

Integrated Example of MSC8122 Local Bus Usage: Ethernet, TDM, DMA, and DSI

By Iantha Scheiwe

The local bus on the MSC8122 and MSC8126 DSPs is a 64-bit wide bus operating at up to 166 MHz. The local bus is used for high-bandwidth communication between the various peripherals of the device. It connects the TDM and Ethernet ports to internal M1 and M2 memories. The DMA controller uses the local bus to transfer data between M1 and M2 memories as well as between these internal memories and external SDRAM on the system bus. A host controller can access internal resources via the DSI interface to the local bus.

When you design a new system based on the MSC8122/26, it is useful to understand the interaction of the local bus masters and their interaction with the device memory structure. The programmable priority hierarchy of the local bus may require adjustment based on an application's transfer requirements. Also, you should run example scenarios to understand the bus usage when you adjust FIFO threshold levels for the TDM and Ethernet peripherals. These values must be managed to meet system latency requirements while maximizing bus usage.

The example project described in this application note includes code examples for initializing the various peripherals attached to the local bus. This project gives you a starting-point for determining the available bandwidth of a given system design. It runs on the MSC8122 DSP configured with Ethernet selected and exposed on the low part of the DSI/system data bus. Therefore, the DSI and the system bus are both 32 bits wide.

CONTENTS

1	Local Bus Structure	2
2	Local Bus Usage Project	2
2.1	Directory Structure	2
2.2	Bandwidth Usage Parameters	5
3	Peripherals	8
3.1	DMA Controller	8
3.2	TDM	11
3.3	Ethernet	13
3.4	DSI	15
3.5	Timer	17
4	MSC8122 Main Program	19
5	Bus Usage Parameters	20
6	Running the Project on CodeWarrior	21
6.1	Linker Parameters	21
6.2	Board Settings	22
6.3	Output With all Peripherals Enabled	22

1 Local Bus Structure

The local bus is the main internal conduit for TDM transactions. The DMA, DSI, and Ethernet controller send their data either to the memories on the local bus (four M1 and M2) or to external SDRAM residing on the system bus. Also, the SC140 cores use the bridge from the system bus to the local bus to access the M1 memories of other cores. **Figure 1** shows the internal connection of these masters to the local bus. If the arrow in the diagram originates from a block, the block can act as a master on the bus and request accesses according to the programmed bus arbitration scheme. All four cores access the system bus through the SQBus. It is the SQBus that requests mastership of the system bus when an SC140 core requests access to that bus.

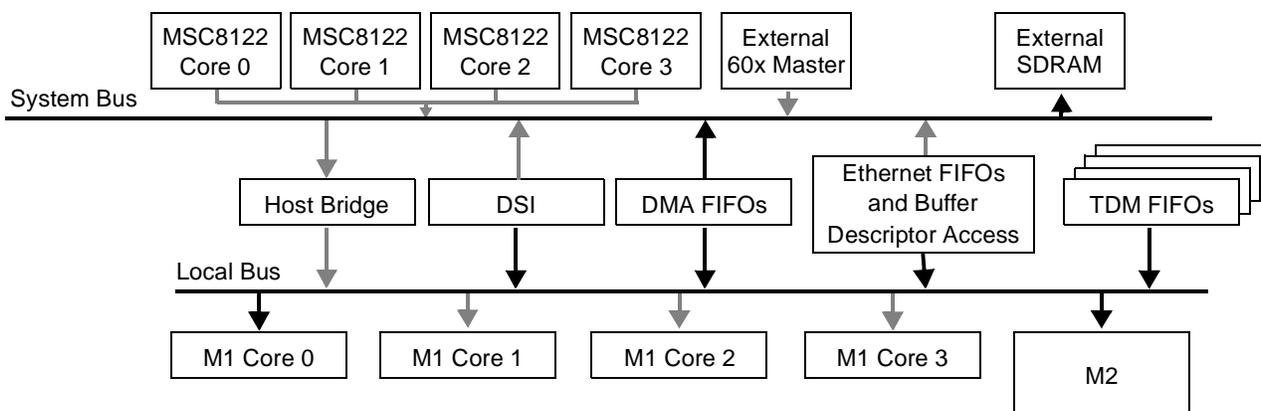


Figure 1. MSC8122 System and Local Bus Structure

As **Figure 1** shows, the host bridge, DSI, DMA controller, Ethernet FIFOs, and TDM FIFOs can each act as a master on the local bus to access the M1 memory of all four MSC8122 cores and the M2 memory. The DSI, DMA FIFOs, and Ethernet FIFOs can also access external SDRAM on the system bus. All paths shown are available on the device. However, the lines from the DSI, DMA FIFOs, Ethernet FIFOs, and TDM FIFOs to core 0 M1 and M2 memory as well as DMA to external SDRAM are drawn darker to indicate the data paths exercised in the example project described in this application note. The host bridge is an additional master on the bus, but this path tends to be low in bandwidth usage, primarily for single control accesses by one of the cores or an external master. In general, applications should minimize core accesses to the M1 memory of other cores because such accesses can significantly degrade core performance.

2 Local Bus Usage Project

The local bus usage project is a reference example for programming the MSC8122. The program has a modular design so that peripherals can be enabled/disabled or added/deleted without affecting the behavior of the rest of the program. The project includes timing mechanisms for regulating the bandwidth usage of the peripherals, so it can be used as a framework to investigate bus usage combinations and peripheral configurations. All code files in the initial release of software have a heading that indicates the version of the file. If a file is changed, its version number must be updated. The initial version number is 1.0.

2.1 Directory Structure

The project includes four main directories, as shown in **Figure 2**. The contents of each directory are described in the remainder of this section.

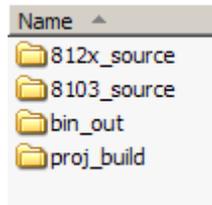


Figure 2. Project Directory Structure

2.1.1 812x_source

Figure 3 shows the contents of the 812x_source directory that holds all MSC8122 source code. The main program and linker command file are in this directory, as well as the include file for defining the project functionality and the rate timing.

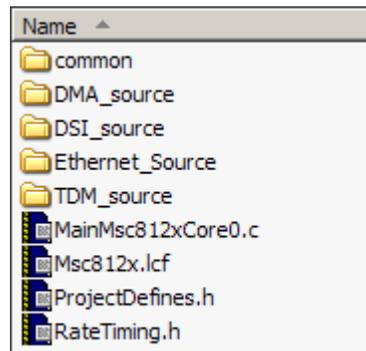


Figure 3. 812x_source Folder Contents

The folders in 812x_source contain the code to configure and manage each peripheral in the project. The common folder contains the files shown in **Figure 4**.

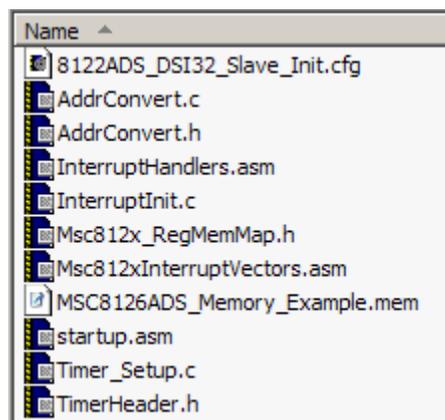


Figure 4. common Folder Contents

The files in the common folder include the interrupt handlers and timer configuration common to all the peripherals. The startup.asm file is not included in the project itself, but it is the startup.asm file to create the custom startup_be-r.e1n file used by the linker in the project (see **Section 6.1**). It does not require modification and is included in this directory only as a reference. CodeWarrior™ uses the

8122ADS_DSI32_Slave_Init.cfg file to initialize MSC8122 registers when the debugger is invoked. The project was tested with this configuration file. If another configuration file is used, check the settings to ensure that they are correct and optimal for the bandwidth test performed.

2.1.2 8103_source

Figure 5 lists the contents of 8103_source.

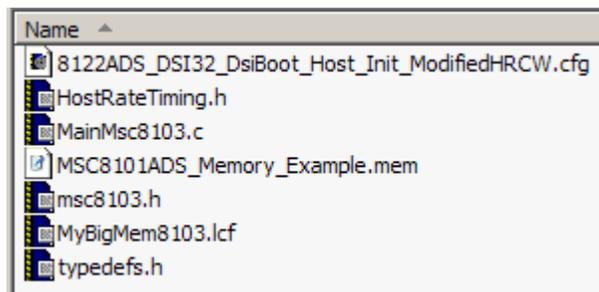


Figure 5. 8103_source Folder Contents

The MSC8103 device requires a configuration file when CodeWarrior invokes the debugger. This file is included in the 8103_source folder, as shown in Figure 5. The project was tested with this configuration file. If another configuration file is used, check the settings to ensure that they are correct and optimal for the bandwidth test performed. In addition, the MSC8103 has its own main program to initialize the MSC8122ADS board control and status registers (BCSR) and exercises the DSI on the MSC8122 using the MSC8103 system bus-to-MSC8122 DSI interface on the ADS board. Because the host exercises the DSI, the HostRateTiming.h file determines the bandwidth of MSC8103 host accesses to the DSI. The code assumes that the MSC8103 uses a 50 MHz input oscillator on the MSC8122ADS, and the switches set the MSC8103 to MODCK1. If these values change, the values in HostRateTiming.h must be updated. All other peripheral timing is defined in the RateTiming.h file in the 812x_source directory, as shown in Figure 3.

2.1.3 bin_out

Figure 6 lists the contents of bin_out.

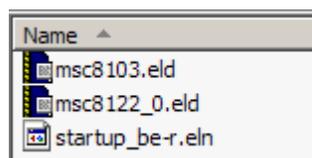


Figure 6. bin_out Folder Contents

The msc8103.eld and msc8122_0.eld files are created when the project is compiled. The linker requires the startup_be-r.eln file, which was compiled from the startup.asm source in the 812x_source/common directory. This custom start-up file leaves the upper portion of the interrupt vector table (0x200–0xFFFF) undefined so that a separate file, Msc812xInterruptVectors.asm, can be used explicitly to define the code loaded to the interrupt vector locations 0x200–0xFFFF. The startup_be-r.eln file should not be moved or modified.

2.1.4 proj_build

Figure 7 lists the contents of proj_build.

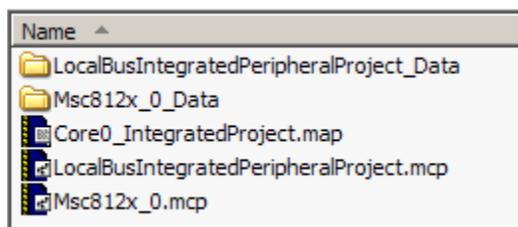


Figure 7. proj_build Folder Contents

proj_build contains the CodeWarrior project files for the MSC8103 (LocalBusIntegratedPeripheralProject.mcp) and the MSC8122 (Msc812x_0.mcp). The linker generates the MSC8122 map file and places it into this directory. CodeWarrior creates the two <>_Data folders to hold files related to the project. Opening either .mcp file launches the project.

2.2 Bandwidth Usage Parameters

The 812x_source directory includes a file named ProjectDefines.h that holds the definitions referenced in the project code. There are four groupings of defines. The first group includes a single define for the device revision when the code runs:

```
#define REV0or1K98M
```

MSC8122 1K98M mask set devices require changes to the local bus programming to complete the DMA transactions correctly. This #define should be commented out for a Rev 2 or later version of the MSC8122 because the define workaround slightly decreases the available bandwidth of the local bus. If bandwidth estimation is performed, a 2K98M mask set MSC8122 device should be used to get an accurate estimate of throughput performance.

The second group of definitions determines which peripherals controlled by the MSC8122 are active for the test:

```
#define TDMTEST
#define ENETTEST
#define LARGEDMA
#define FLYBYDMA
```

If defined, the peripheral is included in the project. If the define for a peripheral is commented out, that peripheral is not enabled and does not consume bandwidth on the local bus.

The third group of definitions relates to project parameters. By default, the Ethernet gets its clock from the PHY on the MSC8122ADS. This default is considered CONFIG_OPTION1. However, there is an option to clock the Ethernet from a timer by placing a jumper wire from a timer signal pin on the MSC8122ADS to the Ethernet clock input pin on the MSC8122ADS. This configuration is not recommended, but if a clock other than the default 25 MHz Ethernet clock is desired, this is an option. If the timer clock is used, comment out the CONFIG_OPTION1 definition. The CHECK_RATES definition enables the MSC8103 host to output the throughput values for a given code run and their associated timer values. This definition can document a particular code execution result as well as verify that the compiler properly calculated the values of the bandwidth regulations:

```
#define CONFIG_OPTION1
#define CHECK_RATES
#define PERFORMANCE_VERIFICATION
#define DEBUG_REPORT
```

The remaining two project definitions in this group affect the flow of the main program.

PERFORMANCE_VERIFICATION is defined by default and activates code checks to ensure that each peripheral does not run out of bandwidth between transfers. DEBUG_REPORT is commented out by default. If it is defined, the main program prints a statement each time a peripheral is serviced. However, the `printf` statement is a large cycle count function call and can invalidate bandwidth checking. This statement is useful to ensure that a peripheral continues to be serviced, but it should be commented out to get an accurate bandwidth estimation. TDM and Ethernet data buffers are checked regularly during code execution. However, to minimize additional local bus traffic not related to the targeted bandwidth usage, DMA buffer data is verified for accuracy only when DEBUG_REPORT is defined.

The final group of defines provides another method to disable peripherals:

```
#define DSI_STOP
#define ENET_STOP
#define DMA_STOP
```

Defining DSI_STOP is the only way to turn off host DSI transactions so that the host initializes the ADS board BCSR registers as required but does not complete any DSI transfers. ENET_STOP and DMA_STOP deactivate code in the main program and re-enable these two peripherals so that the Ethernet and DMA controllers each run one iteration and then stop. Note that there is no _STOP definition for TDM. If TDMTEST is defined in the second grouping, it runs until the program completes.

2.2.1 Rate Timing

Because this example project can be used to estimate bandwidth, it is important to understand how rate timing for the different peripherals is defined. The values for programming the MSC8122 timers to regulate peripheral timing is defined in the `RateTiming.h` file in the `812x_source` directory. The comments in this file mention the MSC8122ADS oscillator that is assumed to be feeding the MSC8122 as well as the board switch settings. Bolded values in the rate timing descriptions are the values that you can change for a specific case.

```
// MSC812x Input Crystal: 33.3 MHz
// SW4 (1-4): OFF-ON-ON-ON
#define COREFREQ(400*MEGA)
#define COREBUSRATIO3
```

If a different oscillator or MODCK setting for the MSC8122ADS is selected, update the comments and the COREFREQ and COREBUSRATIO definitions to reflect the new settings. The code was tested with values of COREFREQ = 400 and COREBUSRATIO = 3 or 4. Any valid device frequency combination should work with the code, but other combinations were not tested. If a different combination is selected, you can use the values printed by the MSC8103 host when CHECK_RATES is defined to validate the resulting calculated throughput values. The peripheral timing is based on definitions of the desired peripheral throughput and the core bus frequency. This file includes a section for defining DMA, TDM, and Ethernet throughput.

The DMA throughput is defined in MBps, DMA_MBPS. One throughput value is used for each DMA transfer enabled in the project. If more than one DMA transfer type is enabled, actual DMA throughput consumed during code execution is the DMA_MBPS value times the number of DMA transfer types (dual-access, flyby).

```
// DMA throughput in MegaBYTES per second
#define DMA_MBPS 100
// timer ticks per transfer
#define TimerLargeCompareVal (LargeDMAxferSize*(TIMERCLK/MEGA)/DMA_MBPS)
#define TimerFlyByCompareVal (FlyByDMAxferSize*(TIMERCLK/MEGA)/DMA_MBPS)
```

Based on the DMA_MBPS value, a timer comparison value is calculated to determine when the next DMA transaction is activated. The calculation includes the rate of the timer clock, which is based on the bus frequency and the size of the DMA transfer. The DMA throughput consumption can therefore be regulated. In addition to the COREFREQ and COREBUSRATIO definitions, you need only to modify the DMA_MBPS value. The rest of the DMA throughput definitions are automatically calculated.

The TDM operates differently. It is programmed to run continuously without interruption. The TDM throughput definition is for the clock used to drive the TDM:

```
//TDM Serial Speed in MegaBITS per second
#define TDM_MbPS          33
//Timer ticks per bit
#define TDMClockTimerCompareVal((BUSFREQ/MEGA)/(TIMER_TOGGLE*TDM_MbPS))
```

TDM throughput is defined in Mbps, TDM_MbPS. The granularity of the clock speed selection is regulated by the timer granularity in relation to the bus frequency. The timer operates at the same frequency as the MSC8122 system and local buses. However, the timer is programmed to toggle each time it reaches the timer comparison value, so the maximum calculated timer clock frequency is one-half the bus frequency. The maximum rate of the TDM on a 400 MHz MSC8122 device is 62.5 Mbps. The fastest timer clock that can be generated from a 133 MHz bus frequency is (133 MHz/2), 66.5 MHz, which is above the TDM limit of 62.5 Mbps. Therefore, the fastest timer clock to meet the TDM specification using the programming method in this example is 33 MHz, and the fastest rate of the TDM in this example is 33 Mbps. This is the default value in the RateTiming.h file. The value can be decreased. If a 166 MHz bus frequency is used, the fastest TDM rate using this clocking method is 41.5 MHz. Of course, an external clock source could be used to generate a higher-frequency input to the TDM. This example is a starting-point for evaluation and can be changed as necessary for a specific case.

Ethernet throughput is regulated in much the same way as the DMA throughput. However, Ethernet throughput is defined in Mbps, similar to TDM. By default, the Ethernet uses the on-board PHY clock to drive the Ethernet transactions. The initial definition of the Ethernet throughput is 70 Mbps. This default is referred to as CONFIG_OPTION1, and the Ethernet is driven with a 25 MHz clock. However, if you choose to drive the Ethernet controller with a timer clock, then a calculation based on the ENET_CLK value generates a timer clock. Due to the timer programming granularity, the timer clock is a ratio of the bus frequency. With ENET_CLK defined at 25 MHz, the timer clock generated is 22 MHz.

```
//Ethernet Serial Speed in MHz
#ifdef CONFIG_OPTION1
#define ENET_CLK          25
#else //Use MSC812x timer to generate clock for ethernet
//Timer ticks per bit
#define EnetClockTimerCompareVal(1+(BUSFREQ/MEGA)/(TIMER_TOGGLE*ENET_CLK))
//Ethernet throughput in MegaBITS per second
#define ENET_MbPS        70
//Transfers per second
#define ENET_TRANS_S     (ENET_MbPS*(MEGA/8)/ENET_BYTES_PER_TRANSFER)
//Timer ticks per transfer
#define EnetSchedTimerCompareVal(1+(TIMERCLK/ENET_TRANS_S))
```

The RateTiming.h file includes additional definitions to translate these timer values into throughput values that the MSC8103 device outputs to standard I/O when the project runs to verify that the timer value was calculated correctly. You do not change these definitions, so they are not shown or discussed here.

While this project is a useful starting-point for estimating throughput consumption and evaluating peripheral interaction, keep in mind that several factors affect peripheral throughput. The Ethernet controller operates at 10/100 Mbps. This example project uses the smallest possible data size, so it may not maximize Ethernet

throughput. If the Ethernet throughput in this project is increased to 100 Mbps, the probable result is bandwidth failures. To reach the highest bandwidth on the peripherals, you must choose the correct data size, FIFO threshold values, and bus priorities. **Section 5** describes device settings that affect bandwidth.

2.2.2 Host Rate Timing

In addition to the peripheral timing of the MSC8122, this example defines the MSC8103 usage of the MSC8122 local bus via the DSI interface, which is host-regulated and therefore defined in a separate file, `HostRateTiming.h`, in the `8103_source` directory.

```
//MSC8103 Input Crystal:50 MHz, Clock Mode:1, MODCK_H:000 MODCK[1-3]:001
//SW 6(1-8): OFF-ON-ON-ON-ON-ON-OFF-ON (MODCK1)
//Core clock speed=300 MHz
#define HOSTCOREFREQ    300000000          // (Hz)
#define HOSTBUSRATIO    3
#define HOSTBUSFREQ     (HOSTCOREFREQ/HOSTBUSRATIO)
```

If a different oscillator or MODCK setting for the MSC8122ADS is selected, update the comments and the COREFREQ and COREBUSRATIO definitions to reflect the new settings.

```
//DSI Performance target in MBps
#define DSI_MBPS        30
#define NumDSITicksAllowed ((M2_DSI_TEST_BUFFER*2)*(HOSTCOREFREQ/1000000)/DSI_MBPS)
```

DSI throughput is defined in MBps, `DSI_MBPS`, like the DMA throughput. Because it regulates only one peripheral, the MSC8103 uses its EOnCE to regulate the DSI usage rather than the timer method used on the MSC8122. Therefore, the bandwidth regulation calculation is based on the MSC8103 core frequency in addition to the DSI transfer size and throughput target number. Instead of generating a timer comparison value, the MSC8103 main program compares this `NumDSITicksAllowed` value with the value of the EOnCE counter to determine when the DSI transaction should be reactivated and whether it has met its bandwidth target.

3 Peripherals

Five resources, or peripherals, are used in this project: DMA, TDM, Ethernet, DSI, and timers. The first four peripherals consume bandwidth on the local bus. The timers regulate the bandwidth consumption. Each peripheral is programmed in a default operating mode that you can change to fit a specific test case. The code is arranged so that each peripheral has `<peripheral>Header.h` file and a `<peripheral>_Setup.c` file. Inside these files, the functionality of the peripheral is programmed. The peripheral header file includes references to global variable declarations that must be available to other files. It also includes the function prototypes for the functions in the peripheral set-up source file.

In addition, where appropriate, the peripherals have `<peripheral><data_classification>.c` files to define their global variables/buffers. Each of these files defines a `data_seg_name` referenced in the linker command file for placing the global variables in a specific memory region. A separate file is used to declare the variables in each level of memory: M1, M2, and SDRAM.

3.1 DMA Controller

The MSC8122 DMA controller has two primary transfer types: flyby and dual-access. Flyby transactions transfer entirely on the local bus and do not use the DMA FIFO. They can be transfers between the M1 memories of two SC140 cores or transfers between M1 and M2 memory. Dual-access transfers are used for DMA transfers between internal memories on the local bus (M1 and M2) and external memory on the system bus (SDRAM). The transaction goes through the DMA FIFO, which sits between the two buses. Because throughput usage may vary

for these two types of transactions due to memory latencies and bus/FIFO interactions, this example project provides both transfer types as options. Also, the DMA arbitration requests are configured, via the LCLALRH and LCLALRL register settings, as the lowest priority on the local bus. Therefore, if bandwidth starvation occurs in a specific test case, the starvation is likely to appear first as a DMA bandwidth error. The DMA code is organized in the MSC8122 project window as shown in **Figure 8**.

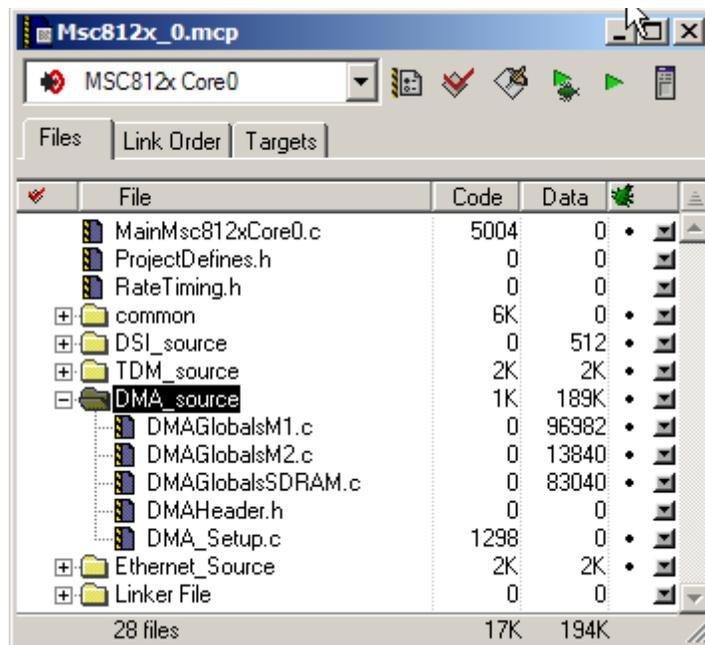


Figure 8. DMA Code Files

Three files allocate the DMA global buffers: `DMAGlobalsM1.c`, `DMAGlobalsM2.c`, and `DMAGlobalsSDRAM.c`. As the file names indicate, these files allocate the buffers in each level of memory. `DMAHeader.h` defines the DMA configuration values, DMA transfer attributes, DMA transfer size, and size of the source data pattern. These values are defined as needed for each type of DMA transfer in the project. Both DMA transfer type attributes are programmed to be cyclic (`BD_ATTR:CYC = 1`) so the buffer address and size are reset at the end of each transfer. To restart the DMA transaction, simply activate the associated channel `DCHCR:ACTV` bit and reprogram the flyby counter for the flyby DMA transactions.

This example project includes programming for two DMA transfers: `LARGEDMA` and `FlyByDMA`. `LARGEDMA` is a dual-access transfer that transfers a large block (65520 Bytes) of data. `FlyByDMA` is a smaller transfer that transfers between M1 and M2 memory on the local bus.

```
#define FlyByDMAxferSize 0x00003610 //bytes
#define LargeDMAxferSize 0x0000FFF0 //bytes
#define FlyByDCHCR      0x0085c400 //local bus, use DRACK,DMA PRAM#5,core 0 flyby counter a
#define FlyByAttr       0x00000200 //burst transactions
#define LargeDMALclDCHCR 0x00000040 //local bus, DMA PRAM #0, initiated by DMA controller
#define LargeDMASysDCHCR 0x40010040 //system bus, DMA PRAM #1, initiated by DMA controller
#define LargeDMAReadAttr 0x04000210 //DMA high priority bus request, burst transactions, read
#define LargeDMAWrtAttr  0x04000200 //DMA high priority bus request, burst transactions, write
#define DMA PattSize     0x00000050 //bytes
```

The remaining definitions in the header pertain to register settings and bus address offsets that should not be changed. The DMA configuration values defined in the `DMAHeader.h` file are referenced by the initial DMA configuration executed in the `DMA_Setup.c` file and again in the DMA channel reactivation in the `MainMsc812xCore0.c` file.

DMA_Setup.c includes a single function: DMA_Setup. The main program calls this function to initialize the source buffers to a known value and the destination buffers to 0x0. Next, the function initializes the DMA arbitration mode, channel parameter RAM for each DMA transfer type (flyby and dual-access), and DMA channel configuration registers for each DMA channel to be used. The DMA channels are not activated by this function. The main program activates each peripheral when device initialization (including the timers) completes.

3.1.1 Flyby Transfer (FLYBYDMA)

The DMA flyby transaction is programmed to transfer on the local bus as shown in **Figure 9**.

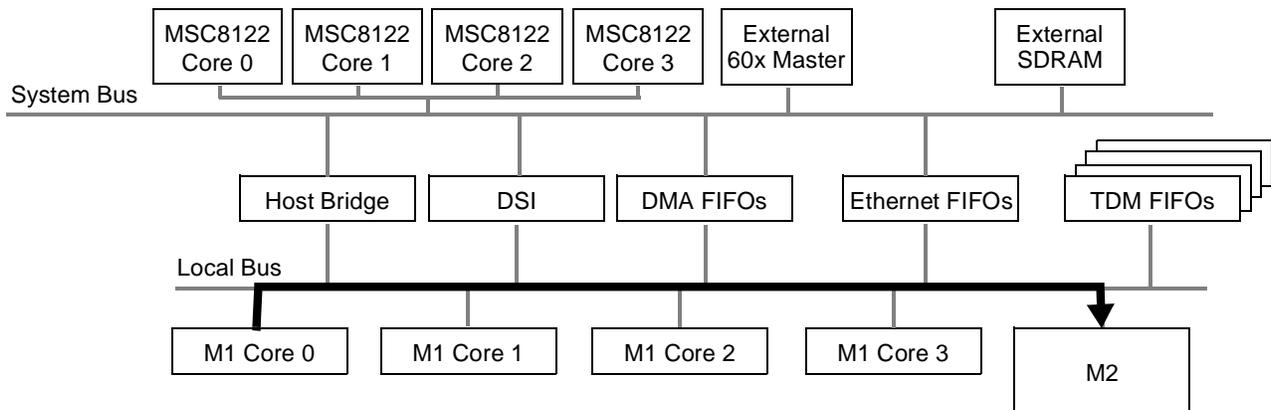


Figure 9. Flyby DMA Transaction Diagram

FlyByDMAxferSize number of bytes (13,840 bytes by default) are transferred using DMA channel 5 from M1 at a location determined by the linker placement of the buffer `gaulim1Source1` to the buffer in M2 called `gaulim2Dest`, also placed in memory by the linker. By default, the flyby transaction arbitrates for bus mastership using the DMA high-priority request (BD_ATTR:BP = 0b10).

3.1.2 Dual-Access Transfer (LARGEDMA)

The DMA dual-access transaction is programmed to transfer from M1 memory on the local bus to SDRAM on the system bus as shown in **Figure 10**.

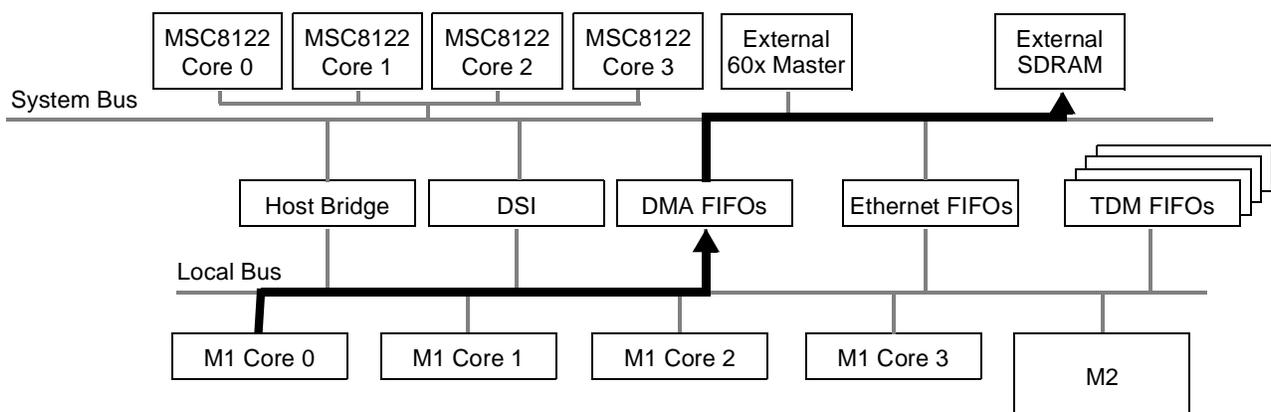


Figure 10. Dual-Access DMA Transaction Diagram

LargeDMAferSize number of bytes (65520 bytes by default) are transferred using DMA channels 0 and 1 from M1 at a location determined by the linker placement of the buffer `gauliM1Source2` to the buffer in SDRAM called `gauliSDRAMDest`, also placed in memory by the linker. By default, the dual-access attributes are programmed so that the channels arbitrate for bus mastership using the DMA low-priority request (BD_ATTR:BP = 0b00). This request is programmed by default in the main program to be the lowest priority transaction on the bus. This request is also programmed as the parked master on the local and system buses. It is a large transfer that acts as a background task and uses the local bus when no other transaction is requesting the bus. Although it has the lowest priority on the local bus, after flyby DMA, it may not experience bandwidth starvation first. The flyby DMA transaction may indicate bandwidth starvation first because it has a smaller transfer size. Its timer comparison value is smaller, so it checks bandwidth usage more frequently. Of course, transfer sizes, bus priorities, and DMA arbitration priorities can be adjusted according to the needs of the application.

3.2 TDM

The TDM transaction transfers TDM data to and from M2 memory on the local bus as shown in **Figure 11**.

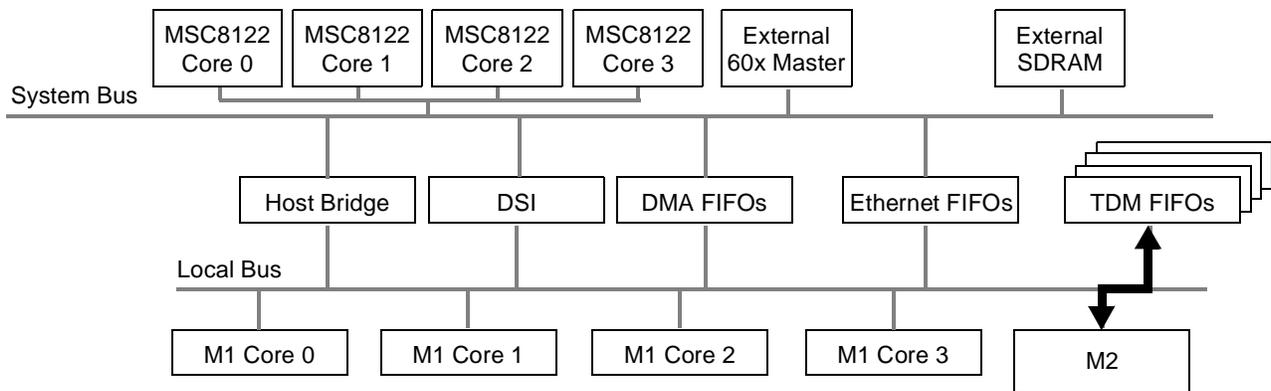


Figure 11. TDM Transaction Diagram

The TDM rate is determined by the throughput setting in the `RateTiming.h` file. In `TDMGlobalsM2.c` the TDM transmit and receive buffers are defined in a single multi-dimensional array called `TDM_M2_Buffer`. The linker places them into M2 memory. **Figure 12** shows TDM code organization in the MSC8122 project window.

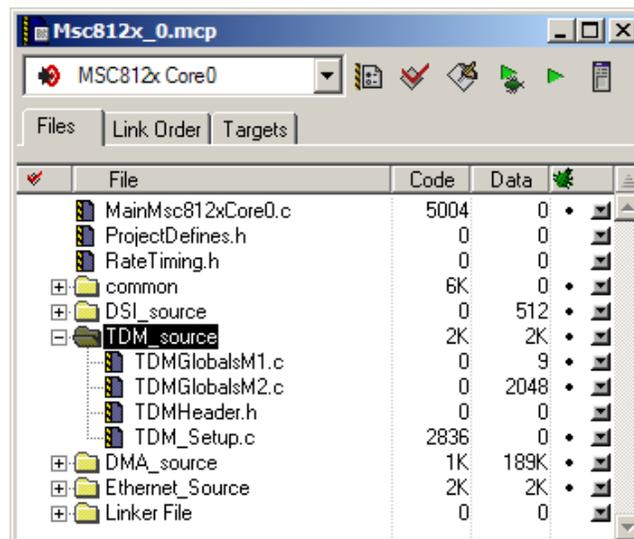


Figure 12. TDM Code Files

TDM global parameters are defined in `TDMGlobalSM1.c` and `TDMGlobalSM2.c`. `TDMHeader.h` defines the parameters for configuring the TDM module, and `TDM_Setup.c` configures the TDM for the desired operation. The parameters in `TDMHeader.h` include number of buffers, size of buffer (bytes), size of data, the number of channels, and the number of TDM modules activated for the test:

```
//TDM Transfer Parameters
#define NUM_OF_BUFFERS 32
#define BUFFER_SIZE 32
#define DATASIZE sizeof(short int)
#define NUM_OF_CHANNELS 32
#define NUM_OF_TDMS_IN_TEST 1
```

The MSC8122 device has four TDM modules, TDM[0–3]. This example project is configured to use TDM1. Enabling other TDMs in the code is straightforward. In `TDMHeader.h`, the `NUM_OF_TDMS_IN_TEST` definition can be changed to indicate more than one TDM. The active TDMs are specified in `TDMGlobalSM1.c` as follows.

```
volatile unsigned char Active_TDMS_in_Test[NUM_OF_TDMS_IN_TEST] ={
//TDM0,
TDM1//,
//TDM2,
//TDM3
};
```

The `Active_TDMS_in_Test` array defines which TDM module(s) are to be initialized and activated. This code was tested only with the MSC8122ADS board configuration, as described in **Section 6.2**, with TDM1 enabled. While other configurations are possible in the code, you must determine the appropriate board settings and test the code for any new configuration. `TDM_Setup.c` includes two functions:

- `TDM_GPIO_Setup`. Programs the GPIO registers to enable the TDMxRDAT pins for all four TDMs. However, by default, this example uses only the TDM1 data pins. Because the TDM is programmed in full-duplex loopback mode, the TDMxTDAT pins are unnecessary. The function also programs the GPIO register so that GPIO16 functions as TDM1CLK. The function includes alternative commented-out code that could be used if the TDM were programmed to use TDM0CLK. However, since the current MSC8122ADS boards have a secondary clock tied to that pin on the board, TDM module 1 and TDM1CLK is a better clock source for this project.
- `TDM_Setup`. Contains the primary TDM module initialization code. The function is passed the number of the TDM module being initialized and addresses of the TDM receive and transmit buffers. TDM is initialized to share clock and sync between transmit and receive and also to operate in full-duplex mode (TDMxGIR). TDMxRNB and TDMxTNB are initialized with the number of buffers in the TDM local buffer derived from the value of `NUM_OF_BUFFERS` defined in `TDMHeader.h`.

Next, the TDMxTIR and TDMxRIR registers are initialized. The transmit threshold interrupts are configured to be pulse. Transmit sync is an output with width of one bit and is driven on the rising edge of the clock. There is a one clock delay between the transmit sync and first bit of transmit data in the frame. The transmit data is driven on the falling edge of the clock and the sync is sampled on the falling edge of the clock. The receive threshold interrupts are level, there is no delay between the sync signal and the first bit of data in the receive frame. Receive data is sampled on the rising edge of the clock, and the sync is sampled on the falling edge.

The frame parameters are programmed for 32 channels of transmit and 32 channels of receive data, a maximum latency of 128 bits for both transmit and receive, and a channel size of 8 bits (TDMxTFP, TDMxRFP). `BUFFER_SIZE` defines the transmit and receive buffer sizes to be 32 bytes (TDMxRDBS, TDMxTDBS). The TDM buffer base addresses are initialized in `TDMxRGBA` and `TDMxTGBA` based on the address of the buffers passed to the `TDM_Setup` function. Receive and transmit thresholds are set at 8 bytes and `BUFFER_SIZE-7` bytes

(TDMxTDBFT, TDMxTDBST, TDMxRDBFT, TDMxRDBST). Interrupts are enabled for errors only (TDMxTIER, TDMxRIER). However, these interrupts are not currently enabled in the interrupt controller since the main program checks the status of the error bits manually. All transmit and receive pending events are cleared (TDMxTER, TDMxRER). TDMxACR is cleared to disable the adaptation machine. Next, the channel parameter registers are initialized for all NUM_OF_CHANNELS. This includes initializing the TDMxRCPR and TDMxTCPR registers with the data buffer address for each individual channel, activating the channel, and defining it as a transparent channel (no A-Law or μ -Law). Before the function is exited, the TDM receive and transmit buffers in TDM local memory are initialized to known values (0x0 for receive, 0x11 for transmit), and the TDM receive and transmit buffers in M2 memory are also initialized to known values (0x0 for receive, temporary calculated value for transmit). The TDM module is then ready to be enabled.

3.3 Ethernet

The Ethernet transaction is programmed to transfer Ethernet data to and from M2 memory on the local bus as shown in **Figure 13**.

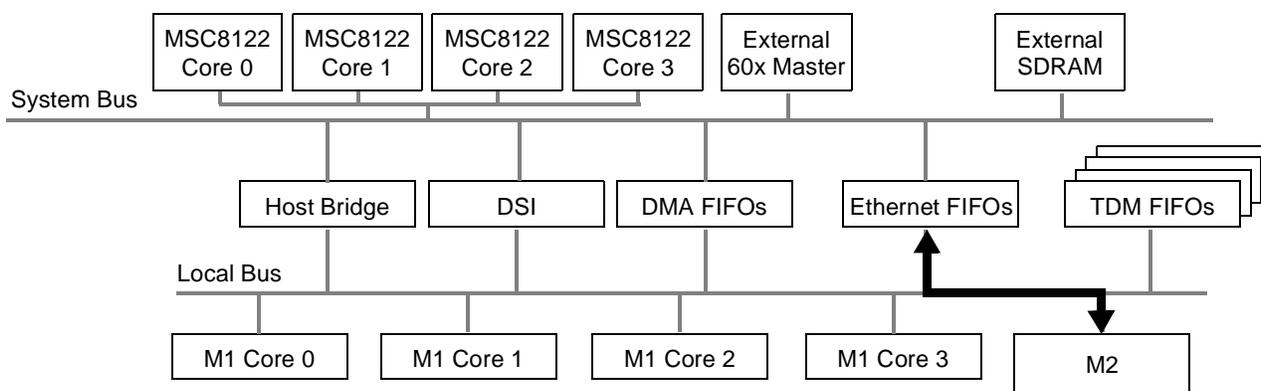


Figure 13. Ethernet Transaction Diagram

The Ethernet code is organized in the MSC8122 project window as shown in **Figure 14**.

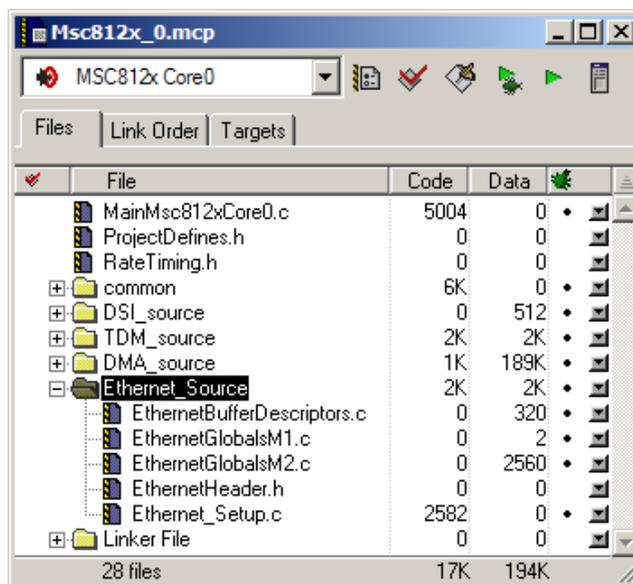


Figure 14. Ethernet Code Files

Ethernet global variables are defined in three files: `EthernetBufferDescriptors.c`, `EthernetGlobalsM1.c`, and `EthernetGlobalsM2.c`. The names of the second two files indicate where the linker should place the global variables. The buffer descriptors can be placed into M1 or M2 memory. This project places them into M1 memory in the linker command file. `EthernetHeader.h` and `Ethernet_Setup.c` serve functions similar to their counterparts for the other peripherals. `EthernetHeader.h` includes definitions to configure the Ethernet controller. It also references the Ethernet global variables, function prototypes, and definitions of Ethernet register bits used in Ethernet controller initialization. There is a definition for the number of Ethernet frames to be transmitted/received, the transmit frame size, receive frame size (based on transmit frame size), transmit and receive buffer sizes, and the number of transmit and receive buffer descriptors. In this example, the number of buffer descriptors equals the number of frames.

```
//Ethernet Transfer Parameters
#define NUM_OF_FRAMES 20
#define TX_FRAME_SIZE 60
#define RX_FRAME_SIZE (TX_FRAME_SIZE+4)

//Buffer sizes in bytes (must be divisible by 16)
#define TX_BUFFER_SIZE 64
#define RX_BUFFER_SIZE 64

//Number of Receive and Transmit Buffers and Buffer Descriptors
#define NUM_RXBDS NUM_OF_FRAMES
#define NUM_TXBDS NUM_OF_FRAMES
```

`Ethernet_Setup.c` contains a single function called `Ethernet_Setup`. The Ethernet configuration follows the steps defined in the *MSC8122 Reference Manual* to configure the Ethernet controller after reset. The Ethernet controller is enabled by setting the `MIIGSK_ENR[EN]` bit. When the controller is ready, it is placed into soft reset and then released. The MAC configuration is set to full-duplex mode with a frame specification that includes a 7 byte preamble and appended CRC. The MAC station address is initialized to a known value. For this example project, the Ethernet is transmitting in a MAC-to-MAC configuration, so no PHY configuration is required. The controller is initialized for MII loopback mode. All pending events are cleared and interrupts are disabled. The pattern matching and hash function registers are initialized, but the Ethernet controller is configured in promiscuous mode, so pattern matching is ignored. The DMA maintenance register handles the priority for Ethernet transactions as they near overrun/underrun situations. In the default configuration, all three Ethernet priorities are defined together in the arbitration scheme, so the setting in this register has little effect. However, for a specific application, it may be appropriate to change the arbitration hierarchy scheme in `LCL_ALRH` and `LCL_ALRL` and the arbitration priority values in `DMAMR`.

At this point, the buffer descriptors are built. The transmit buffer descriptors are built first. Although the Ethernet controller is not yet fully enabled, the code follows the recommendation that the first buffer descriptor in the transmit buffer descriptor ring be the last to be marked as ready. Therefore, all transmit buffer descriptors in the ring are initialized except transmit buffer descriptor 0, which is initialized last. The frame size is initialized to be 60 bytes, which is the smallest possible Ethernet frame size. The transmit data buffer is also initialized. Next, the receive buffer descriptors are initialized. Both the transmit and receive buffer descriptor rings are the same size. Finally, the FIFOs are initialized and the Ethernet controller is ready to transmit and receive data when the main program fully enables it.

3.4 DSI

The DSI transaction is programmed to transfer a block of data from the MSC8103 internal memory to the MSC8103 DMA FIFO and then from the DMA FIFO to the MSC8103 system bus interface and the MSC8122 DSI. From the DSI it transfers data to M2 memory on the MSC8122 local bus. When that transfer completes, a second transaction reads the data back along the same path, as shown in **Figure 15**.

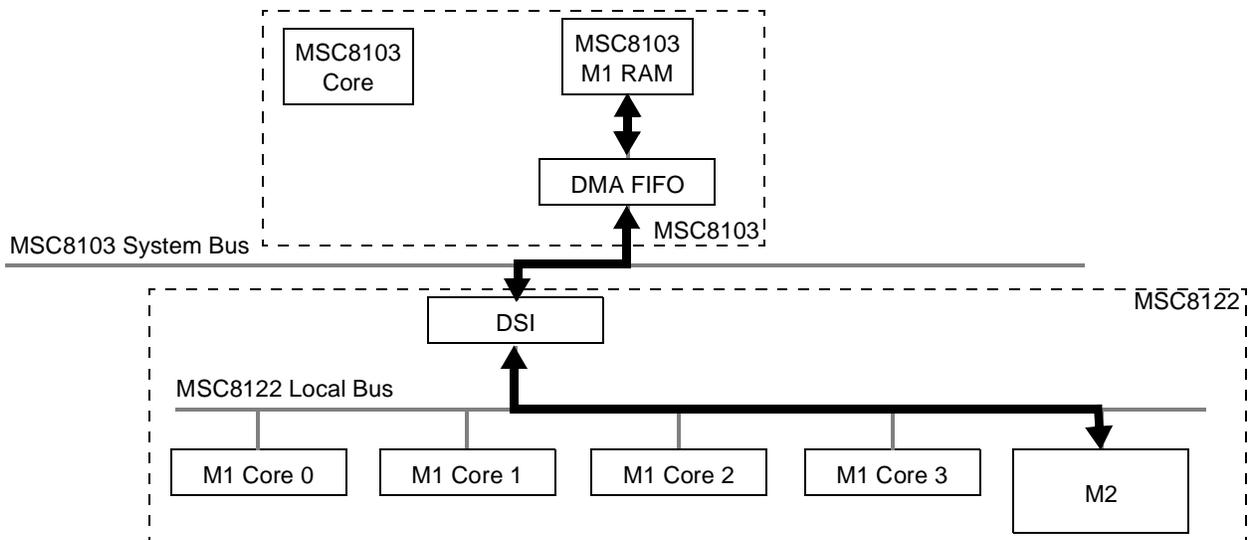


Figure 15. DSI Transaction Diagram

The DSI code is organized in the MSC8122 project window as shown in **Figure 16**.

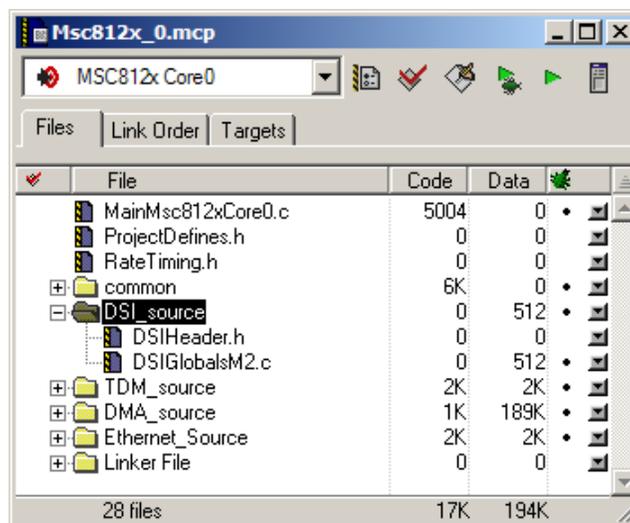


Figure 16. DSI Code Files on the MSC8122

There are only two files: `DSIHeader.h` and `DSIGlobalsM2.c`. `DSIHeader.h` defines the size of the data buffer that is written and read via the DSI (`M2_DSI_BUFFER_SIZE`).

```
#define M2_DSI_BUFFER_SIZE 128
#define M2_DSI_DATA_OFFSET 0x0000
#define HOST_SIDE_M2_BUFFER_ADDRESS (0x24000000 + M2_DSI_DATA_OFFSET)
```

The DSI data buffer size is set to 128 bytes. This definition can be adjusted as needed. The associated `DSIGlobals.c` file declares the buffer in M2 memory, and the array references this same `M2_DSI_BUFFER_SIZE` definition, as does the MSC8103 when initializing its transfer size.

The CodeWarrior configuration files that initialize the MSC8122ADS when loading the debugger define the DSI address and chip ID combination so that the DSI base address for the MSC8103 is `0x24000000`. This address is the first address location in M2 memory that is accessible to the MSC8103 host via DSI. In the MSC8122 linker file, the DSI buffer in M2 is reserved at the start of M2 memory. If the location of this buffer is changed, the offset address must be updated in `DSIHeader.h` so that the MSC8103 host knows where it can transfer its data block. To make this update, change the `M2_DSI_DATA_OFFSET` definition. Technically, the DSI buffer can be placed anywhere in the M1 or M2 memories of the MSC8122, and this offset is simply changed to reflect the new offset. DSI transfers are not restricted to M2 accesses.

The MSC8103 host is responsible for all DSI data transfers. The MSC8122 only tells the linker to reserve the memory block for use by the host. All MSC8103 programming is contained in `MainMsc8103.c`. Besides configuring the Board Control and Status Registers (BCSRx) of the MSC8122ADS to enable the Ethernet PHY clock, the only other task of the MSC8103 is to initiate DMA transactions to the MSC8122 DSI. When the entire example has run, the MSC8103 can output the data rates for the run just completed. The code is organized in the MSC8103 project window as shown in **Figure 17**.

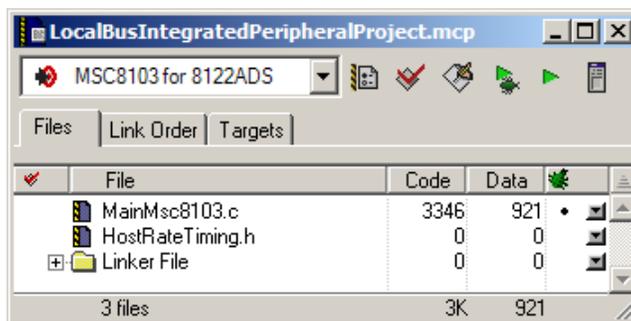


Figure 17. Code Files on the MSC8103 Host Device

As noted in **Section 2.2.2, Host Rate Timing**, on page 8, the MSC8103 uses EOnCE to regulate its bandwidth usage. Immediately after initializing the BCSRs and its system and local bus parked masters, the MSC8103 enables EOnCE to count core clock cycles. It then initializes a source buffer to a known pattern to be used as a reference vector for verifying correct data transmission, and it clears the receive buffer.

The MSC8122ADS has an asynchronous DSI connection between the MSC8103 and MSC8122, so bursting is not possible. In addition, with the system bus enabled to a 32-bit width and Ethernet enabled and configured to be exposed on the MSC8122 system bus/DSI pins, the DSI is 32-bits wide. If a transaction is placed on the 60x-compatible system and local buses that is wider than the slave port size, the transaction is split into smaller transactions to fit the port size. Each of the smaller transactions does not require arbitration between the transactions. The DMA coding of the MSC8103 in this example takes advantage of the throughput increase that can result by taking advantage of this 60x bus feature. Instead of programming the DMA to transfer bursts (which is not possible due to the asynchronous interface) or 32-bit transactions (which would result in slightly lower bandwidth) on its system bus, the `BD_ATTR` parameter for the transactions is programmed for 64-bit transactions. The MSC8103 local bus transactions between MSC8103 M1 memory and the DMA FIFO use bursting since they are on the MSC8103 internal local bus.

The MSC8103 programs the DMA controller for two dual-access transactions. One dual-access (write) transaction reads from M1 memory, writes to the DMA FIFO, then reads from the DMA FIFO and writes to the DSI. The second (read) transaction reads from the DSI and writes to the MSC8103 DMA FIFO, then reads from DMA FIFO and writes to M1 memory. DMA channels 0 and 1 and channel parameter RAM 0 and 1 are configured for the write transaction. DMA channels 2 and 3 and channel parameter RAM 2 and 3 are configured for the read transaction. When the DMA controller is initialized, the MSC8103 enters a while loop that determines the number of times the DSI transaction executes. If `DEBUG_REPORT` is defined, the code runs infinitely.

Inside the `while` loop, the DMA write transaction is activated and the EOnCE counter is reset. The MSC8103 polls the DMA status register until the write transfer completes. Then it activates the DMA read transaction and polls the DMA status register until the transaction completes. Next, the MSC8103 checks the value of the EOnCE counter to record the elapsed time of the DMA transfers to and from the DSI. The elapsed time is compared with the number of core clocks that are allowed for the DSI transaction to complete within its bandwidth allocation. If `DEBUG_REPORT` is enabled, the MSC8103 checks the read data buffer to verify that it matches the reference vector and then changes the reference vector to a new pattern. If the bandwidth target is not met, the MSC8103 hits a debug statement immediately below a comment indicating that the DSI did not meet its throughput commitment. If `DEBUG_REPORT` is not defined, the MSC8103 waits until the elapsed time allowed for the DSI transaction is exhausted and then restarts the DMA transactions to DSI.

3.5 Timer

The timer code is organized in the MSC8122 project window in the common directory as shown in **Figure 18**.

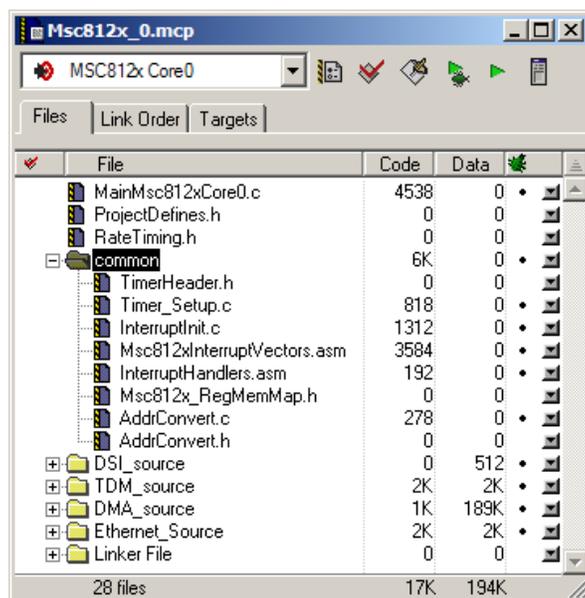


Figure 18. common Files in the MSC8122 Project Window

Two files define the timer behavior: `TimerHeader.h` and `Timer_Setup.c`. `TimerHeader.h` is organized into groupings for each peripheral. Each group includes the timer configuration definition for that peripheral, the function prototypes for timer functions related to that peripheral, and required references for timer-related global variables for the peripheral. In addition, `TimerHeader.h` contains register bit definitions used in `Timer_Setup.c`. In general, the timer definitions should not require modification.

```
//DMA TIMER PARAMETERS
#define TimerFlyByConfigVal 0x00000030
```

```

#define TimerLargeConfigVal 0x00000030
//DMA Timer Function Prototypes
void DMA_Timer_InterruptSetup(void);
void DMA_Timer_Setup(void);
//DMA Timer Global Declarations in other files
extern volatile unsigned char vucDMA_TimerA0_Int_Flag;
extern volatile unsigned char vucDMA_TimerA1_Int_Flag;

//TDM TIMER PARAMETERS
//TDM CLOCK TIMER (B4 internal, Timer3 external)
//Connect J5.B15 to J5.B19 on the MSC812xADS
#define TDMClockTimerConfigVal 0x00000031
//TDM Timer Function Prototypes
void TDM_Clock_Timer_Setup(void);

//ETHERNET TIMER PARAMETERS
#define EnetClockTimerConfigVal 0x00000031
#define EnetSchedTimerConfigVal 0x00000030
//Ethernet Timer Function Prototypes
void Enet_Timer_InterruptSetup(void);
void Enet_Timer_Setup(void);
//Ethernet Timer Global Declarations in other files
extern volatile unsigned char vucEnet_TimerA2_Int_Flag;

```

Timer_Setup.c contains one function for each of the three peripherals it regulates. These are DMA_Timer_Setup, Ethernet_Timer_Setup, and TDM_Clock_Timer_Setup. The Ethernet and DMA set-up functions configure the timer to one-shot mode so that it stops after reaching its comparison value and defines the timer clock source to be the bus clock, which is set at 133 MHz by the board setting defined in this application note, but modifiable as necessary. Each MSC8122 DMA transaction gets its own timer, so there is one configured for the flyby DMA transaction and one for the dual-access (large data buffer size) transaction. The timer comparison registers are initialized with the value calculated in the RateTiming.h file based on the target throughput definitions for each peripheral. These three timers (flyby DMA, large DMA, and Ethernet) each generate an interrupt when the timer reaches its comparison value.

When the interrupt is triggered, the MSC8122 core jumps to the allocated interrupt vector defined in Msc812xInterruptVectors.asm, which directs it to a handler routine in InterruptHandlers.asm. The handler routine clears the timer event and interrupt status registers and sets a flag to indicate the timer event occurred.

The TDM timer does not generate an interrupt. It is a cyclic, free-running timer whose timing is also derived from the bus clock, but whose output is directed to Timer B4. Timer B4 is available on the MSC8122ADS expansion connector J5 as signal TIMER3. This signal (J5:B19) must be jumpered to the TDM1CLK signal on the MSC8122ADS (expansion connector J5:B15) to clock the TDM transactions. The timer rate is also derived from the TDM throughput value defined in RateTiming.h.

The Ethernet_Timer_Setup function also includes a timer clock initialization that executes only if CONFIG_OPTION1 is not defined. In this case, the Timer B0 output frequency is derived from the Ethernet clock definition in RateTiming.h. This signal corresponds to the TIMER2 signal on the MSC8122ADS, which is accessible on expansion connector J5:C22 and can be tied to the Ethernet clock pin on JP4:5. This option is used only if the 25 MHz clock driven from the PHY is not acceptable and a different clock source/rate is necessary.

4 MSC8122 Main Program

When the peripherals and rate timing are set, the main program manages all bandwidth checking and peripheral activation. The main program is divided into four sections.

1. Initialize device registers, including enabling the instruction cache, programming the local bus parameters for bus timing and arbitration, and configuring the EOnCE to count core clock cycles (the EOnCE counter is used to manage the length of time the main program runs).
2. Initialize interrupts and call the set-up functions for each peripheral as described in **Section 3**.
3. Enable each peripheral and its associated timer.
4. Handle the run-time coordination and bandwidth checking for each peripheral. This (largest) portion of the program is contained within a while loop whose length of execution is dependent on the EOnCE counter. When this counter reaches 0, the loop exits. However, if `DEBUG_REPORT` is enabled the loop runs infinitely.

Excluding DSI, which is managed by the MSC8103, each of the four peripheral tests in the example project has a section of management code in the main program. TDM, which is the first peripheral, is checked to see if its receive event register (TDMxRER) indicates an event. If so, the MSC8122 enters the TDM management routine. First it verifies that the event(s) is not an error event. If it is an error event, the MSC8122 increments its `TDM_total_fail` counter. If bandwidth checking is enabled (`PERFORMANCE_VERIFICATION` is defined), the beginning and end of the TDM receive buffer data are checked for correctness. If the data is not correct, the MSC8122 reaches a `debug` statement immediately below a comment indicating a TDM data failure. TDM differs from the Ethernet and DMA controllers because once it is enabled it runs freely without core interaction. Also, the local bus is configured so that TDM requests are the highest priority on the bus. Therefore, they should not experience bandwidth starvation. However, it is still important to check the data to ensure that it is transferred properly and to ensure confidence in the results of the test. After the data is checked, the receive buffer is modified so that it does not hold the same value as the transmit buffer. This check ensures that the receive data arrives correctly each time and not just the first time. If `DEBUG_REPORT` is defined, a message is printed indicating either that the TDM is working as expected (**TDM!**), or if the TDM had receive event errors, the program prints **TDM is not working correctly**.

Each peripheral has its own counter, `num<peripheral test>iterations`, to indicate the number of times the MSC8122 SC140 core has serviced it. This counter can be checked after code execution to ensure that all peripherals were serviced and had the opportunity to indicate potential bandwidth starvation or data errors.

Both DMA transfer types have their own management code in the main program, but the code is organized similarly for each. As noted in **Section 2.2.1, Rate Timing**, on page 6 and **Section 3.5, Timer**, on page 17, each DMA and Ethernet transfer has its own timer programmed with a comparison value to regulate its bandwidth usage. When the timer reaches the comparison value, it generates an interrupt. In the interrupt service routine the SC140 core sets a flag to indicate a timer interrupt. The main program checks this flag. If the flag is set, the main program clears the flag and checks to see if the associated transfer completed. If not, an error flag is set (`FlyByDMAErrorFlag`, `LargeDMAErrorFlag`, `EnetRxErrFlag`, or `EnetTxErrFlag`). When `PERFORMANCE_VERIFICATION` is defined, these flags are checked and the MSC8122 hits a `debug` statement if they are set.

The timer associated with the peripheral is immediately re-enabled to start counting for the next bandwidth check. For DMA transfers, the error flag is checked and if `DEBUG_REPORT` is defined, the DMA data buffers are checked for correctness. A print statement indicates whether the DMA transfer is executing as expected or encountered errors. If `DMA_STOP` is not defined, the DMA transfer is restarted. A fail counter is updated to indicate whether a failure occurred on this iteration. This fail counter is useful if `PERFORMANCE_VERIFICATION` is not defined to

put the MSC8122 into debug mode when failures occur. Finally, the iteration counter is incremented to indicate that the peripheral was serviced and restarted. The Ethernet timer is managed in the same way as the DMA timers. The timer flag is cleared immediately when the Ethernet management portion of the main program is entered, and the Ethernet bandwidth regulation timer is re-enabled. Next, the Ethernet transmit and receive are checked to ensure that they completed their transfer. If not, the Ethernet Rx or Tx error flag is set. As for other peripherals, if `PERFORMANCE_VERIFICATION` is defined, the SC140 core hits a debug statement when a bandwidth error is detected. Also, the Ethernet buffers are checked for errors. Unlike the DMA management, which checks the data buffers only if `DEBUG_REPORT` is defined, Ethernet buffers are checked any time performance is verified. Next, the Ethernet receive and transmit buffer descriptors are reinitialized and the receive buffer data is modified to ensure that it is getting a new transfer each time. If `DEBUG_REPORT` is defined, the program prints a statement indicating whether Ethernet is working properly. The Ethernet error flags are cleared and its iteration counter is incremented.

5 Bus Usage Parameters

Numerous factors affect bus usage. If an application is maximizing bus usage, check the following list for tips on getting the most throughput on the local bus:

- Verify that the maximum bus frequency is being used. With 400 MHz MSC8122 devices, the maximum bus frequency is achieved with a 1:3 bus:core ratio and results in a 133 MHz local bus. The 500 MHz MSC8122 devices can run the bus at 166 MHz. Verify that the input oscillator and clock mode settings for the MSC8122 are configured for the correct bus speed.
- Check the data pipelining setting for the bus. Setting `BCR[PLDP]` decreases the bus throughput. This bit is cleared by default in this project.
- Extended transfer mode (`BCR[LETM]`, `BCR[ETM]`) improves bus performance. It must be disabled on the local bus on 1K98M or 2K98M mask sets of the MSC8122 (see the definition to handle this in **Section 2.2.1, Rate Timing**, on page 6). However, for Rev 2 or later devices, extended transfer mode should be enabled to attain higher throughput.
- Both TDM and Ethernet have their own DMA engines to transfer data between the interface local memory or FIFOs and MSC8122 internal M1 and M2 memories. There are registers to manage the threshold for determining when to transfer the data. These thresholds can be set to maximize bus usage, and there can be multiple back-to-back transactions on the bus to improve bus usage significantly. However, especially for TDM, which generally has low latency requirements, ensure that the data is transferred in time to meet the requirements of the application. For the Ethernet controller, the thresholds are set in `FTXTHR` and `FRXALAR`. The TDM thresholds are programmed in `TDMxRFP[RCDBL]` and `TDMxTFP[TCDBL]`.
- Adjust the bus arbitration priorities in the `LCLALRH` and `LCLALRL` registers. In general, the highest-priority master should require the lowest latency. Ideally, the high-priority transactions are also relatively short transactions (that is, not a 65 KB DMA transaction that stalls all other transactions for a significant period of time).
- Each Ethernet FIFO has an alarm setting to raise the Ethernet arbitration priority on the bus if it is not getting data on time. An application can take advantage of this feature by programming the Ethernet controller as a low-priority bus requestor with an alarm mode to raise its priority and give it access to the bus when it is starved of bus usage (`DMAMR[APR]`).
- The DMA controller has three different priority levels to request the bus. These levels can be managed on a channel-by-channel basis. These levels differ from the DMA priority levels for mastership of the

DMA engine itself. When DMA round-robin arbitration is not in use, a high-priority channel should be programmed for both high channel priority (DCHCR[PRIO]) and high bus request priority (BD_ATTR[BP]).

- To save arbitration cycles, ensure that one of the masters is programmed as a parked master. In the project example, the parked master is the one with the lowest arbitration priority and highest bandwidth usage (dual-access DMA). In general, the master with the highest bandwidth usage should be configured as the parked master to get the most benefit out of savings in arbitration cycles.
- If system bus accesses or DSI accesses are running out of bandwidth and Ethernet is enabled, check the system design to see whether Ethernet can be exposed on the GPIO pins instead of the system bus/DSI pins. Either the DSI or system bus can be configured to 64 bits instead of 32 bits for twice the throughput capacity. This decision is dependent on the configuration and usage of the TDM.
- Ethernet frames have significant flexibility in how they are transmitted. To increase Ethernet throughput, consider decreasing the preamble size, increasing the data load in each frame, decreasing the inter-packet gap size, and determining whether CRC or padding is required. Adjusting these parameters, an application can send more data and less overhead through the Ethernet interface.
- If DSI throughput reaches a ceiling, consider switching to a synchronous interface so that burst transfers can be used. Note that synchronous mode is not supported on the MSC8122ADS, so this throughput improvement cannot be measured on the ADS.
- If an interface to SDRAM creates a bottleneck, employ the DMA interleaving technique and SDRAM configuration settings described in AN2704, *Using DMA-SDRAM to Optimize Bandwidth on the StarCore MSC8102 DSP Device*, and AN3011, *DMA Interleaving Technique for High Bandwidth Between the MSC8122 and SDRAM*.

6 Running the Project on CodeWarrior

When the definitions and rate timing that dictate the behavior of the example code are set to reflect the particular test being verified, the code is ready to run. The device configuration set by the `.cfg` files in the MSC8103 and MSC8122 source file directories and selected in the project settings must be the *Initialization File* when each project is loaded into the debugger. This section discusses the linker and board settings necessary to ensure that the code runs as expected.

6.1 Linker Parameters

In the *MSC812x Core 0 Target Settings* → *Enterprise Linker* panel, the linker is directed to use a custom start-up file derived from the default start-up file used by CodeWarrior. This custom file has been modified to leave the interrupt vector memory locations from 0x200–0xFFFF empty so that the MSC8122 project can include a separate `Msc812xInterruptVectors.asm` file to define handlers explicitly for each interrupt vector, referred to as `.myintvec` in the MSC8122 linker command file.

The linker command file is also programmed to include the interrupt handler routines from `InterruptHandlers.asm` in M1 memory along with the `<peripheral>GlobalSM1.c` variable declarations and the Ethernet buffer descriptors. All other code is placed into M2 memory along with the DSI buffer and the Ethernet and TDM transmit and receive buffers. M2 memory also holds the destination data buffer for the flyby DMA transfer. SDRAM contains the destination data buffer for the dual-access DMA transfer.

The MSC8103 uses the default linker command file with everything, code and data, defined in internal M1 memory.

6.2 Board Settings

The Ethernet code depends on a clock source driven from the PHY on the MSC8122ADS. Therefore, the MSC8103 host programs the MSC8122ADS board registers (BCSR 7, 8, and 9) to enable the PHY to generate the 25 MHz clock for Ethernet. This code is the first code the MSC8103 executes. It is essential for the operation of the Ethernet and should not be removed. The code also disables a clock driven from the TSI on the MSC8122ADS to the TDM.

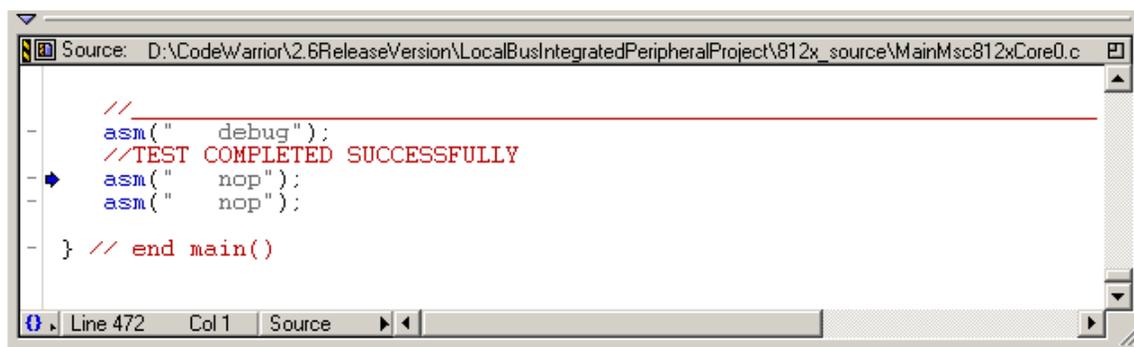
The MSC8122ADS switches and jumpers should be configured as follows:

- SW4> OFF-ON-ON-ON
- SW5> OFF-ON-ON-ON
- SW6> OFF-ON-ON-ON-ON-ON-OFF-ON
- SW7> ON-ON-ON-ON
- JP1> All open.
- JP2> OSCK-IN connected. Others open.
- JP3> All open.
- JP4> All open.
- JP5> CLKOUT-DRV connected. EXCLK open.
- JP6> All open.
- JP8> Open.
- For TDM1 clock> Connect J5:B15 to J5:B19.
- MSC8103 Oscillator> 50 MHz
- MSC8122 Oscillator> 33.3 MHz

This project is intended as a starting-point for peripheral and throughput evaluation. You can change these settings for other clock frequency combinations or peripheral combinations. Refer to the *MSC8122ADS Reference Manual* to determine necessary ADS adjustments.

6.3 Output With all Peripherals Enabled

If `DEBUG_REPORT` is not defined and the code finishes executing, the two debugger windows (`msc8103.e1d` and `msc8122_0.e1d`) stop to indicate successful completion, as shown in **Figure 19**.



```

Source: D:\CodeWarrior\2.6ReleaseVersion\LocalBusIntegratedPeripheralProject\812x_source\MainMsc812xCore0.c
//
asm("  debug");
//TEST COMPLETED SUCCESSFULLY
asm("  nop");
asm("  nop");
} // end main()
Line 472 Col 1 Source

```

Figure 19. Indication of Successful Code Completion

How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations not listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:

Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:

Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale, the Freescale logo, and CodeWarrior are trademarks of Freescale Semiconductor, Inc. StarCore is a licensed trademark of StarCore LLC. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2005.