FlexBus examples

By: Technical Information Center

Content

1 Introduction

FlexBus provides an easy way to extended memory or to communicate with some peripherals that features External Bus Interfaces (EBI).

The following document is intended to demonstrate FlexBus usage with the K60 and K70 tower modules. It contains several examples featuring FlexBus implementations with MQX and bareboard projects using the TWR-MEM and the TWR-LCD.

The bareboard examples are implemented in CW10.6 and KDS1.1, for the MQX examples MQX4.1 is used. Please refer to: MQX Labs, for an introduction to MQX and instructions for the installation and building of the libraries.

2 Bareboard projects

The following examples use FlexBus for:

- Simple Read/Write tasks to an external memory
- 16-byte burst transfers using DMA (DMA initialization examples)
- Parallel bus interface with a graphical LCD.

2.1 FlexBus Read/Write K60/K70

This example sets an external memory via FlexBus it works on either TWR-K60 or TWR-K70 platform. In order to initialize the corresponding FlexBus registers the following steps are performed.

1. Set the base address in the corresponding chip select address register, Chip Select 0 is being used in this example. See the chip memory map (Chapter 4 in the reference manual of both K60 and K70) for the applicable FlexBus "expansion" address range for which the chip-selects can be active.

0x6000_0000–0x7FFF_FFFF	FlexBus (External Memory - Write-back)	All masters
0x8000_0000–0x9FFF_FFFF	FlexBus (External Memory - Write-through)	All masters
0xA000_0000–0xDFFF_FFFF	FlexBus (External Peripheral - Not executable)	All masters

Figure 1 - K60 FlexBus memory map

0x6000_0000–0x6FFF_FFFF	Flexbus (External memory - Write-back)	All masters	S4
0x9000_0000–0x9FFF_FFFF	FlexBus (External memory - Write-through)	All masters	S4
0xA000_0000–0xDFFF_FFFF	FlexBus (External peripheral - not executable)	All masters	S4

Figure 2 - K70 FlexBus memory map

FlexBus (External memory - Write-back)
Here all the data is being written to the cache only and it will not reach the storage device until cache buffer has been flushed.

FlexBus (External memory - Write-through)
Here all the data is being written synchronously both to the cache and to the storage device.

(External peripheral - not executable)
This is meant to write to external devices memories.

In this example a macro definition is being used to set the base address of the chip select. This is achieved in both CodeWarrior 10.6 and KDS 1.1 by the following code:

```
#define MRAM_START_ADDRESS (*(volatile unsigned char*)(0x60000000))
```

The CodeWarrior 10.6 code that initializes this register is the following:

```
FB_CSAR0 = (unsigned int)&MRAM_START_ADDRESS;
```

The KDS 1.1 code that initializes this register is the following:

```
FB->CS[0].CSAR = (unsigned int)&MRAM_START_ADDRESS;
```

2. Set the Chip Select Control Register (FB_CSCRn) which controls the auto-acknowledge, address setup and hold times, port size, burst capability and number of wait states for the associated chip select. In this example the following configuration is being used:
   - 8-bit port size.
   - Auto-acknowledge.
   - Assert chip select on second clock edge after address is asserted.
   - One wait state (dependent on the bus speed).

   The CodeWarrior 10.6 code that initializes this register is the following:

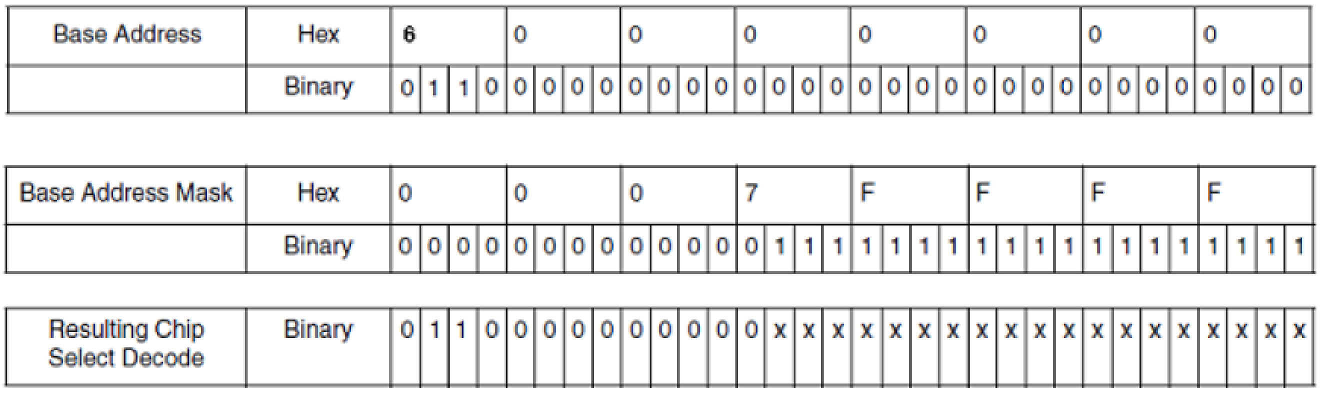
```
FB_CSCR0  =   FB_CSCR_PS(1)      // 8-bit port
        | FB_CSCR_AA_MASK    // auto-acknowledge
        | FB_CSCR_ASET(0x1)  // assert chip select on second clock edge
        | FB_CSCR_WS(0x1)    // 1 wait state
       ;
```

   The KDS 1.1 code that initializes this register is the following:

```
FB->CS[0].CSCR  =   FB_CSCR_PS(1)    // 8-bit port
            | FB_CSCR_AA_MASK    // auto-acknowledge
            | FB_CSCR_ASET(0x1)  // assert chip select on second clock edge
            | FB_CSCR_WS(0x1)    // 1 wait state
       ;
```

3. Define chip select's memory block size and validates the corresponding registers for the chip select by writing to the Chip Select Mask Register (FB_CSMRn).

   Setting CSMR[BAM] to 0x0007 results in a base address mask of 0x0007FFFF. With base address 0x60000000 and base address mask 0x0007FFFF, the chip select is active for address range 0x60000000 – 0x6007FFFF.
   In binary looks like the following:



Base Address	Hex	6				0				0				0				0				0				0				0			
	Binary	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Base Address Mask	Hex	0				0				0				7				F				F				F				F			
	Binary	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Resulting Chip Select Decode	Binary	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 3 - Base Address Mask explanation

In this example a base address mask for 512Kbytes space will be set as well as the chip select enable signal.

The CodeWarrior 10.6 code to initialize the register is the following:

```
FB_CSMR0 = FB_CSMR_BAM(0x7)  //Set base address mask for 512K address space
          |FB_CSMR_V_MASK    //Enable cs signal
          ;
```

The KDS 1.1 code to initialize the register is the following:

```
FB->CS[0].CSMR = FB_CSMR_BAM(0x7)  //Set base address mask for 512K address space
               | FB_CSMR_V_MASK    //Enable cs signal
               ;
```

4. Set the FlexBus clock divider at the corresponding field in the System Clock Divider Register 1. In this example a clock divider of 3 is being used, since the MCG is running in FEI mode (out of reset) and it runs at 20 MHz the FlexBus clock frequency is of 6.6 MHz.
   The CodeWarrior 10.6 code to initialize the register is the following:

```
SIM_CLKDIV1 |= SIM_CLKDIV1_OUTDIV3(0x3);
```

The KDS 1.1 code to initialize the register is the following:

```
SIM->CLKDIV1 |= SIM_CLKDIV1_OUTDIV3(0x3);
```

5. Configure and enable the necessary pins (see Appendix in this document).

Now that the initialization is complete the program performs reads and writes to the TWR-MEM module through the FlexBus. The results are printed on the console in the debugger platform (For using console in KDS please follow this tutorial to enable semihosting).
The following output must be observed in the console for both CodeWarrior 10.6 and KDS 1.1 independently of the used platform (K70 or K60):

```
Initializing the FlexBus

FB_CSCR0 is 00100540

FB_CSMR0 is 00070001

FB_CSAR0 is 60000000

FB_CSPMCR is 00000000

SIM_CLKDIV1 is 00310000

Testing 8-bit write/reads
ADDR: 0x60000000 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000001 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000002 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000003 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000004 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000005 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000006 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000007 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000008 WRITE: 0xa5 READ: 0xa5
ADDR: 0x60000009 WRITE: 0xa5 READ: 0xa5
ADDR: 0x6000000a WRITE: 0xa5 READ: 0xa5
ADDR: 0x6000000b WRITE: 0xa5 READ: 0xa5
ADDR: 0x6000000c WRITE: 0xa5 READ: 0xa5
ADDR: 0x6000000d WRITE: 0xa5 READ: 0xa5
ADDR: 0x6000000e WRITE: 0xa5 READ: 0xa5

Testing 16-bit write/reads
ADDR: 0x60000010 WRITE: 0x1234 READ: 0x1234
ADDR: 0x60000012 WRITE: 0x1234 READ: 0x1234
ADDR: 0x60000014 WRITE: 0x1234 READ: 0x1234
ADDR: 0x60000016 WRITE: 0x1234 READ: 0x1234
ADDR: 0x60000018 WRITE: 0x1234 READ: 0x1234
ADDR: 0x6000001a WRITE: 0x1234 READ: 0x1234
ADDR: 0x6000001c WRITE: 0x1234 READ: 0x1234
ADDR: 0x6000001e WRITE: 0x1234 READ: 0x1234

Testing 32-bit write/reads
ADDR: 0x60000020 WRITE: 0x87654321 READ: 0x87654321
ADDR: 0x60000024 WRITE: 0x87654321 READ: 0x87654321
ADDR: 0x60000028 WRITE: 0x87654321 READ: 0x87654321
ADDR: 0x6000002c WRITE: 0x87654321 READ: 0x87654321
```

2.2 16-byte FlexBus transfer with DMA K60/K70

In the following example the TWR-MEM module is being used. For examples and an explanation of the DMA module configuration please refer to (DMA initialization examples).

In this example the DMA module is being used in conjunction with the FlexBus interface to perform 16-byte transfers. The eDMA module is being configured to transfer data from the RAM_START_ADDRESS (0x2000_0000) to the external memory (0x6000_0000) via FlexBus. Each transfer is set to 16-bytes in burst mode, unfortunately the TWR-MEM module does not support burst transfers; therefore this example is just a proof of concept.

The following code is being used to initialize the DMA module, in this example channel 0 is used to perform two 16-byte transfers.

1. First the DMA channel and Enable Request register for the corresponding channel must be enabled, this is carried out by the following code:

```
/*Enable DMA MUX ch 0*/
DMAMUX_CHCFG0 |= DMAMUX_CHCFG_ENBL_MASK;
/*S*/
DMA_ERQ |= DMA_ERQ_ERQ0_MASK;
```

2. Source and destination addresses must be declared, the following code sets both addresses:

```
/* Set the Source Address*/
DMA_TCD0_SADDR = (uint32_t)(&RAM_START_ADDRESS);
/* Destination address */
DMA_TCD0_DADDR = (uint32_t)(&MRAM_START_ADDRESS);
```

3. Disable source address modulo and destination address modulo and set source data transfer size and destination data transfer size:

```
/*Modulo off and port sizes*/
DMA_TCD0_ATTR = DMA_ATTR_SSIZE(4)|DMA_ATTR_SMOD(0)|DMA_ATTR_DSIZE(4)|DMA_ATTR_DMOD(0);
```

4. Set source and destination offsets (once a transfer has been completed this offsets are used to adjust source and destination addresses)

```
/* Source offset*/
DMA_TCD0_SOFF = 0x10; // 16 bytes
/* Destination offset*/
DMA_TCD0_DOFF = 0x10; //16 bytes
```

The FlexBus initialization remains almost the same as in Example 2.1 with the only exception of the Chip Select Control Register (CSCR) and the Chip Select Port Multiplexing Control Register (CSPMCR). In this example the Burst-Write and Burst-Read Enable fields are being set in the CSCR as well as the Transfer size signals (FB_TSIZ1 – FB_TSIZ0) in the CSPMCR.

The Chip Select Control Register configuration is initialized as follows:

```
      FB_CSCR0  =   FB_CSCR_PS(1)      // 8-bit port
                  | FB_CSCR_AA_MASK    // auto-acknowledge
                  | FB_CSCR_WS(2)      // 2 wait state
                  | FB_CSCR_BSTW_MASK  // Burst write enable
                  | FB_CSCR_BSTR_MASK  // Burst read enable
                  ;
```

The Chip Select Port Multiplexing Control Register is initialized as follows:

```
       FB_CSPMCR = (FB_CSPMCR & ~(FB_CSPMCR_GROUP2_MASK | FB_CSPMCR_GROUP3_MASK |
                  FB_CSPMCR_GROUP4_MASK))
                  | FB_CSPMCR_GROUP2(1)
                  | FB_CSPMCR_GROUP3(1);
```

In this example the FlexBus module is initialized and the flexbus_test routine used in Example 2.1 performs simple Read/Write tasks but this time in burst mode, once the test routine has finished a DMA request is asserted via software, this will start the 16-byte in burst mode transfer. The the following code is being used to assert the DMA request via software:

```
      DMA_SSRT = DMA_SSRT_SSRT(0);
```

In this example Burst Reads and Writes are enabled, the main difference with respect Example 2.1, in which burst mode is disabled, is the Chip Select behavior and how the data and addresses are asserted. If burst capability is disabled every time a byte has been transferred, The Chip Select is disabled (change state to logical 1) and the address is sent, if another byte is still pending the chip select is enabled (change state to logical 0) and the data is sent this process continues until all of the data has been transferred . The process can be summarized as follows and can be seen in Figure 4:

<address><data><address><data> <address><data> … <address><data>

With burst enabled the address is asserted just at the beginning of the transfer, following it all the data is sent and Chip Select remains enabled throughout the whole transfer. The process can be summarized as follows and can be seen in Figure 5:

<address><data><data><data><data><data> … <data>

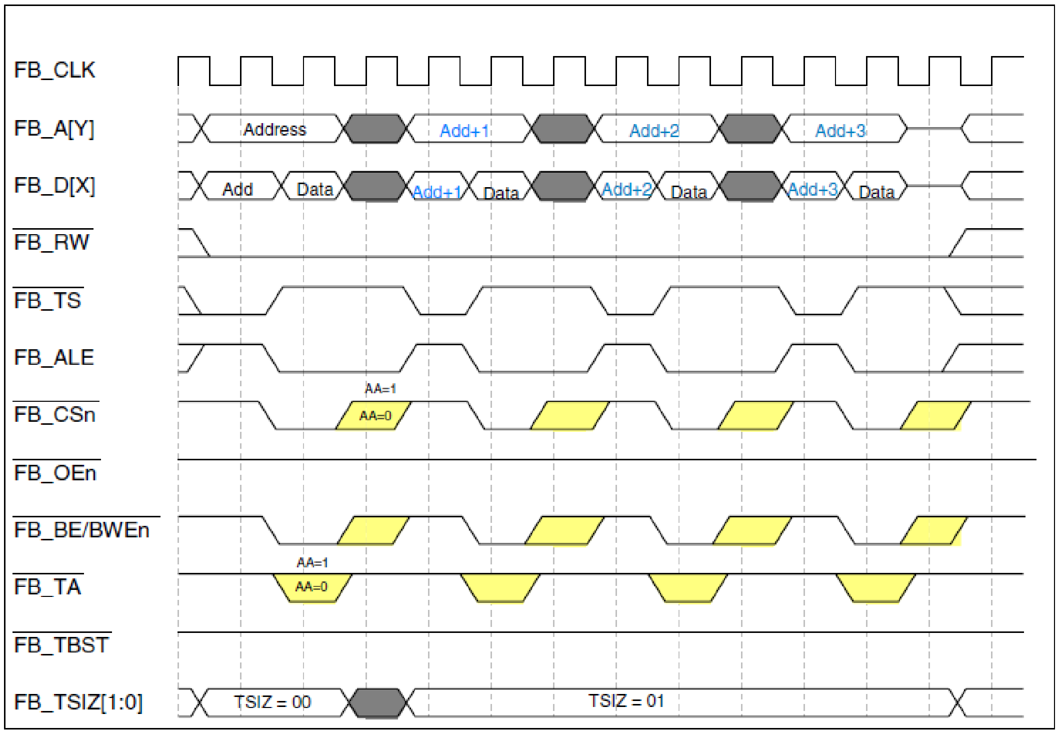


Figure 4 - Functional description without burst

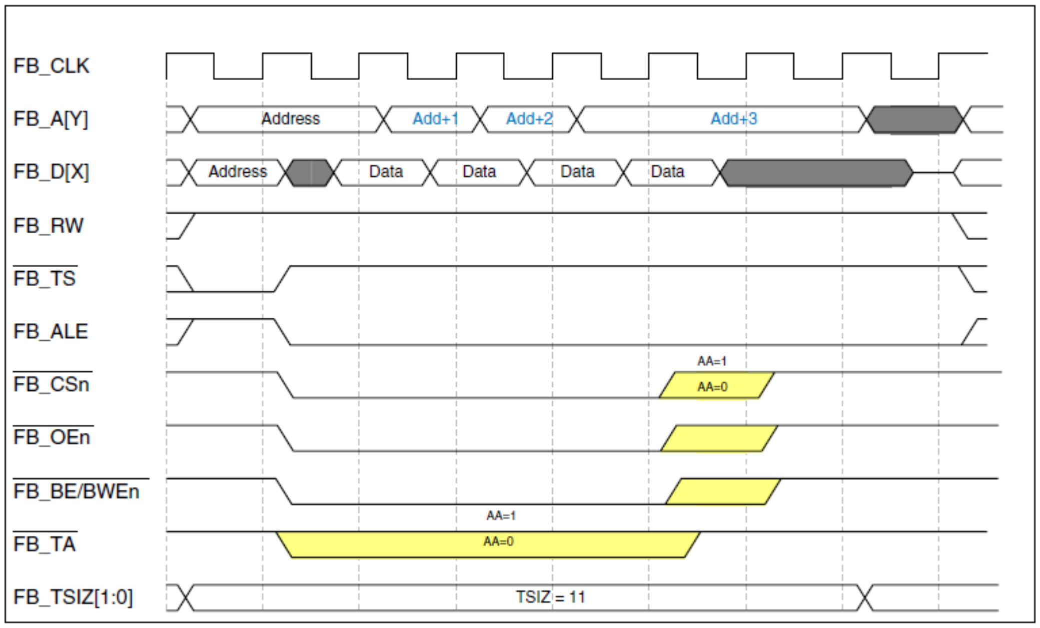


Figure 5 - Functional description with burst

In the following images the functional mechanism can be distinguished from the transfers made with burst enabled and burst disabled, Channel 2 is the FlexBus clock signal and Channel 3 is the Chip Select 0. First a Write is being carried out immediately followed by a Read:

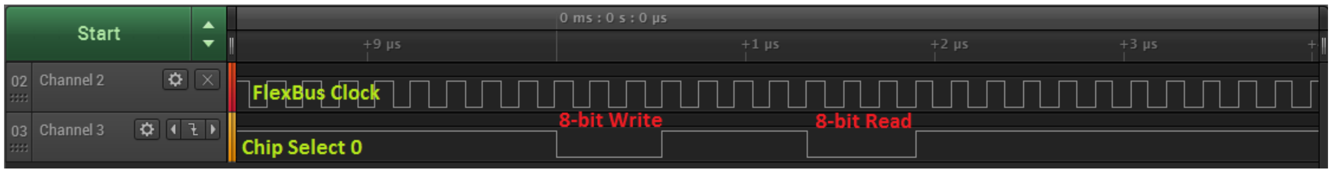


Figure 6- 8-bit transfer without burst

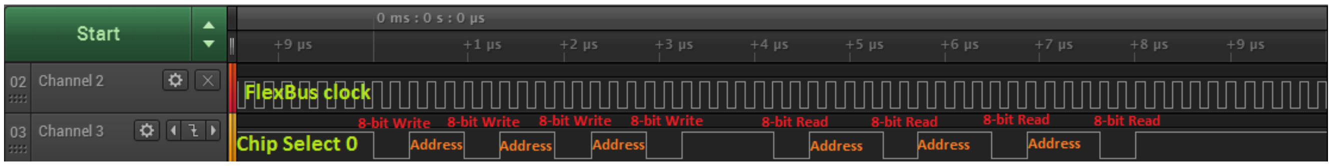


Figure 7 - 16-Bit transfer without burst

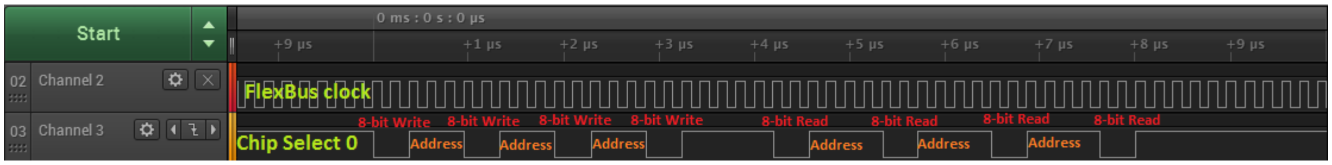


Figure 8 - 32-bit transfer without burst

Hardware only supports 8-bit and 32-bit transfers this is why 16-bit and 32-bit transfers look identical in Figure 7 and Figure 8. Nevertheless Software is capable of handling 16-bit words.

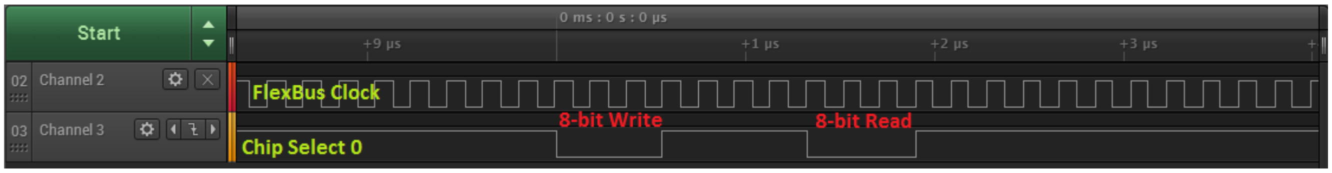


Figure 9 - 8-bit transfer burst enabled

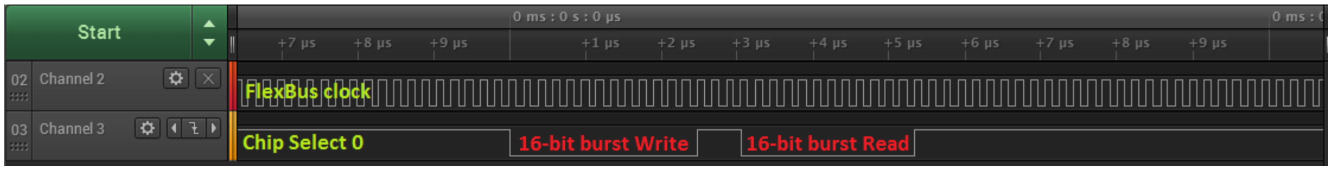


Figure 10 - 16-bit transfer burst enabled

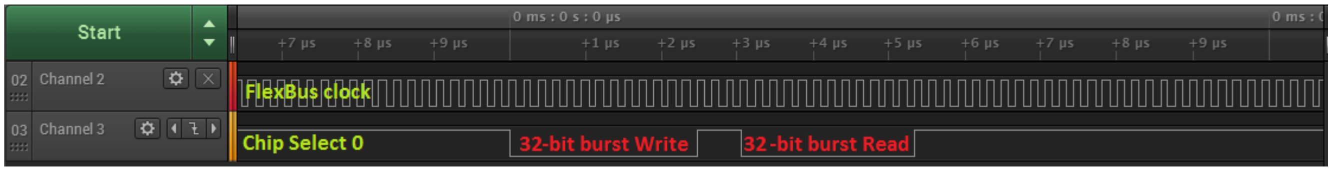


Figure 11 - 32-bit transfer burst enabled

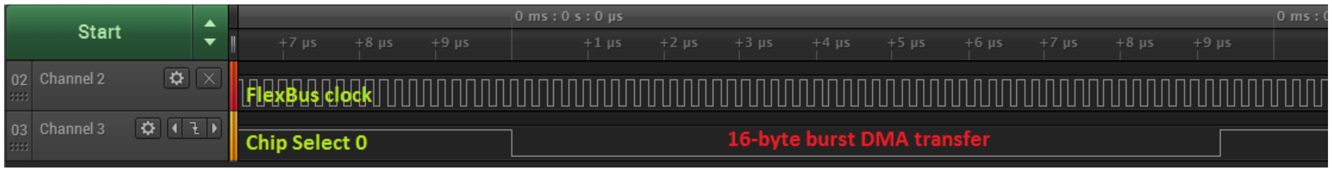


Figure 12 - 16 byte transfer burst enabled

As can be seen in Figure 9 and Figure 6, with 8-bit transfers the mechanism remains the same since just a “pack” of data is been sent. With 16-bit, 32-bit and 16-byte transfers it can be seen that the mechanism differs as expected the Chip Select remains enabled throughout the entire transfer.

2.3 FlexBus parallel bus interface with Graphical LCD K60

The TWR-LCD can work on either SPI communication or via an External Bus Interface (EBI), in this example it will be used in conjunction with the FlexBus interface of the TWR-K60. (For further information please refer to: *Display (eGUI) using TWR-LCD*, *Using Freescale eGUI with TWR-LCD on MCF51MM Family*)

Functional description:

1. MCG configuration at 96MHz in PEE mode (PLL Engaged External) with a divider for the FlexBus of 2 (FlexBus = Sysclk/2 = 48 MHz).
2. The FlexBus is configured as follows:

```
   #define ALT5 (PORT_PCR_MUX(5)|PORT_PCR_DSE_MASK) // Alternative function 5 = FB enable
   #define FLEX_CLK_INIT (SIM_CLKDIV1 |= SIM_CLKDIV1_OUTDIV3(1))//FlexBus=Sysclk/2=~48MHz

   #define DISPLAY_MCU_USER_INIT SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK| SIM_SCGC5_PORTB_MASK|
    SIM_SCGC5_PORTC_MASK | SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK;\
    PORTC_PCR0=ALT5; PORTC_PCR1=ALT5; PORTC_PCR2=ALT5; PORTC_PCR3=ALT5; PORTC_PCR4=ALT5;
    PORTC_PCR5=ALT5; PORTC_PCR6=ALT5; PORTC_PCR7=ALT5; PORTC_PCR8=ALT5; PORTC_PCR9=ALT5;
   PORTC_PCR10=ALT5; PORTC_PCR11=ALT5;\
   PORTD_PCR1=ALT5; PORTD_PCR2=ALT5; PORTD_PCR3=ALT5; PORTD_PCR4=ALT5; PORTD_PCR5=ALT5;
   PORTD_PCR6=ALT5; PORTB_PCR17=ALT5; PORTB_PCR18=ALT5;\
   FLEX_CLK_INIT;SIM_SOPT2 |= SIM_SOPT2_FBSL(3); SIM_SCGC7 |= SIM_SCGC7_FLEXBUS_MASK;
   /*PTC0 PTC1 PTC2 PTC3 PTC4 PTC5 PTC6 PTC7 PTC8 PTC9 PTC10 PTC11
      14    13   12  CLK  11   10    9    8     7    6    5     RW
   PTD1 PTD2 PTD3 PTD4 PTD5 PTD6 PTB17 PTB18
   CS    4     3    2    1   0    16D/C    15
                                           */
   #define FLEX_BASE_ADDRESS  0x60010000
   #define FLEX_DC_ADDRESS    0x60000000
   #define FLEX_ADRESS_MASK   0x00010000
   #define FLEX_CS 0
   #define CSCR_RESET 0x00000000
   // Kinetis Flexbus Register Macro redefinitions
   #define FLEX_CSAR FB_CSAR0
   #define FLEX_CSMR FB_CSMR0
   #define FLEX_CSCR FB_CSCR0
   // MUX mode + Wait States
   #define FLEX_CSCR_MUX_MASK  (FB_CSCR_BLS_MASK | CSCR_RESET)
   #define FLEX_CSMR_V_MASK    FB_CSMR_V_MASK
   #define FLEX_CSCR_AA_MASK   FB_CSCR_AA_MASK
   #define FLEX_CSCR_PS1_MASK  (FB_CSCR_PS(2))

   void vfnInitFlexBus(void){
      DISPLAY_MCU_USER_INIT

      FLEX_CSAR = FLEX_DC_ADDRESS; // CS0 base address
      FLEX_CSMR = FLEX_ADRESS_MASK | FLEX_CSMR_V_MASK; // The address range is set to
128K because the DC signal is connected on address wire
      FLEX_CSCR = FLEX_CSCR_MUX_MASK | FLEX_CSCR_AA_MASK | FLEX_CSCR_PS1_MASK; // FlexBus
setup as fast as possible in multiplexed mode
   }
```

After building and running the example the Freescale logo must be visible in your screen.

3 MQX projects

The following examples use MQX to perform:

- Reads/Writes of an external memory.
- RAM Memory extension.

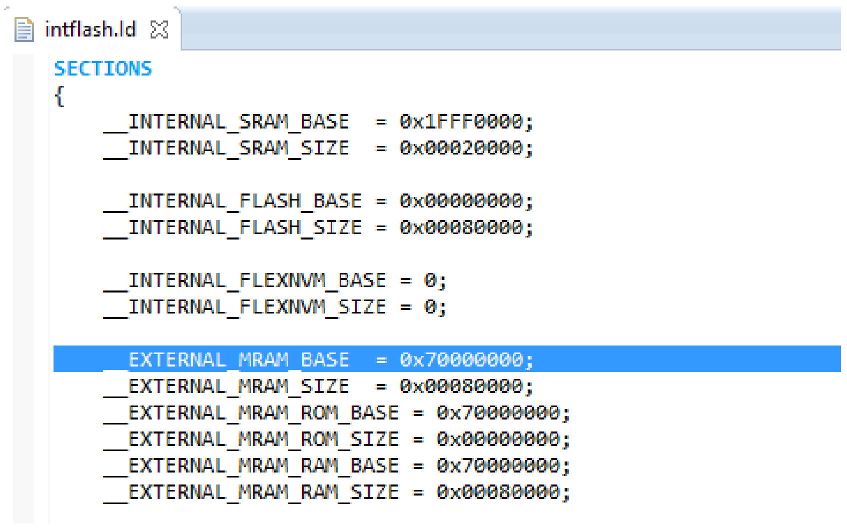
3.1 FlexBus Read/Write K60/K70

This example performs the same task that example 2.1 but it uses MQX. MQX provides its own FlexBus initialization which can be modified to fit your application needs or overwritten by some initialization routine as that used in example 2.1. If any modification is required the memory address must be verified so that it is consistent with the memory address of the corresponding chip select. This can be achieved by looking at the linker files generated by MQX at the library compilation stage. The location of these files is the following:

TWR-K60D100 -> C:\Freescale\Freescale_MQX_4_1\lib\twrk60d100m.cw10gcc\debug\bsp

TWR-K70F12 -> C:\Freescale\Freescale_MQX_4_1\lib\twrk70f120m.cw10gcc\debug\bsp

Look for the file "intflash.ld" and simply drag and drop it to the CodeWarrior environment to look at its content. The declaration of the memory start address must now be visible as "__EXTERNAL_MRAM_BASE" modify the declaration of "MRAM_START_ADDRESS" so that it matches the one declared in the linker file "intflash.ld".



```
SECTIONS
{
    __INTERNAL_SRAM_BASE  = 0x1FFF0000;
    __INTERNAL_SRAM_SIZE  = 0x00020000;

    __INTERNAL_FLASH_BASE = 0x00000000;
    __INTERNAL_FLASH_SIZE = 0x00080000;

    __INTERNAL_FLEXNVM_BASE = 0;
    __INTERNAL_FLEXNVM_SIZE = 0;

    __EXTERNAL_MRAM_BASE  = 0x70000000;
    __EXTERNAL_MRAM_SIZE  = 0x00080000;
    __EXTERNAL_MRAM_ROM_BASE = 0x70000000;
    __EXTERNAL_MRAM_ROM_SIZE = 0x00000000;
    __EXTERNAL_MRAM_RAM_BASE = 0x70000000;
    __EXTERNAL_MRAM_RAM_SIZE = 0x00080000;
```

NOTE: The "intflash.ld" file can be modified to change the address, nevertheless if the MQX BSP library is recompiled the "intflash.ld" will be overwritten and the modifications will be lost. This is due to the fact that any MQX application uses a copy of the original linker file, if a permanent change is desired the linker file used at the moment of building the MQX libraries must be modified in this example the first approach is being used due to the required time to re-built the libraries and the risk of permanently modifying some features.

The I/O default channel output is declared at the moment of building the MQX libraries (UARTx, Console).

3.2 FlexBus extended memory K60/K70

The following example extends the RAM memory so that whenever the MQX kernel runs out of internal RAM to place tasks' stack, an external memory can be used. In order to use the function _mem_extend() in the MQX environment the "user_config.h" file, which can be found in the User_Config folder of the MQX libraries (refer to Lab 1 Building MQX libraries), must be modified by adding the following preprocessor declarations:

```
#define MQX_USE_MEM 1 //Enable to use _mem_extend
#define MQX_USE_LWMEM 0
#define MQX_USE_LWMEM_ALLOCATOR 0
```

Once the file has been modified the BSP and PSP libraries must be built again.
With the libraries already modified and built the function _mem_extend() is now available. The prototype of this function is the following:

```
_mqx_uint  _mem_extend( void * start_of_pool, _mem_size size)
```

Where:
start_of_pool [IN] — Pointer to the start of the memory to add
size [IN] — Number of single-addressable units to add

The CodeWarrior code used in this example to add memory is the following:

```
result = _mem_extend((void *)0x70000000,0x80000);
      if (result == MQX_OK){
             printf("\nMem extend Succeed\n");
      }
      else
      {
             printf("\nMem extend failed. Error: 0x%X\n", result);
      }
```

If the MQX configuration will be used, the address declared in "intflash.ld" must match that one used at the calling of _mem_extend().
In this example three tasks are declared with a stack size of 30,000 each, so that if no memory is added the processor will run out of memory to store the tasks.
In order to visualize the stack usage of the system the following steps must be followed:

1. Configure the debugger so that it has access to the additional memory.
    a. Open the .mem file at the debugger folder in your project

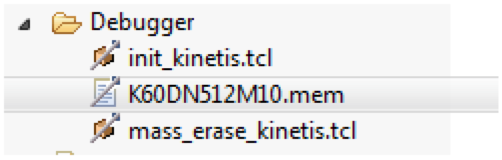


Figure 13- mem file location

    b. Modify the external memory range in the following manner:

```
range     0x60000000 0xDFFFFFFF 4 ReadWrite   // FlexBus for external memory
```

2. Start a debugging session.

3. Select MQX->Stack Usage as in the following figure.
4. Select MQX->Task Summary.

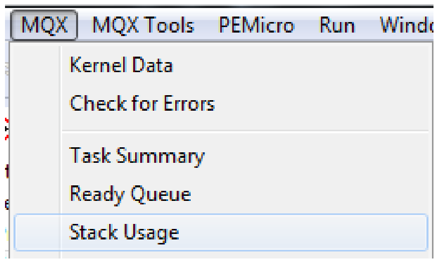
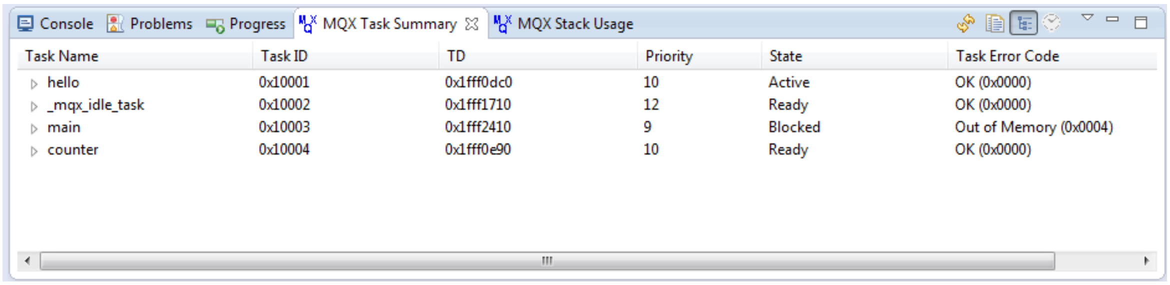


Figure 14 - Stack Usage and Task Summary consoles

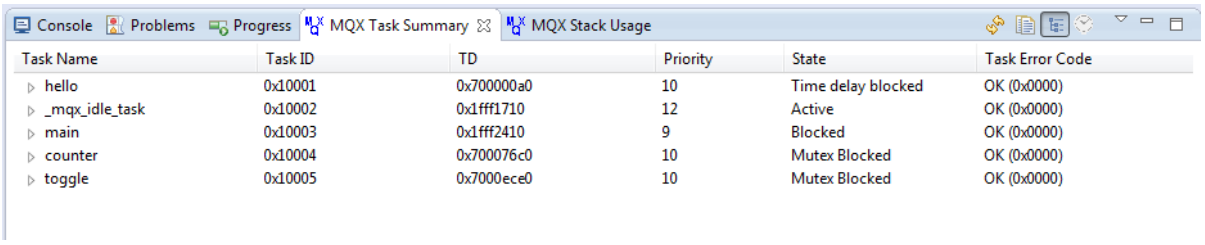
It can be easily verified that if the example is run without adding memory the processor runs out of memory as in the following image.



Task Name	Task ID	TD	Priority	State	Task Error Code
hello	0x10001	0x1fff0dc0	10	Active	OK (0x0000)
_mqx_idle_task	0x10002	0x1fff1710	12	Ready	OK (0x0000)
main	0x10003	0x1fff2410	9	Blocked	Out of Memory (0x0004)
counter	0x10004	0x1fff0e90	10	Ready	OK (0x0000)

Figure 15 - Without additional memory

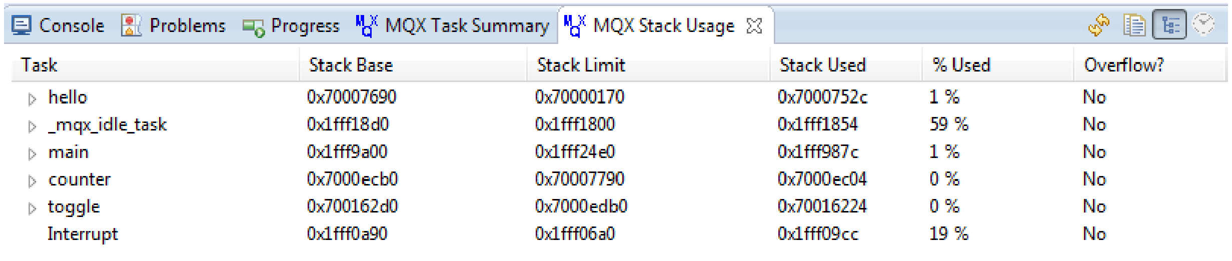
In the following example memory was added and it can be seen at everything is running as it should.



Task Name	Task ID	TD	Priority	State	Task Error Code
hello	0x10001	0x700000a0	10	Time delay blocked	OK (0x0000)
_mqx_idle_task	0x10002	0x1fff1710	12	Active	OK (0x0000)
main	0x10003	0x1fff2410	9	Blocked	OK (0x0000)
counter	0x10004	0x700076c0	10	Mutex Blocked	OK (0x0000)
toggle	0x10005	0x7000ece0	10	Mutex Blocked	OK (0x0000)

Figure 16 - Extended memory

In the following image it can be seen how now the processor uses the extended memory (0x7000 0000) to store the tasks.



Task	Stack Base	Stack Limit	Stack Used	% Used	Overflow?
hello	0x70007690	0x70000170	0x7000752c	1 %	No
_mqx_idle_task	0x1fff18d0	0x1fff1800	0x1fff1854	59 %	No
main	0x1fff9a00	0x1fff24e0	0x1fff987c	1 %	No
counter	0x7000ecb0	0x70007790	0x7000ec04	0 %	No
toggle	0x700162d0	0x7000edb0	0x70016224	0 %	No
Interrupt	0x1fff0a90	0x1fff06a0	0x1fff09cc	19 %	No

Figure 17- Stack usage with memory extended

4 Appendix

Pin configuration (Reference Manual chapters 10.3.1 K60 Signal Multiplexing and Pin Assignments and 10.3.1 K70 Signal Multiplexing and Pin Assignments):

The CodeWarrior 10.6 code used in this is example is the following:

```
 // Enable clocks
SIM_SCGC5 = SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTC_MASK |
SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK;

  //address
 PORTB_PCR11 = PORT_PCR_MUX(5);           //  fb_ad[18]
 PORTB_PCR16 = PORT_PCR_MUX(5);           //  fb_ad[17]
 PORTB_PCR17 = PORT_PCR_MUX(5);           //  fb_ad[16]
 PORTB_PCR18 = PORT_PCR_MUX(5);           //  fb_ad[15]
 PORTC_PCR0  = PORT_PCR_MUX(5);           //  fb_ad[14]
 PORTC_PCR1  = PORT_PCR_MUX(5);           //  fb_ad[13]
 PORTC_PCR2  = PORT_PCR_MUX(5);           //  fb_ad[12]
 PORTC_PCR4  = PORT_PCR_MUX(5);           //  fb_ad[11]
 PORTC_PCR5  = PORT_PCR_MUX(5);           //  fb_ad[10]
 PORTC_PCR6  = PORT_PCR_MUX(5);           //  fb_ad[9]
 PORTC_PCR7  = PORT_PCR_MUX(5);           //  fb_ad[8]
 PORTC_PCR8  = PORT_PCR_MUX(5);           //  fb_ad[7]
 PORTC_PCR9  = PORT_PCR_MUX(5);           //  fb_ad[6]
 PORTC_PCR10 = PORT_PCR_MUX(5);           //  fb_ad[5]
 PORTD_PCR2  = PORT_PCR_MUX(5);           //  fb_ad[4]
 PORTD_PCR3  = PORT_PCR_MUX(5);           //  fb_ad[3]
 PORTD_PCR4  = PORT_PCR_MUX(5);           //  fb_ad[2]
 PORTD_PCR5  = PORT_PCR_MUX(5);           //  fb_ad[1]
 PORTD_PCR6  = PORT_PCR_MUX(5);           //  fb_ad[0]

 //data
 PORTB_PCR20 = PORT_PCR_MUX(5);           //  fb_ad[31] used as d[7]
 PORTB_PCR21 = PORT_PCR_MUX(5);           //  fb_ad[30] used as d[6]
 PORTB_PCR22 = PORT_PCR_MUX(5);           //  fb_ad[29] used as d[5]
 PORTB_PCR23 = PORT_PCR_MUX(5);           //  fb_ad[28] used as d[4]
 PORTC_PCR12 = PORT_PCR_MUX(5);           //  fb_ad[27] used as d[3]
 PORTC_PCR13 = PORT_PCR_MUX(5);           //  fb_ad[26] used as d[2]
 PORTC_PCR14 = PORT_PCR_MUX(5);           //  fb_ad[25] used as d[1]
 PORTC_PCR15 = PORT_PCR_MUX(5);           //  fb_ad[24] used as d[0]

 //control signals
 PORTB_PCR19 = PORT_PCR_MUX(5);          // fb_oe_b
 PORTC_PCR11 = PORT_PCR_MUX(5);          // fb_rw_b
 PORTD_PCR1  = PORT_PCR_MUX(5);          // fb_cs0_b
  PORTD_PCR0  = PORT_PCR_MUX(5);          // fb_ale
```

The KDS 1.1 code used in this example is the following:

```
 // Enable clocks
SIM->SCGC5 = SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTC_MASK |
SIM_SCGC5_PORTD_MASK | SIM_SCGC5_PORTE_MASK;
```

```
//address
PORTB->PCR[11] = PORT_PCR_MUX(5);           //  fb_ad[18]
PORTB->PCR[16] = PORT_PCR_MUX(5);           //  fb_ad[17]
PORTB->PCR[17] = PORT_PCR_MUX(5);           //  fb_ad[16]
PORTB->PCR[18] = PORT_PCR_MUX(5);           //  fb_ad[15]
PORTC->PCR[0]  = PORT_PCR_MUX(5);           //  fb_ad[14]
PORTC->PCR[1]  = PORT_PCR_MUX(5);           //  fb_ad[13]
PORTC->PCR[2]  = PORT_PCR_MUX(5);           //  fb_ad[12]
PORTC->PCR[4]  = PORT_PCR_MUX(5);           //  fb_ad[11]
PORTC->PCR[5]  = PORT_PCR_MUX(5);           //  fb_ad[10]
PORTC->PCR[6]  = PORT_PCR_MUX(5);           //  fb_ad[9]
PORTC->PCR[7]  = PORT_PCR_MUX(5);           //  fb_ad[8]
PORTC->PCR[8]  = PORT_PCR_MUX(5);           //  fb_ad[7]
PORTC->PCR[9]  = PORT_PCR_MUX(5);           //  fb_ad[6]
PORTC->PCR[10] = PORT_PCR_MUX(5);           //  fb_ad[5]
PORTD->PCR[2]  = PORT_PCR_MUX(5);           //  fb_ad[4]
PORTD->PCR[3]  = PORT_PCR_MUX(5);           //  fb_ad[3]
PORTD->PCR[4]  = PORT_PCR_MUX(5);           //  fb_ad[2]
PORTD->PCR[5]  = PORT_PCR_MUX(5);           //  fb_ad[1]
PORTD->PCR[6]  = PORT_PCR_MUX(5);           //  fb_ad[0]

//data
PORTB->PCR[20] = PORT_PCR_MUX(5);           //  fb_ad[31] used as d[7]
PORTB->PCR[21] = PORT_PCR_MUX(5);           //  fb_ad[30] used as d[6]
PORTB->PCR[22] = PORT_PCR_MUX(5);           //  fb_ad[29] used as d[5]
PORTB->PCR[23] = PORT_PCR_MUX(5);           //  fb_ad[28] used as d[4]
PORTC->PCR[12] = PORT_PCR_MUX(5);           //  fb_ad[27] used as d[3]
PORTC->PCR[13] = PORT_PCR_MUX(5);           //  fb_ad[26] used as d[2]
PORTC->PCR[14] = PORT_PCR_MUX(5);           //  fb_ad[25] used as d[1]
PORTC->PCR[15] = PORT_PCR_MUX(5);           //  fb_ad[24] used as d[0]

//control signals
PORTB->PCR[19] = PORT_PCR_MUX(5);          // fb_oe_b
PORTC->PCR[11] = PORT_PCR_MUX(5);          // fb_rw_b
PORTD->PCR[1]  = PORT_PCR_MUX(5);          // fb_cs0_b
 PORTD->PCR[0]  = PORT_PCR_MUX(5);          // fb_ale
```