[PWM1].dutyCycle = 100;
// ADC trigger point 25%
ch[PWM1].triggerAdc = 50;

// Configure eMIOS channels PWM1
Mcu_Emios_0_Channel_Config(PWM1);
}


void Mcu_Emios_0_Init(void)
{
// Module Configuration register
// Global prescaler enable, divide ratio 4  --> 4MHz input clock for UC's
EMIOS_0.MCR.R = 0x04000300;

// Module Output Update Disable register
// Enable transfers (A2 -> A1, B2 -> B1) for all 28 channel registers
EMIOS_0.OUDR.R = 0x00000000;

    // Enable channels: 0, 23 (0 -> enable)
EMIOS_0.UCDIS.R = 0xFFFFFFFF;
EMIOS_0.UCDIS.B.CHDIS0 = 0;
EMIOS_0.UCDIS.B.CHDIS23 = 0;

// Channel 23 configuration - MC mode - counter bus A 100Hz
// Enable prescaler (divide ration 1 --> 4MHz), global clock, MC mode
EMIOS_0.CH[23].CCR.R = 0x02000310;
// Config A1: 4MHz/100Hz = 40000, counter bus A period = A1 + 1 => A1 = 39999
EMIOS_0.CH[23].CADR.R = COUNTER_PERIOD_100 - 1;

// PWM1 channel configuration - OPWMT mode
// Enable prescaler (divide ration 1), counter bus A clock, OPWMT mode
EMIOS_0.CH[PWM1].CCR.R = 0x02000026;
// A match comparator A sets output, while match B clears it.
EMIOS_0.CH[PWM1].CCR.B.EDPOL = 1;
// Flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.DMA   = 1;
// Enable the flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.FEN   = 1;

// Load PWM data into channels registers
Mcu_Emios_0_Channel_Init();

// Global time base enable
EMIOS_0.MCR.B.GTBE = 1;
}

void MODE_INIT(void)
{
   /* Enbable all peripheral clocks */
   CGM.SC_DC[0].B.DIV = 0x80;
   CGM.SC_DC[1].B.DIV = 0x80;
   CGM.SC_DC[2].B.DIV = 0x80;
   /* Setting RUN Configuration Register ME_RUN_PC[0] */
   ME.RUNPC[0].R=0x000000FE; /* Peripheral ON in every mode */

   /* Re-enter in DRUN mode to update */
   ME.MCTL.R = 0x30005AF0;       /* Mode & Key */
   ME.MCTL.R = 0x3000A50F;       /* Mode & Key */

}

void DISABLE_WATCHDOG()
{
```

```
  SWT.SR.R = 0x0000c520; /* key */
  SWT.SR.R = 0x0000d928; /* key */
  SWT.CR.R = 0xC000010A; /* disable WEN */
}
```

A.2 DSPI CSI mode

A.2.1 Main.c

```
/*************************************************************************/
/* DESCRIPTION:                                                          */
/* 32bit serialized SPI frame in DSI mode. Data is taken from the        */
/* DSPIMasterTransmitBuffer, but kept constant in this example           */
/* The device is running on 16MHz IRC and eMIOS0 channel0 is routed to   */
/* DSPI bit position 31.                                                 */
/* A 32bit SPI frame is always sent at a emios0.channel0 data change     */
/* It is assumed that DSPI_0 output is connectd to DSPI_0 input on the EVB*/
/* Another value is written unsynchronized to the SPI push register      */
/* to transmit a normal SPI frame                                        */
/*                                                                       */
/* SETUP:                                                                */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach          */
/*  */
/*************************************************************************/

#include "MPC5646x.h"

/*=======================================================================*/
/*Global variables             */
/*=======================================================================*/


const uint32_t  DSPIMasterTransmitBuffer[2] = {0x55555555,0x55555555};
uint32_t DSPI_frame, temp;
uint32_t TransmissionOKCounter = 0, TransmissionErrorCounter = 0, DDIFFlagCounter = 0;



// testcase is to loop DSPI_1 SOUT to DSPI_0 SIN on the EVB
int main(int argc, char *argv[])
{

int i = 0, k = 0;

 DISABLE_WATCHDOG(); // watchdog is enabled by default
MODE_INIT();   // enable all peripherals in all modes with all peripheral clocks

  // Set PCR for channel0
   SIU.PCR[0].R = 0x0604;        // MPC56xxB: Config pad as EMIOS0.channel0 output

  // initialize the EMIOS.channel0 for PWM generation in OPWMT mode 100Hz
  Mcu_Emios_0_Init();

  // initialize DSPI's
  // DSPI's are part of peripheral set 2, which can run up to 64MHz max
  // but is running at 16MHz in this case
  initDSPI_0_serial();



while (1)
  {
```

```
  //Configure ASDR registers of DSPI A and B
  DSPI_0.ASDR.R = DSPIMasterTransmitBuffer[i];
  i++;

  for (k=0;k<100000;k++);

  // send an intermediate SPI frame. This might have to be done by hand
  // or with breakpoints in the debugger as this is not synchronised to the serialized frame
  DSPI_0.PUSHR.R = 0x00011234; // an intermediate SPI frame

 // assuming the DSPI_0 SOUt is connected to DSPI_0 SIN pin
  // Wait the transmission to be completed
  while(DSPI_0.SR.B.TCF != 0x1);
  // Clear the TCF
  DSPI_0.SR.R = 0x80000000;

  DSPI_frame = DSPI_0.DDR.R;

  if (DSPI_frame == DSPI_0.COMPR.R)
    TransmissionOKCounter++;
  else
    // Error Counter may be one as the transmission starts directly
    TransmissionErrorCounter++;

  // DDIFFlagCounter should toggle every second transmission, so TransmissionOKCounter/2 =
DDIFFlagCounter
  if (DSPI_0.SR.B.DDIF) {DDIFFlagCounter++;DSPI_0.SR.B.DDIF=1;}



  if (i==2) {i=0;}
  }

}
```

A.2.2 function.c

```
/*****************************************************************************/
/* FILE NAME: functions.c                  COPYRIGHT (c) Freescale 2010   */
/*                                                All Rights Reserved     */
/* DESCRIPTION:                                                           */
/* This is a basic collection of useful functions                         */
/*                                                                        */
/* SETUP:                                                                 */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach           */
/*  */
/*===========================================================================*/

#include "MPC5646x.h"
#include <ppc_ghs.h>

/*===========================================================================*/
/* DEFINES    */
/*===========================================================================*/

#define DEBOUNCEDELAYTIME 0xFFFF

#define DRUN_MODE 0x03
#define RUN0_MODE 0x04
#define STOP_MODE 0x0A
#define STANDBY_MODE 0x0D


// eMIOS PWM channel assignment
#define PWM1  0
```

```
// PWM 100Hz period
// PWM Duty cycle 0.5% step
#define DUTY_CYCLE_STEP_100   200
// PWM Start delay 0.5% step
#define START_DELAY_TIME_STEP_100   200
// PWM ADC trigger point 0.5% step
#define TRIGGER_ADC_STEP_100  200
// PWM period counter roll over value
#define COUNTER_PERIOD_100    40000

/*=========================================================================*/
/* VARIABLES  */
/*=========================================================================*/
long count;

// eMIOS channel control structure
typedef struct
{
vuint8_t dutyCycle;       // PWM Duty cycle (from 0 to 200 - 0.5% step)
vuint8_t startDelayTime;  // PWM shift to base (from 0 to 200 - 0.5% step)
vuint8_t triggerAdc;      // ADC trigger point (from 0 to 200 - 0.5% step)
}chPar;

// Channels parameters array
chPar ch[];

/*=========================================================================*/
/* FUNCTIONS  */
/*=========================================================================*/

 void initDSPI_0_serial(void) {

   //Program to enter into master CSI mode
  //-- Configure DSPIA as master
  DSPI_0.MCR.R = 0xA0010001;
  // configure CTAR0
  // with 16Mhz input clock, BR=128, PBR=5, DBR=0
  // SCK baud rate = (fsys/PBR) * (1+DBR/BR) = 25kHz --> 40us
  // for 32Bit frame it will take ~1,28ms
  DSPI_0.CTAR[0].R = 0x780A7727;

  //Test: DSPI A transmission is started when ADSR is loaded
  //DSPI A - DSI: FMSZ[4], use DSPI_ASDR
  //transmit when CID, assert PCS0
  DSPI_0.DSICR.R = 0x40010001;
  // transmit continuously
  //DSPI_0.DSICR.R = 0x40000001;

  // enable DSPIs
  DSPI_0.MCR.B.HALT = 0x0;


  // DSPI_0 pin settings
  SIU.PCR[13].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SOUT output */
  SIU.PCR[12].R = 0x0103;        /* MPC56xxB: Config pad as DSPI_0 SIN input */
  SIU.PCR[14].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SCK output */
  SIU.PCR[15].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 PCS0 output */


  // DSPI serialized source select register
  DSPI_0.SSR.R = 0xFFFFFFFE; // take all bits except bit31 from the ASDR register

  // play with the DDIF interrupt flag
  // DSPI data Interrupt Mask register
  DSPI_0.DIMR.R = 0x00000001; // enable only bit31 of DSPI_0

  // DSPI deserialized data polarity interrupt register
  DSPI_0.DPIR.R = 0x00000001; // enable only bit31 of DSPI_0 triggers the flag when it is 1

}
```

```
void Mcu_Emios_0_Channel_Config(vuint8_t i)
{
vuint32_t chDutyCycle, chStartDelay, chTriggerAdc, counterPeriod;
vuint16_t chDutyCycleStep, chStartDelayStep, chTriggerAdcStep;


// Load recalculation constants
chDutyCycleStep = START_DELAY_TIME_STEP_100;
chStartDelayStep = DUTY_CYCLE_STEP_100;
chTriggerAdcStep = TRIGGER_ADC_STEP_100;
counterPeriod = COUNTER_PERIOD_100;

    // Channel Start delay time
    // Recalculate PWM leading edge Start delay time
chStartDelay = (vuint32_t)(ch[i].startDelayTime * chStartDelayStep);
// Load PWM leading edge start delay time to A1
EMIOS_0.CH[i].CADR.R = chStartDelay;


    // Channel PWM Duty Cycle
// Recalculate Duty cycle
chDutyCycle = (vuint32_t)(ch[i].dutyCycle * chDutyCycleStep);
    // Load duty cycle to channel B2 register (transfered to B1 on counter A1 match)
if (chDutyCycle == counterPeriod)
{
// PWM Duty cycle 100%
EMIOS_0.CH[i].CBDR.R = chDutyCycle;
}
else if ((chDutyCycle + chStartDelay) > counterPeriod)
{
// PWM trailing edge after counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay - counterPeriod;
}
else
{
// PWM trailing edge before counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay;
}


    // Channel ADC trigger point
// Recalculate Trigger ADC
chTriggerAdc = (vuint32_t)(ch[i].triggerAdc * chTriggerAdcStep);
// Load channel trigger ADC to A2 register
if ((chTriggerAdc + chStartDelay) > counterPeriod)
{
// Channel ADC trigger occurs after counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay - counterPeriod;
}
else
{
// Channel ADC trigger occurs before counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay;
}
}

void Mcu_Emios_0_Channel_Init(void)
{
// Channel PWM1
// Leading edge at begin of counting cycle
ch[PWM1].startDelayTime = 0;
// Duty cycle 50%
ch[PWM1].dutyCycle = 100;
// ADC trigger point 25%
ch[PWM1].triggerAdc = 50;

// Configure eMIOS channels PWM1
Mcu_Emios_0_Channel_Config(PWM1);
```

```
}


void Mcu_Emios_0_Init(void)
{
// Module Configuration register
// Global prescaler enable, divide ratio 4  --> 4MHz input clock for UC's
EMIOS_0.MCR.R = 0x04000300;

// Module Output Update Disable register
// Enable transfers (A2 -> A1, B2 -> B1) for all 28 channel registers
EMIOS_0.OUDR.R = 0x00000000;

    // Enable channels: 0, 23 (0 -> enable)
EMIOS_0.UCDIS.R = 0xFFFFFFFF;
EMIOS_0.UCDIS.B.CHDIS0 = 0;
EMIOS_0.UCDIS.B.CHDIS23 = 0;

// Channel 23 configuration - MC mode - counter bus A 100Hz
// Enable prescaler (divide ration 1 --> 4MHz), global clock, MC mode
EMIOS_0.CH[23].CCR.R = 0x02000310;
// Config A1: 4MHz/100Hz = 40000, counter bus A period = A1 + 1 => A1 = 39999
EMIOS_0.CH[23].CADR.R = COUNTER_PERIOD_100 - 1;

// PWM1 channel configuration - OPWMT mode
// Enable prescaler (divide ration 1), counter bus A clock, OPWMT mode
EMIOS_0.CH[PWM1].CCR.R = 0x02000026;
// A match comparator A sets output, while match B clears it.
EMIOS_0.CH[PWM1].CCR.B.EDPOL = 1;
// Flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.DMA   = 1;
// Enable the flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.FEN   = 1;

// Load PWM data into channels registers
Mcu_Emios_0_Channel_Init();

// Global time base enable
EMIOS_0.MCR.B.GTBE = 1;
}

void MODE_INIT(void)
{
   /* Enbable all peripheral clocks */
   CGM.SC_DC[0].B.DIV = 0x80;
   CGM.SC_DC[1].B.DIV = 0x80;
   CGM.SC_DC[2].B.DIV = 0x80;
   /* Setting RUN Configuration Register ME_RUN_PC[0] */
   ME.RUNPC[0].R=0x000000FE; /* Peripheral ON in every mode */

   /* Re-enter in DRUN mode to update */
   ME.MCTL.R = 0x30005AF0;       /* Mode & Key */
   ME.MCTL.R = 0x3000A50F;       /* Mode & Key */

}

void DISABLE_WATCHDOG()
{
  SWT.SR.R = 0x0000c520; /* key */
  SWT.SR.R = 0x0000d928; /* key */
  SWT.CR.R = 0xC000010A; /* disable WEN */
}
```

A.3 DSI mode with hardware-triggered ADC conversion

A.3.1 Main.c

```
/*************************************************************************/
/* FILE NAME: DSPI_SDR_ASDR32_CTU.c        COPYRIGHT (c) Freescale 2010   */
/*                                                  All Rights Reserved    */
/* DESCRIPTION:                                                           */
/* 32bit serialized SPI frame in DSI mode. Data is taken from the         */
/* DSPIMasterTransmitBuffer, but kept constant in this example            */
/* The device is running on 16MHz IRC and eMIOS0 channel0 is routed to    */
/* DSPI bit position 31.                                                  */
/* A 32bit SPI frame is always sent at a emios0.channel0 data change      */
/* The eMIOS is setup in the OPWMT mode with CTU trigger in the mid of the*/
/* PWM period. The CTU will trigger ADC0 channel0. A pin is toggled in the*/
/* ISR to signal the trigger to the outside world                         */
/*                                                                        */
/* SETUP:                                                                 */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach           */
/*  */
/*************************************************************************/

#include "MPC5646x.h"

/*=======================================================================*/
/*Global variables  */
/*=======================================================================*/

const uint32_t  DSPIMasterTransmitBuffer[2] = {0x55555555,0x55555555};
uint32_t DSPI_frame, temp;
uint32_t TransmissionOKCounter = 0, TransmissionErrorCounter = 0, DDIFFlagCounter = 0;



// testcase is to loop DSPI_1 SOUT to DSPI_0 SIN on the EVB
int main(int argc, char *argv[])
{

int i = 0, k = 0;

DISABLE_WATCHDOG(); // watchdog is enabled by default
MODE_INIT();   // enable all peripherals in all modes with all peripheral clocks

   // Set PCR for channel0
   SIU.PCR[0].R = 0x0604;        // MPC56xxB: Config pad as EMIOS0.channel0 output

   set_GPIOout_pin(68);     // blink with PE4 for ADC interrupt

  // initialize INTC
  initIVPR();               // load IVPR
  initINTC();               // Initialize INTC for hardware vector mode
  INTC.PSR[62].R = 1;    // ADC_EOC interrupt
  enableIrq();             // set MSR EE bit to 1, CPR = 0


  // initialize the EMIOS.channel0 for PWM generation in OPWMT mode 100Hz
  Mcu_Emios_0_Init();

  // ADC 0 initialization
  Mcu_Adc_0_Init();

  // CTU initialization
  Mcu_Ctu_Init();

  // initialize DSPI's
  // DSPI's are part of peripheral set 2, which can run up to 64MHz max
  // but is running at 16MHz in this case
  initDSPI_0_serial();
```

```
while (1)
  {
   //Configure ASDR registers of DSPI A and B
  DSPI_0.ASDR.R = DSPIMasterTransmitBuffer[i];
  i++;

 // assuming the DSPI_0 SOUt is connected to DSPI_0 SIN pin
  // Wait the transmission to be completed
  while(DSPI_0.SR.B.TCF != 0x1);
  // Clear the TCF
  DSPI_0.SR.R = 0x80000000;

  if (i==2) {i=0;}
  }

}

__interrupt void ADC0ISR(void) {
  SIU.GPDO[68].R = !SIU.GPDO[68].R;
  ADC_0.ISR.B.EOCTU = 1;// Clear ADC CTU flag */
  ADC_0.CE0CFR0.B.EOC_CH0 = 1;
  //SIU.GPDO[68].R = !SIU.GPDO[68].R;
  INTC.EOIR_PRC0.R = 0;        /* restore end of interrupt register   */
}
```

A.3.2 function.c

```
/*************************************************************************/
/* FILE NAME: functions.c                  COPYRIGHT (c) Freescale 2010   */
/*                                                All Rights Reserved     */
/* DESCRIPTION:                                                           */
/* This is a basic collection of useful functions                         */
/*                                                                        */
/* SETUP:                                                                 */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach           */
/*  */
/*=======================================================================*/

#include "MPC5646x.h"
#include <ppc_ghs.h>

/*=======================================================================*/
/* DEFINES    */
/*=======================================================================*/

#define DEBOUNCEDELAYTIME 0xFFFF

#define DRUN_MODE 0x03
#define RUN0_MODE 0x04
#define STOP_MODE 0x0A
#define STANDBY_MODE 0x0D


// eMIOS PWM channel assignment
#define PWM1  0

// PWM 100Hz period
// PWM Duty cycle 0.5% step
#define DUTY_CYCLE_STEP_100   200
// PWM Start delay 0.5% step
#define START_DELAY_TIME_STEP_100   200
// PWM ADC trigger point 0.5% step
#define TRIGGER_ADC_STEP_100  200
// PWM period counter roll over value
#define COUNTER_PERIOD_100    40000
```

```
/*=========================================================================*/
/* VARIABLES  */
/*=========================================================================*/
long count;

// eMIOS channel control structure
typedef struct
{
vuint8_t dutyCycle;       // PWM Duty cycle (from 0 to 200 - 0.5% step)
vuint8_t startDelayTime;  // PWM shift to base (from 0 to 200 - 0.5% step)
vuint8_t triggerAdc;      // ADC trigger point (from 0 to 200 - 0.5% step)
}chPar;

// Channels parameters array
chPar ch[];

/*=========================================================================*/
/* FUNCTIONS  */
/*=========================================================================*/

 void initDSPI_0_serial(void) {

   //Program to enter into master DSI mode
  //-- Configure DSPIA as master
  DSPI_0.MCR.R = 0x90010001;
  // configure CTAR0
  // with 16Mhz input clock, BR=128, PBR=5, DBR=0
  // SCK baud rate = (fsys/PBR) * (1+DBR/BR) = 25kHz --> 40us
  // for 32Bit frame it will take ~1,28ms
  DSPI_0.CTAR[0].R = 0x780A7727;

  //Test: DSPI A transmission is started when ADSR is loaded
  //DSPI A - DSI: FMSZ[4], use DSPI_ASDR
  //transmit when CID, assert PCS0
  DSPI_0.DSICR.R = 0x40010001;
  // transmit continuously
  //DSPI_0.DSICR.R = 0x40000001;

  // enable DSPIs
  DSPI_0.MCR.B.HALT = 0x0;


  // DSPI_0 pin settings
  SIU.PCR[13].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SOUT output */
  SIU.PCR[12].R = 0x0103;        /* MPC56xxB: Config pad as DSPI_0 SIN input */
  SIU.PCR[14].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SCK output */
  SIU.PCR[15].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 PCS0 output */


  // DSPI serialized source select register
  DSPI_0.SSR.R = 0xFFFFFFFE; // take all bits except bit31 from the ASDR register

  // play with the DDIF interrupt flag
  // DSPI data Interrupt Mask register
  DSPI_0.DIMR.R = 0x00000001; // enable only bit31 of DSPI_0

  // DSPI deserialized data polarity interrupt register
  DSPI_0.DPIR.R = 0x00000001; // enable only bit31 of DSPI_0 triggers the flag when it is 1

}

void Mcu_Adc_0_Init(void)
{

   // Init ADC configuration registers
    ADC_0.MCR.B.ADCLKSEL      = 1;       // Set AD clk to ADCclk = 16MHz
    ADC_0.MCR.B.PWDN          = 0;       // Clear the Power down enable bit
    ADC_0.MCR.B.CTUEN         = 1;       // enable CTU
    ADC_0.MCR.B.MODE          = 0;      // Configure One Shot conversion
```

```
    ADC_0.MCR.B.OWREN         = 1;      // Overwrite newer results
    ADC_0.MCR.B.ACKO          = 1;      // Auto clock off enabled

 // Reset NCMR registers
    ADC_0.NCMR0.R = 0x00000000;
    ADC_0.NCMR1.R = 0x00000000;
    ADC_0.NCMR2.R = 0x00000000;
    // Configure ANP[0] to normal mode
    ADC_0.NCMR0.B.CH0 = 1;

    // Conversion timing registers
    // ANP[x], 1.126us @ 32MHz (Teval ~ 938ns, Tsample ~ 172ns)
    ADC_0.CTR0.R = 0x00008C06;

// enable the CTU interrupt
ADC_0.IMR.B.MSKEOCTU = 1;
ADC_0.CIMR0.B.CIM0 = 1;
}


void Mcu_Ctu_Init(void)
{
// EMIOS[0] triggers ADC ANP[0]
CTU.EVTCFGR[0].R = 0x00008000;
CTU.EVTCFGR[0].B.CHANNEL_VALUE = 0;

}

void Mcu_Emios_0_Channel_Config(vuint8_t i)
{
vuint32_t chDutyCycle, chStartDelay, chTriggerAdc, counterPeriod;
vuint16_t chDutyCycleStep, chStartDelayStep, chTriggerAdcStep;


// Load recalculation constants
chDutyCycleStep = START_DELAY_TIME_STEP_100;
chStartDelayStep = DUTY_CYCLE_STEP_100;
chTriggerAdcStep = TRIGGER_ADC_STEP_100;
counterPeriod = COUNTER_PERIOD_100;

    // Channel Start delay time
    // Recalculate PWM leading edge Start delay time
chStartDelay = (vuint32_t)(ch[i].startDelayTime * chStartDelayStep);
// Load PWM leading edge start delay time to A1
EMIOS_0.CH[i].CADR.R = chStartDelay;


    // Channel PWM Duty Cycle
// Recalculate Duty cycle
chDutyCycle = (vuint32_t)(ch[i].dutyCycle * chDutyCycleStep);
    // Load duty cycle to channel B2 register (transfered to B1 on counter A1 match)
if (chDutyCycle == counterPeriod)
{
// PWM Duty cycle 100%
EMIOS_0.CH[i].CBDR.R = chDutyCycle;
}
else if ((chDutyCycle + chStartDelay) > counterPeriod)
{
// PWM trailing edge after counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay - counterPeriod;
}
else
{
// PWM trailing edge before counter roll over
EMIOS_0.CH[i].CBDR.R = chDutyCycle + chStartDelay;
}


    // Channel ADC trigger point
// Recalculate Trigger ADC
```

```
chTriggerAdc = (vuint32_t)(ch[i].triggerAdc * chTriggerAdcStep);
// Load channel trigger ADC to A2 register
if ((chTriggerAdc + chStartDelay) > counterPeriod)
{
// Channel ADC trigger occurs after counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay - counterPeriod;
}
else
{
// Channel ADC trigger occurs before counter roll over
EMIOS_0.CH[i].ALTCADR.R = chTriggerAdc + chStartDelay;
}
}

void Mcu_Emios_0_Channel_Init(void)
{
// Channel PWM1
// Leading edge at begin of counting cycle
ch[PWM1].startDelayTime = 0;
// Duty cycle 50%
ch[PWM1].dutyCycle = 100;
// ADC trigger point 25%
ch[PWM1].triggerAdc = 50;

// Configure eMIOS channels PWM1
Mcu_Emios_0_Channel_Config(PWM1);
}


void Mcu_Emios_0_Init(void)
{
// Module Configuration register
// Global prescaler enable, divide ratio 4  --> 4MHz input clock for UC's
EMIOS_0.MCR.R = 0x04000300;

// Module Output Update Disable register
// Enable transfers (A2 -> A1, B2 -> B1) for all 28 channel registers
EMIOS_0.OUDR.R = 0x00000000;

    // Enable channels: 0, 23 (0 -> enable)
EMIOS_0.UCDIS.R = 0xFFFFFFFF;
EMIOS_0.UCDIS.B.CHDIS0 = 0;
EMIOS_0.UCDIS.B.CHDIS23 = 0;

// Channel 23 configuration - MC mode - counter bus A 100Hz
// Enable prescaler (divide ration 1 --> 4MHz), global clock, MC mode
EMIOS_0.CH[23].CCR.R = 0x02000310;
// Config A1: 4MHz/100Hz = 40000, counter bus A period = A1 + 1 => A1 = 39999
EMIOS_0.CH[23].CADR.R = COUNTER_PERIOD_100 - 1;

// PWM1 channel configuration - OPWMT mode
// Enable prescaler (divide ration 1), counter bus A clock, OPWMT mode
EMIOS_0.CH[PWM1].CCR.R = 0x02000026;
// A match comparator A sets output, while match B clears it.
EMIOS_0.CH[PWM1].CCR.B.EDPOL = 1;
// Flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.DMA   = 1;
// Enable the flag generate DMA request (CTU trigger)
EMIOS_0.CH[PWM1].CCR.B.FEN   = 1;

// Load PWM data into channels registers
Mcu_Emios_0_Channel_Init();

// Global time base enable
EMIOS_0.MCR.B.GTBE = 1;
}

void MODE_INIT(void)
{
   /* Enbable all peripheral clocks */
```

```
   CGM.SC_DC[0].B.DIV = 0x80;
   CGM.SC_DC[1].B.DIV = 0x80;
   CGM.SC_DC[2].B.DIV = 0x80;
   /* Setting RUN Configuration Register ME_RUN_PC[0] */
   ME.RUNPC[0].R=0x000000FE; /* Peripheral ON in every mode */

   /* Re-enter in DRUN mode to update */
   ME.MCTL.R = 0x30005AF0;       /* Mode & Key */
   ME.MCTL.R = 0x3000A50F;       /* Mode & Key */

}

void DISABLE_WATCHDOG()
{
  SWT.SR.R = 0x0000c520; /* key */
  SWT.SR.R = 0x0000d928; /* key */
  SWT.CR.R = 0xC000010A; /* disable WEN */
}

/*************************************************************************/
/* FUNCTION     : set_out_pin      */
/* PURPOSE      : Sets the corresponding pin to output         */
/*************************************************************************/
void set_GPIOout_pin(int PIN_NUMBER)
{
    /* Configure GPIO[PIN_NUMBER] for OUTPUT */
    SIU.PCR[PIN_NUMBER].R = 0x0200; /* PA = 00, OBE = 1, */
    /* SIU_GPDO[PIN_NUMBER] set PDO (PIN DATA OUT to 1) */
    SIU.GPDO[PIN_NUMBER].R = 0x1;
}

/*************************************************************************/
/* FUNCTION     : blink_pin          */
/* PURPOSE      : blinks with a pin the specified number of times with a  */
/*                specified delay      */
/*************************************************************************/
void blink_pin(int PIN_NUMBER, int count, int delay)
{
 int loop;
 int loop2;
for (loop = 1; loop <= count; loop++)
    {
      /* SIU_GPDO[PNI_NUMBER} set PDO (PIN DATA OUT to 0) */
      if (SIU.GPDO[PIN_NUMBER].R == 0x1) {
        SIU.GPDO[PIN_NUMBER].R = 0x0;
      }
      else {
    SIU.GPDO[PIN_NUMBER].R = 0x1;
      }
  for (loop2 = 1; loop2 <= delay; loop2++)
      {
    asm ("nop");
      }
    }
}
```

A.4 Multiple transfer operation — serial chaining

A.4.1 Main.c

```
/*************************************************************************/
/* FILE NAME: DSPI_SDR_ASDR_serial_chaining64.c   COPYRIGHT (c) Freescale 2010   */
/*                                                All Rights Reserved     */
/* DESCRIPTION:                                                           */
/* 64bit serialized SPI frame in DSI mode. Data is taken from the         */
/* DSPIMasterTransmitBuffer for DSPI0 and DSPI1                           */
/* The device is running on 16MHz IRC and the test was done on an EVB     */
/* connecting DSPI_1 out to DSPI_0 in                                     */
/* A 64bit SPI frame is always sent at data change                        */
/*                                                                        */
/* SETUP:                                                                 */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach           */
/*  */
/*************************************************************************/

#include "MPC5646x.h"

/*=======================================================================*/
/* Global variables  */
/*=======================================================================*/


const uint32_t  DSPIMasterTransmitBuffer[4] = {0x55555555,0x66666666, 0xAAAAAAAA,
0xBBBBBBBB};
uint64_t DSPI_frame;
uint32_t TransmissionOKCounter = 0, TransmissionErrorCounter = 0, DDIFFlagCounter = 0;


// Testcase is to loop DSPI_1 SOUT to DSPI_0 SIN on the EVB
// Remove jumper J8.4 from the pushbuttons to see a better signal on PE[3]

int main(int argc, char *argv[])
{

int i = 0, k = 0;

 DISABLE_WATCHDOG(); // watchdog is enabled by default
MODE_INIT();   // enable all peripherals in all modes with all peripheral clocks

  // initialize DSPI's
  // DSPI's are part of peripheral set 2, which can run up to 64MHz max
  // but is running at 16MHz in this case
  initDSPI_0_1_serial();



while (1)
  {
  //Configure ASDR registers of DSPI A and B
  DSPI_1.ASDR.R = DSPIMasterTransmitBuffer[i];
  i++;
  DSPI_0.ASDR.R = DSPIMasterTransmitBuffer[i];
  i++;


 // assuming the DSPI_2 SOUt is connected to DSPI_1 SIN pin
  // Wait the transmission to be completed
  while(DSPI_0.SR.B.TCF != 0x1);
  // Clear the TCF
  DSPI_0.SR.R = 0x80000000;
  DSPI_1.SR.R = 0x80000000;

  for (k=0;k<1000;k++);

  DSPI_frame = (((unsigned long long)DSPI_1.DDR.R) <<32) + DSPI_0.DDR.R;

  if ((DSPI_frame == (((unsigned long long)DSPIMasterTransmitBuffer[i-2]) <<32) +
```

```
DSPIMasterTransmitBuffer[i-1]))
    TransmissionOKCounter++;
  else
    TransmissionErrorCounter++;

  // DDIFFlagCounter should toggle every second transmission, so TransmissionOKCounter/2 =
DDIFFlagCounter
  if (DSPI_1.SR.B.DDIF) {DDIFFlagCounter++;DSPI_1.SR.B.DDIF=1;}



  if (i==4) {i=0;}
  }

}
```

A.4.2 function.c

```
/*******************************************************************************/
/* FILE NAME: functions.c                  COPYRIGHT (c) Freescale 2010   */
/*                                                 All Rights Reserved     */
/* DESCRIPTION:                                                            */
/* This is a basic collection of useful functions                          */
/*                                                                         */
/* SETUP:                                                                  */
/* Software tested on MPC564xB EVB using GHS 5.1.7 + Lauterbach            */
/*  */
/*=========================================================================*/

#include "MPC5646x.h"
#include <ppc_ghs.h>

/*=========================================================================*/
/* DEFINES    */
/*=========================================================================*/

#define DEBOUNCEDELAYTIME 0xFFFF

#define DRUN_MODE 0x03
#define RUN0_MODE 0x04
#define STOP_MODE 0x0A
#define STANDBY_MODE 0x0D

/*=========================================================================*/
/* PROTOTYPES  */
/*=========================================================================*/
long count;
/*=========================================================================*/
/* FUNCTIONS  */
/*=========================================================================*/


 void initDSPI_0_1_serial(void) {

   //Program to enter into master DSI mode
  //-- Configure DSPIA as master
  //-- Configure DSPIB as slave
  DSPI_0.MCR.R = 0x90010001;
  DSPI_1.MCR.R = 0x10010001;
  // configure CTAR0
  // with 16Mhz input clock, BR=128, PBR=5, DBR=0
  // SCK baud rate = (fsys/PBR) * (1+DBR/BR) = 25kHz --> 40us
  // for 64Bit frame it will take ~2,56ms
  DSPI_0.CTAR[0].R = 0x780A7727;
  DSPI_1.CTAR[1].R = 0xF80A7727; /* Configure CTAR1 for DSI slave mode */
```

```
  //Configuring the SIUL registers for serial chaining of DSPI0 and DSPI1
  SIU.DISR.R=0x01540000;


  //Test: DSPI A transmission is started when ADSR is loaded
  //DSPI A - DSI: MTOE, FMSZ[4], MTOCNT = 64, use DSPI_ASDR
  //transmit when CID, assert PCS0
  DSPI_0.DSICR.R = 0xFF090001;
  // transmit continuously
  //DSPI_0.DSICR.R = 0xFF080001;

  //DSPI B - DSI: MTOE, FMSZ[4], MTOCNT = 64, use DSPI_ASDR
  //generate trigger when CID, assert PCS0
  DSPI_1.DSICR.R = 0xFF090001;
  // generate trigger continuously
  //DSPI_1.DSICR.R = 0xFF080001;

  // enable DSPIs
  DSPI_0.MCR.B.HALT = 0x0;
  DSPI_1.MCR.B.HALT = 0x0;


  // DSPI_0 pin settings
  //SIU.PCR[13].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SOUT output */
  SIU.PCR[12].R = 0x0103;        /* MPC56xxB: Config pad as DSPI_0 SIN input */
  SIU.PCR[14].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 SCK output */
  SIU.PCR[15].R = 0x0604;        /* MPC56xxB: Config pad as DSPI_0 PCS0 output */

  // DSPI_1 pin settings
  //SIU.PCR[68].R = 0x0903;        /* MPC56xxB: Config pad as DSPI_1 SCK input */
  //SIU.PSMI[7].R = 1;             /* MPC56xxB: Select PCR 68 for DSPI_1 SCK input */
  //SIU.PCR[66].R = 0x0103;        /* MPC56xxB: Config pad as DSPI_1 SIN input */
  //SIU.PSMI[8].R = 1;             /* MPC56xxB: Select PCR 8 for DSPI_1 SIN input */
  SIU.PCR[67].R = 0x0A04;        /* MPC56xxB: Config pad as DSPI_1 SOUT output*/
  //SIU.PCR[69].R = 0x0903;        /* MPC56xxB: Config pad as DSPI_1 PCS0/SS input */
  //SIU.PSMI[9].R = 2;             /* MPC56xxB: Selec PCR 15 for DSPI_1 SS input */

  // play with the DDIF interrupt flag
  // DSPI data Interrupt Mask register
  DSPI_0.DIMR.R = 0x00000000; // disable all DSPI_0 flags
  DSPI_1.DIMR.R = 0x00000001; // enable only bit31 of DSPI_1

  // DSPI deserialized data polarity interrupt register
  DSPI_0.DPIR.R = 0x00000000; // all DSPI_0 bits=0 trigger the DSPI.SR[DDIF] flag
  DSPI_1.DPIR.R = 0x00000001; // enable only bit31 of DSPI_1 triggers the flag when it is 1

}

void MODE_INIT(void)
{
   /* Enbable all peripheral clocks */
   CGM.SC_DC[0].B.DIV = 0x80;
   CGM.SC_DC[1].B.DIV = 0x80;
   CGM.SC_DC[2].B.DIV = 0x80;
   /* Setting RUN Configuration Register ME_RUN_PC[0] */
   ME.RUNPC[0].R=0x000000FE; /* Peripheral ON in every mode */

   /* Re-enter in DRUN mode to update */
   ME.MCTL.R = 0x30005AF0;       /* Mode & Key */
   ME.MCTL.R = 0x3000A50F;       /* Mode & Key */

}

void DISABLE_WATCHDOG()
{
  SWT.SR.R = 0x0000c520; /* key */
  SWT.SR.R = 0x0000d928; /* key */
  SWT.CR.R = 0xC000010A; /* disable WEN */
}
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4238
Rev. 0, Jan 2011

freescale semiconductor