Triscend

Using the E5 Embedded DMA Controller

July 2001

AN-22

Abstract

This application note describes how to use the DMA feature of the E5 by working through example designs.

Contents

Scope

This DMA application note will demonstrate how to use the DMA channels on the E5 device contained on the Triscend Development Board. Although the information contained in this application note is valid for all E5 CSoC devices, this document will focus on the E520 because it featured on the E5 development board. The basics of the DMA feature can be found in the E5 Data Sheet. This application note will cover the following more advanced DMA functions:

- Software Request Mode
- Continuous Initialization
- DMA Reads
- DMA Writes

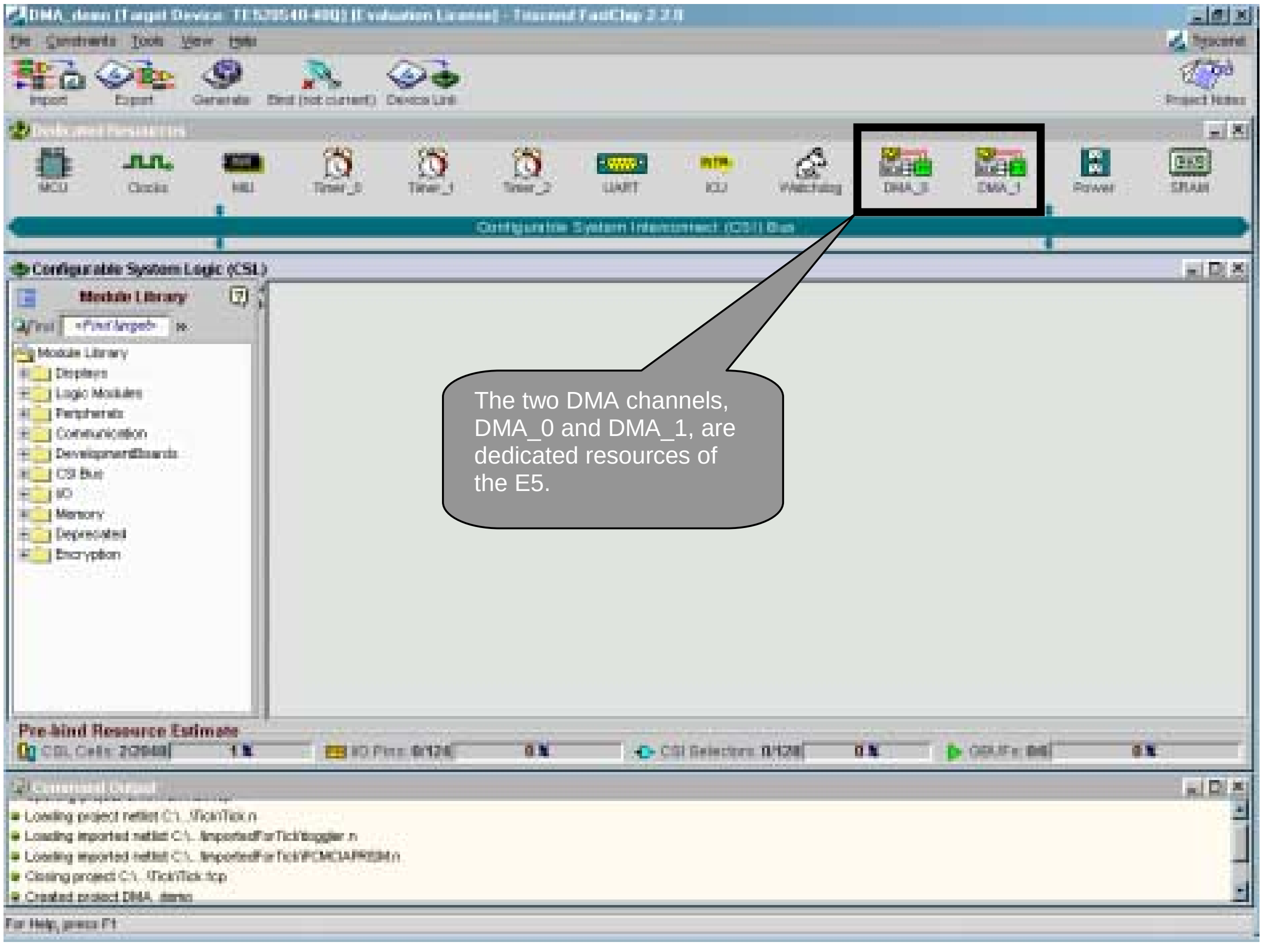
It is assumed that the user understands the basics of FastChip, FastChip Device Link Utility, and Keil Software development tools for software development and debug.

Overview

The E5 features an embedded two-channel DMA controller, which allows the CSL to access memory. Each channel is autonomous from the 8032 micro-controller, freeing the processor from mundane, performance-stealing, data transfer tasks. Each channel is designed to transfer a byte of data on each clock cycle. The following data transfer modes are available:

- DMA Read → I/O (CSL) to Memory
- DMA Write → Memory to I/O (CSL)
- Memory-to-Memory

A device associates itself with a particular channel via the DMA control register (DMA Selector), which contains a request and acknowledge signal pair. The two DMA channels can be paired to perform memory-to-memory transfers.



```
Loading project netlist C:\...\TickTick.n
Loading imported netlist C:\...\ImportedForTick\toggler.n
Loading imported netlist C:\...\ImportedForTick\PCMCIAPRISM.n
Closing project C:\...\TickTick.tcp
Created project DMA_demo
```

For Help, press F1

The two DMA channels, DMA_0 and DMA_1, are dedicated resources of the E5.

Figure 1: DMA channels shown in FastChip

Functional Description

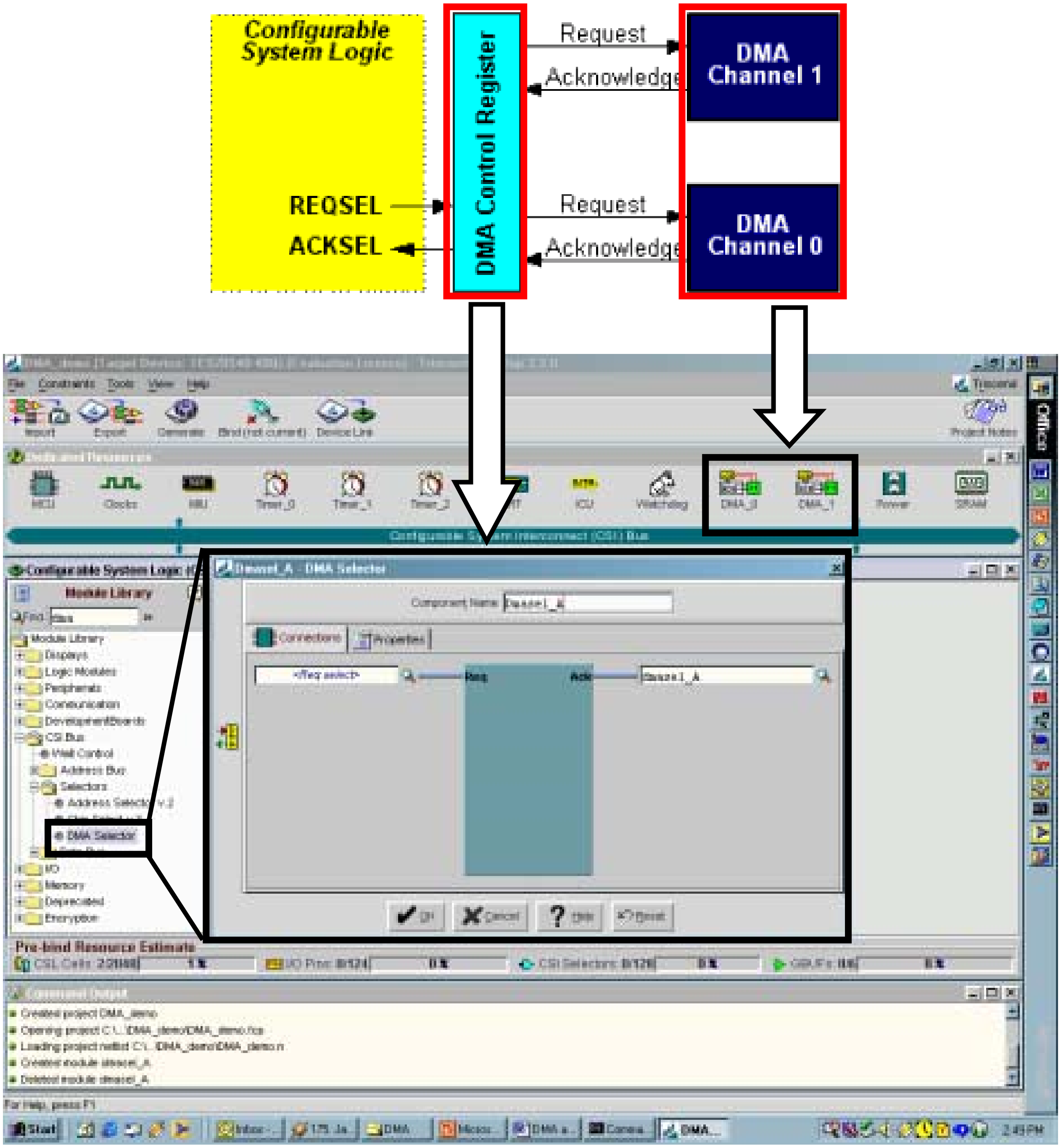
The E5's DMA feature is comprised of two independent channels. The operation for each DMA channel is defined by a set of parameters. From a designer's point of view, the most important set-up parameters are:

- Memory starting address
- Starting transfer count
- Direction of transfer

Although each channel has its own set of registers, a designer can configure each DMA channel without delving into the details of each configuration register. By using FastChip, a designer can easily configure each DMA channel via the GUI.

Configuring the DMA Channels

The easiest and most efficient way to configure the DMA channels is via FastChip's GUI. Figure 2 illustrates the relationship between FastChip and the DMA channels.



```
Created project DMA_demo
Opening project C:\...\DMA_demo\DMA_demo.fcp
Loading project netlist C:\...\DMA_demo\DMA_demo.n
Created module dmasel_A
Deleted module dmasel_A
```

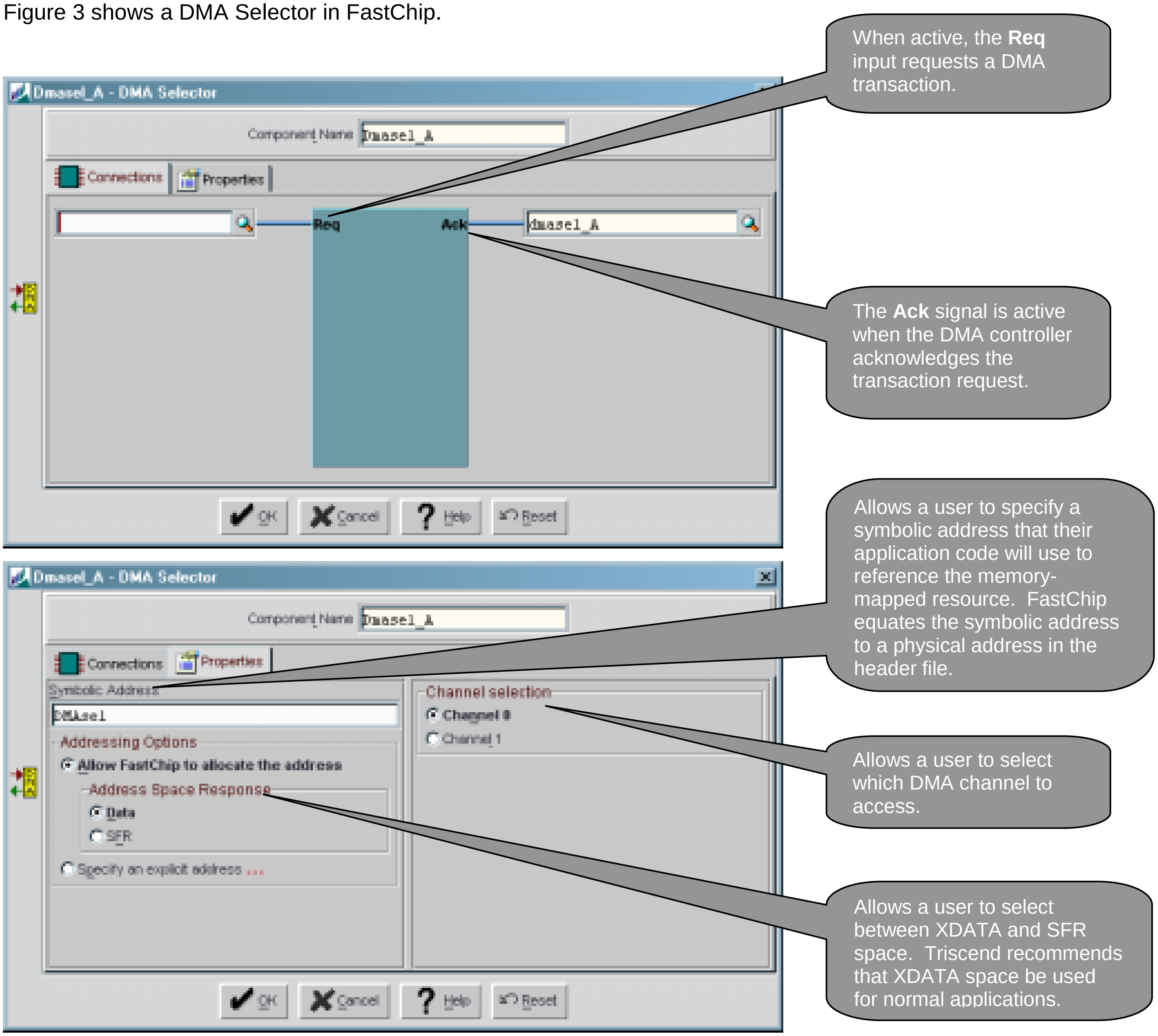
Figure 2: Relationship between E5 Dedicated DMA Resource and FastChip

DMA Selector Module

The DMA Selector is FastChip's version of the DMA Control Register (please see Figure 2). The DMA selector offers DMA services to functions within the CSL logic by providing bus-mastering services for CSL peripherals. The main features of the DMA Selector include:

- Providing request and acknowledge steering to a specific DMA channel
- Always one byte wide
- Exists in either Data or SFR space
- Accessible via a symbolic address

Figure 3 shows a DMA Selector in FastChip.



Dmasel_A - DMA Selector

Component Name: Dmasel_A

Connections | Properties

Req | Ack: dmasel_A

When active, the **Req** input requests a DMA transaction.

The **Ack** signal is active when the DMA controller acknowledges the transaction request.

Dmasel_A - DMA Selector

Component Name: Dmasel_A

Connections | Properties

Symbolic Address: DMAsel

Addressing Options

● Allow FastChip to allocate the address

Address Space Response

● Data

○ SFR

○ Specify an explicit address ...

Channel selection

● Channel 0

○ Channel 1

Allows a user to specify a symbolic address that their application code will use to reference the memory-mapped resource. FastChip equates the symbolic address to a physical address in the header file.

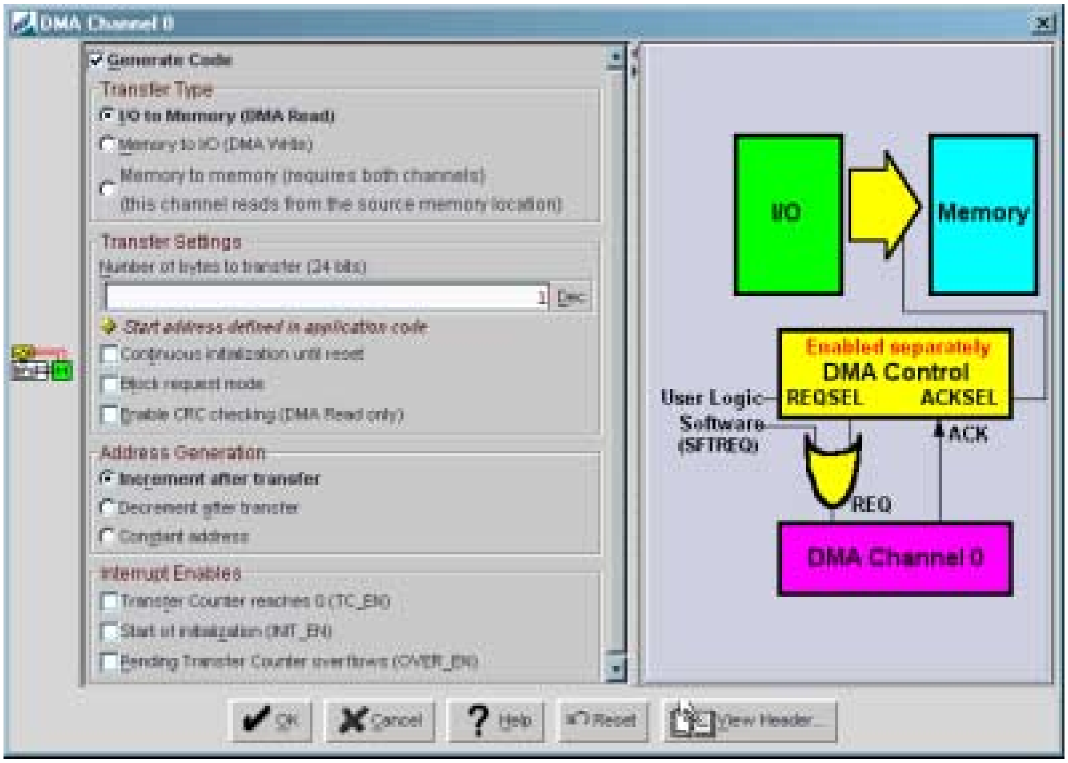
Allows a user to select which DMA channel to access.

Allows a user to select between XDATA and SFR space. Triscend recommends that XDATA space be used for normal applications.

Figure 3: A DMA Selector in FastChip

DMA Channel Module

FastChip provides two DMA Channel Controller Modules (DMA_0 and DMA_1), one for each channel. Figure 4 shows a DMA Channel Controller Module in FastChip.



DMA Channel 0

☑ Generate Code

Transfer Type

● I/O to Memory (DMA Read)

○ Memory to I/O (DMA Write)

○ Memory to memory (requires both channels)
(this channel reads from the source memory location)

Transfer Settings

Number of bytes to transfer (24 bits): 1 Dec

Start address defined in application code

□ Continuous initialization until reset

□ Block request mode

□ Enable CRC checking (DMA Read only)

Address Generation

● Increment after transfer

○ Decrement after transfer

○ Constant address

Interrupt Enables

□ Transfer Counter reaches 0 (TC_EN)

□ Start of initialization (INIT_EN)

□ Pending Transfer Counter overflows (OVER_EN)

Figure 4: DMA Channel 0 in FastChip

The DMA Channel Module is composed of four major control sections, each of which is described below.

Transfer Type

The Transfer Type control allows a user to define the type of DMA transaction for a specified channel.

Transfer Type	Data Read From	Data Written To
DMA Read → I/O to Memory	CSL	memory
DMA Write → Memory to I/O	memory	CSL
Memory-to-Memory	memory	memory

Table 1: DMA Transfer Types

It is important to remember that a CSL location is accessed via a DMA Selector. A memory location is any addressable location, however, the DMA controllers operate in a 32-bit physical address space, not the 16-bit logical space. See the example project (Appendix A) for a software routine to convert 16-bit logical addresses to 32-bit physical addresses. The example at the end of this app note will address this issue.

For **memory-to-memory transfers**, DMA Channel 0 is used to control the transfer and is responsible for reading the source location whereas Channel 1 is responsible for writing to the destination location.

Transfer Settings

The Transfer Settings area allows a user to define a **24-bit transfer count,** a **32-bit start address, set continuous initialization until reset,** and **block request mode.** Each of these options is explained in detail in the E5 Data Sheet and in FastChip help. It is important to remember that the 32-bit **start address** must be defined in the application code. See Figure 5 for an example.

```
// Must set up the Starting Address registers,
// enable and initialize the channel before starting any transfers.
// DMA Start Address is a 32-bit _physical_ address value stored
// in the following locations
// DMASADR0_0 = Start_Address[7:0]
// DMASADR0_1 = Start_Address[15:8]
// DMASADR0_2 = Start_Address[23:16]
// DMASADR0_3 = Start_Address[31:24]
```

Figure 5: Excerpt of C code needed to define the starting address for a DMA transfer

Address Generation

The Address Generation area allows the user to decide how the DMA source address is modified after each transfer in order to set up for the following transfer. There are three possible options:

- **Increment after transfer** – increments the current address by one byte.
- **Decrement after transfer** – decrements the current address by one byte.
- **Constant address** – maintains the current address value for the next transfer.

Interrupt Enables

Each DMA channel has its own set of interrupt sources and interrupt enable controls. The user can choose the source of an interrupt by clicking on the appropriate choice in the Interrupt Enables area. Interrupts may be generated for the following cases:

- **Transfer terminal count** – transfer counter reaches 0. This indicates that the desired number of bytes has been transferred.
- **Transfer initialization** – indicates a DMA channel has been initialized including the cases when the *continuous initialization until reset* mode is used.
- **Pending request overflow** – indicates that the DMA has greater than 64K unserviced requests.

Clicking on the **View Header** button displays the code that is generated by FastChip.

Appendix A: Example Design Project – DMA Wires

Before Getting Started

Before starting this tutorial make sure that you have the following items. Please refer to the *Triscend E5 Development Board* for information on all the items included with the Triscend E5 Evaluation Kit

- Latest version of FastChip properly installed on your computer.
- Keil uVision2 properly installed on your computer.
- A Triscend E5 Development Board with a download cable.

Scope

This example design project introduces you to the basic operations and the design flow of the Triscend FastChip development system. It guides you through a sample project using the E5's DMA controllers.

It will provide an introduction to hardware design with CSoC devices, specifically utilizing the DMA controller modules and some of their main features.

The audience for this example design project is an intermediate user. First time users may first want to review the FastChip online tutorial. The tutorial is available by opening FastChip, then making the menu selection **Help → Triscend CSoC Learning Center (Tutorial)**.

In this tutorial you will learn how to:

- Add IP modules from the FastChip library, including the DMA Selectors.
- Connect the signals between the IP modules.
- Set up CSoC dedicated resources.
- Assign I/O pins
- Generate header and source file for your C application
- Bind the project.
- Download the CSL configuration and the C application image to external Flash memory, via the CSoC device.

Design Overview

This FastChip example project uses the DMA to emulate a set of wires. Information from a DIP switch is displayed on an 7-segment LED display. Using the DMA channels, the same information is sent to the second LED display.

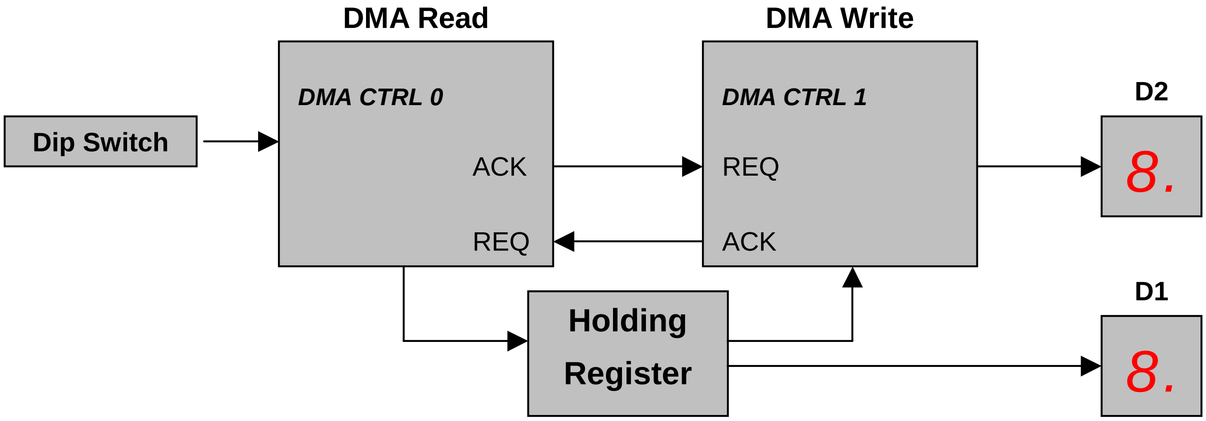


Figure A1: Block Diagram

Configuring the DMA Channels

This project makes use of both DMA channels, DMA_0 and DMA_1. FastChip's interface to the E5's dedicated resources is shown in Figure A2.

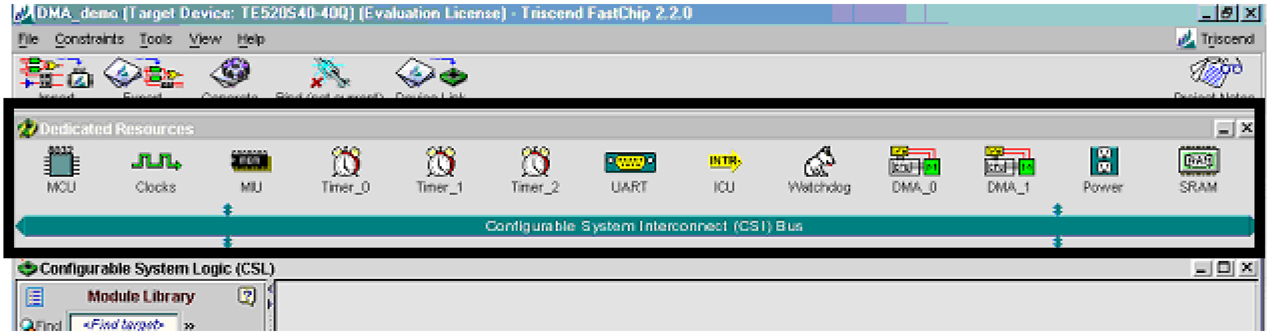

DMA_demo (Target Device: TE520S40-40Q) (Evaluation License) - Triscend FastChip 2.2.0
File
Constraints
Tools
View
Help
Triscend
Dedicated Resources
MCU
Clocks
MIU
Timer_0
Timer_1
Timer_2
UART
ICU
Watchdog
DMA_0
DMA_1
Power
SRAM
Configurable System Interconnect (CSI) Bus
Configurable System Logic (CSL)
Module Library
Find
<Find target>

Figure A2: Dedicated resources window in FastChip.

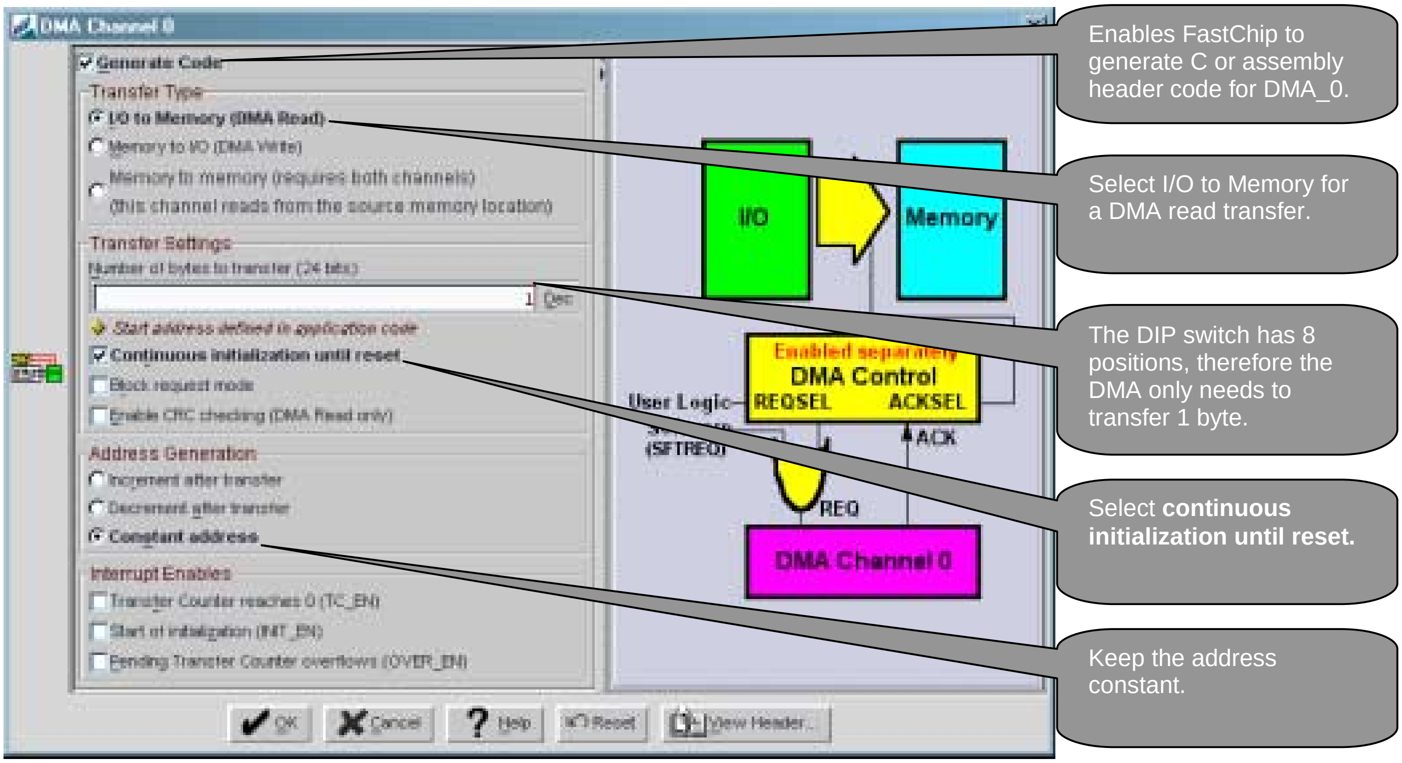
Configuring DMA_0

Click DMA_0 from the dedicated resources window to configure DMA Channel 0. For this project, DMA_0 needs to be configured for **I/O to Memory** transfers, because data from the DIP switch needs to be transferred to a memory mapped register (Holding_Reg).

For this project, DMA_0 needs to be **continuously initialized until reset** so that DMA_0 and DMA_1 can feed off each other continuously. This option can be set in the **Transfer Settings** section (see Figure A3).

Since the source of the DMA transfer is a single memory-mapped register, we need to keep the starting address constant. This option can be set in the **Address Generation** section (see Figure A3).

For this project, DMA_0 should be configured as shown in Figure A3.



DMA Channel 0

☑ Generate Code

Transfer Type

● I/O to Memory (DMA Read)

○ Memory to I/O (DMA Write)

○ Memory to memory (requires both channels)
(this channel reads from the source memory location)

Transfer Settings

Number of bytes to transfer (24 bits)

1 Dec

Start address defined in application code

☑ Continuous initialization until reset

☐ Block request mode

☐ Enable CRC checking (DMA Read only)

Address Generation

○ Increment after transfer

○ Decrement after transfer

● Constant address

Interrupt Enables

☐ Transfer Counter reaches 0 (TC_EN)

☐ Start of initialization (INIT_EN)

☐ Pending Transfer Counter overflows (OVER_EN)

Enables FastChip to generate C or assembly header code for DMA_0.

Select I/O to Memory for a DMA read transfer.

The DIP switch has 8 positions, therefore the DMA only needs to transfer 1 byte.

Select **continuous initialization until reset.**

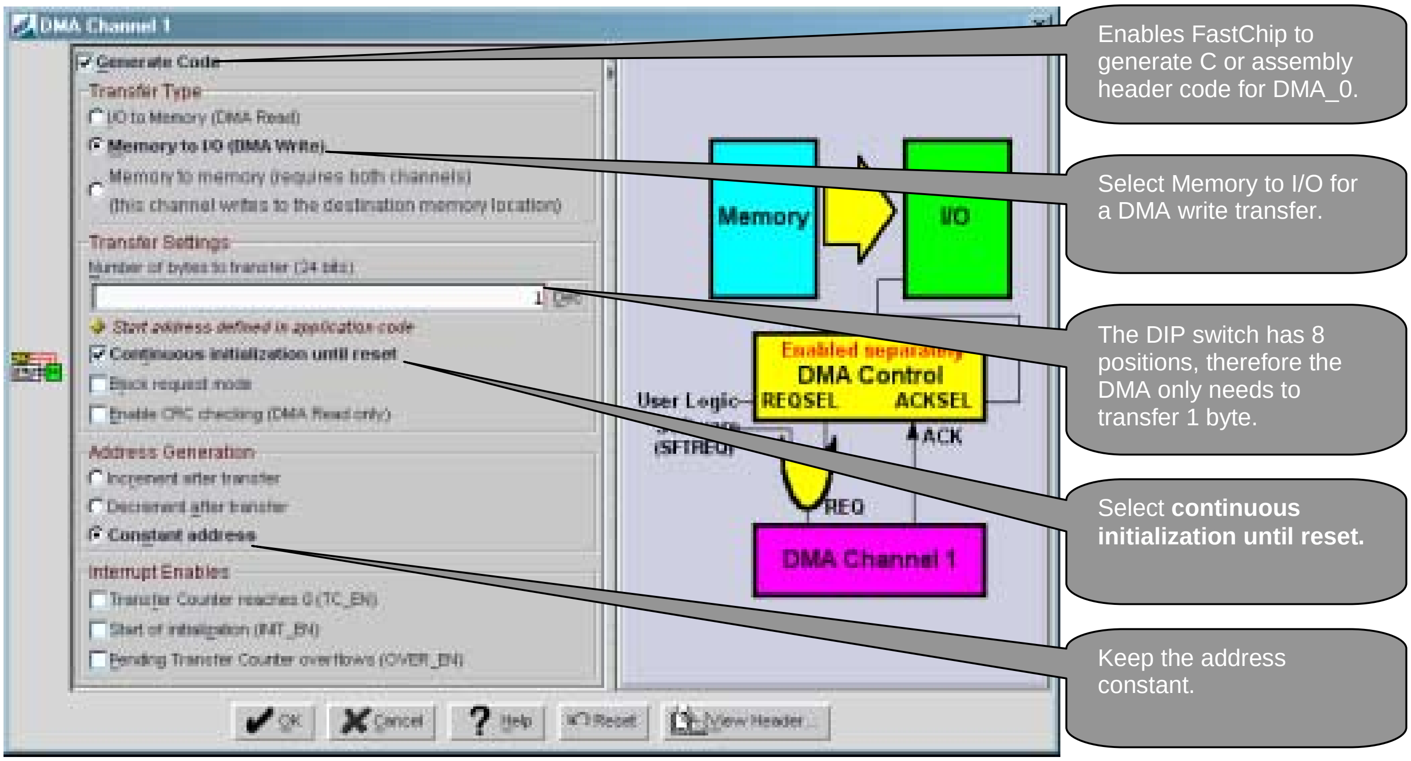
Keep the address constant.

Figure A3: Configuring DMA_0

Checking the **Generate Code** box initiates a header code generation session. All the DMA_0's registers will be set appropriately depending on the options selected in the configuration window. It is important to note that **the start address must be defined in the application code**.

Configuring DMA_1

DMA Channel 1 should be configured exactly like DMA_0, except DMA_1 needs to be configured for **Memory to I/O** transfers. See Figure A4 to see how to configure DMA_1.



DMA Channel 1

☑ Generate Code

Transfer Type
- ○ I/O to Memory (DMA Read)
- ● Memory to I/O (DMA Write)
- ○ Memory to memory (requires both channels) (this channel writes to the destination memory location)

Transfer Settings

Number of bytes to transfer (24 bits): 1 Dec

Start address defined in application code

- ☑ Continuous initialization until reset
- □ Block request mode
- □ Enable CRC checking (DMA Read only)

Address Generation
- ○ Increment after transfer
- ○ Decrement after transfer
- ● Constant address

Interrupt Enables
- □ Transfer Counter reaches 0 (TC_EN)
- □ Start of initialization (INIT_EN)
- □ Pending Transfer Counter overflows (OVER_EN)

Enables FastChip to generate C or assembly header code for DMA_0.

Select Memory to I/O for a DMA write transfer.

The DIP switch has 8 positions, therefore the DMA only needs to transfer 1 byte.

Select **continuous initialization until reset.**

Keep the address constant.

Figure A4: Configuring DMA_1

Selecting and Configuring IP Modules

For this project, six different modules are instantiated directly from the Triscend IP module library. The following modules will be used in this example project:

- DMA Selectors – used for interaction between the CSL and the DMA channels.
- Command Register – used for the holding register.
- Data Write – write data from the CSI bus' point of view.
- Data Read – read data from the CSI bus' point of view.
- Inputs – used to input data from the DIP switch.
- Outputs – used to output data to the 7-segment displays.

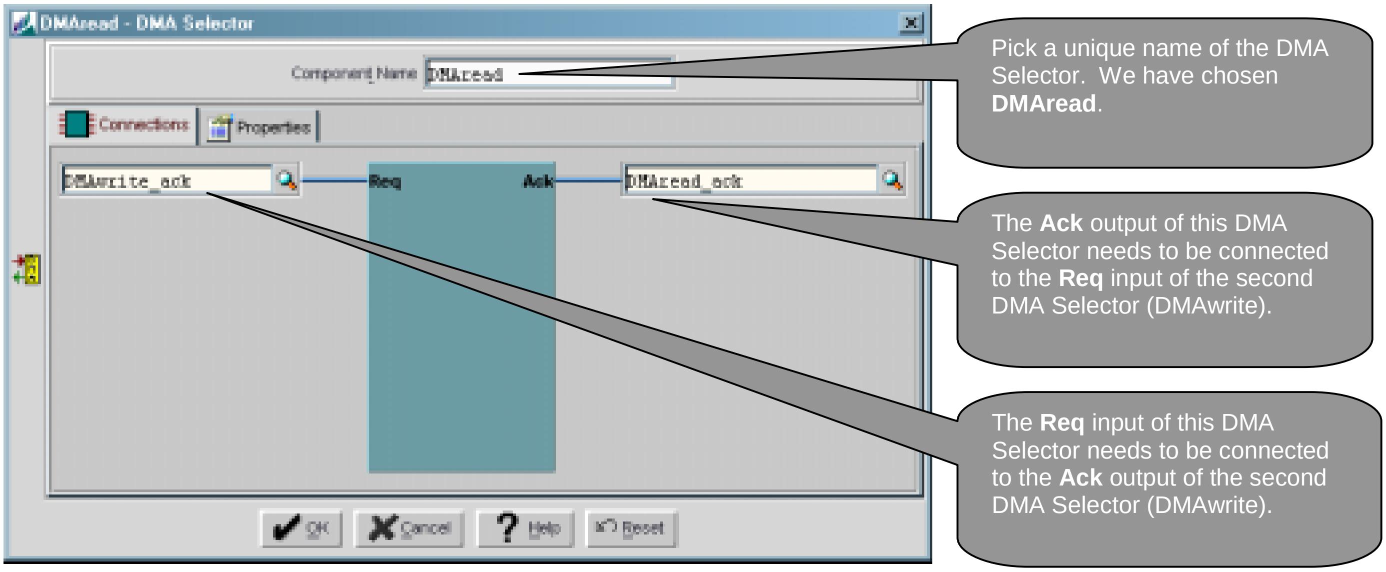
DMA Selectors

Two DMA Selectors are needed for this project – one for DMA Channel 0 (DMA_0) and the other for DMA Channel 1 (DMA_1). The modules can be found in the module library tree under **CSI Bus → Selectors**.

Each IP module has a **Component Name** which can the designer can choose. **The Component Name does not affect the functionality of the design in any way. It is used only to label the modules.** We will name the two selectors **DMAread** and **DMAwrite**.

DMAread

Drag and drop the **DMA Selector** into the CSL window and click the module to access **Connections** and **Properties** dialog boxes. Figure A5 illustrates the **Connections** dialog box for a **DMA Selector**.



DMAread - DMA Selector

Component Name: DMAread

DMAwrite_ack — Req | Ack — DMAread_ack

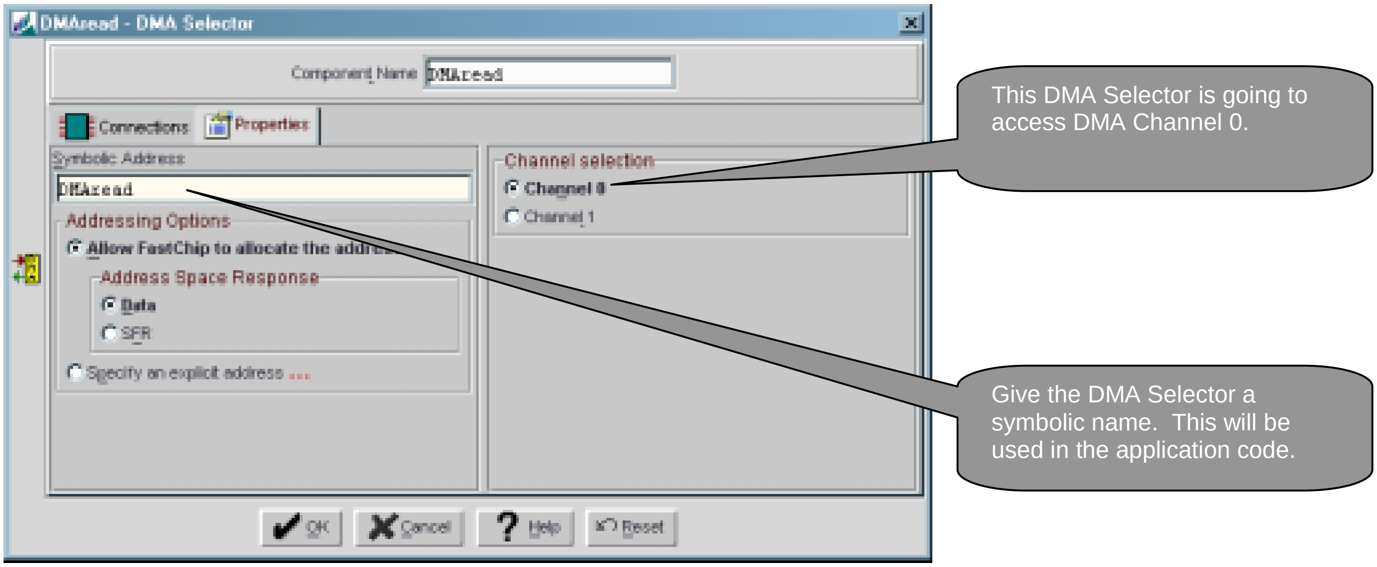
Pick a unique name of the DMA Selector. We have chosen **DMAread**.

The **Ack** output of this DMA Selector needs to be connected to the **Req** input of the second DMA Selector (DMAwrite).

The **Req** input of this DMA Selector needs to be connected to the **Ack** output of the second DMA Selector (DMAwrite).

Figure A5: Configuring the connections of a DMA Selector for DMA_0

After configuring the connections as shown in Figure A5 click on the **Properties** tab to configure the DMA Selector's properties. Figure A6 illustrates the **Properties** dialog box for a **DMA Selector**.



DMAread - DMA Selector

Component Name: DMAread

Symbolic Address: DMAread

Addressing Options
- ● Allow FastChip to allocate the addre
  - Address Space Response
    - ● Data
    - ○ SFR
- ○ Specify an explicit address ...

Channel selection
- ● Channel 0
- ○ Channel 1

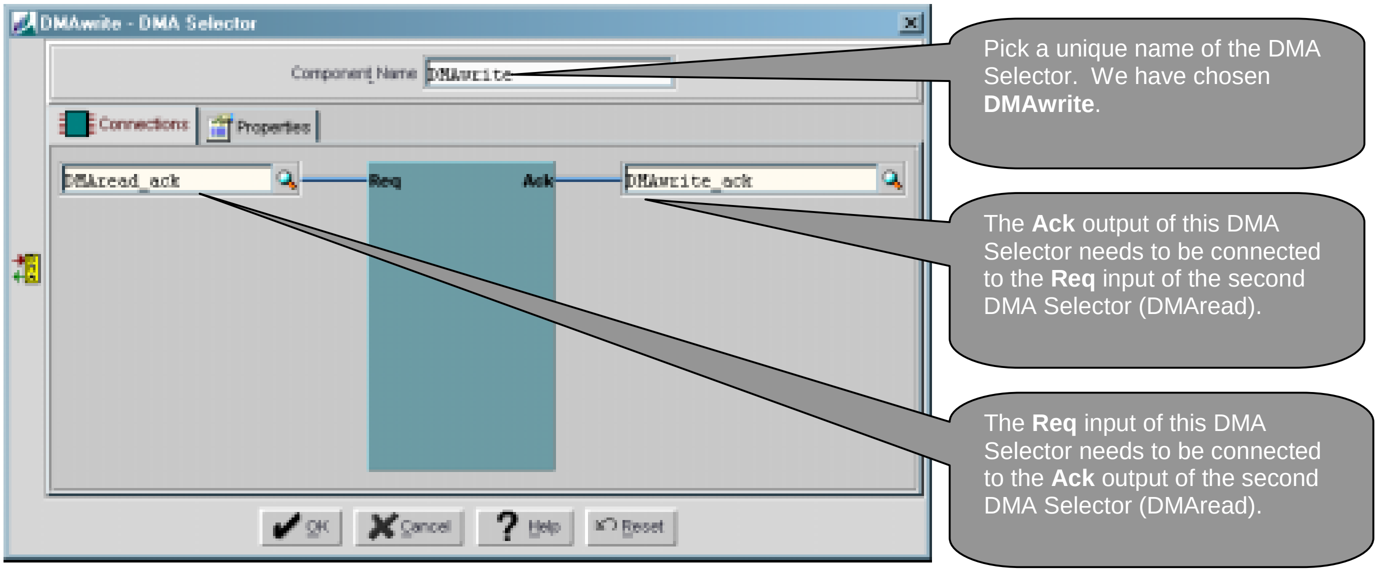
This DMA Selector is going to access DMA Channel 0.

Give the DMA Selector a symbolic name. This will be used in the application code.

Figure A6: Configuring the properties of a DMA Selector for DMA_0

DMAwrite

Drag and drop another **DMA Selector** into the CSL window. As before, click the module to access **Connections** and **Properties** dialog boxes. Figure A7 illustrates the **Connections** dialog box for a **DMA Selector**.



Component Name: DMAwrite

Req: DMAread_ack

Ack: DMAwrite_ack

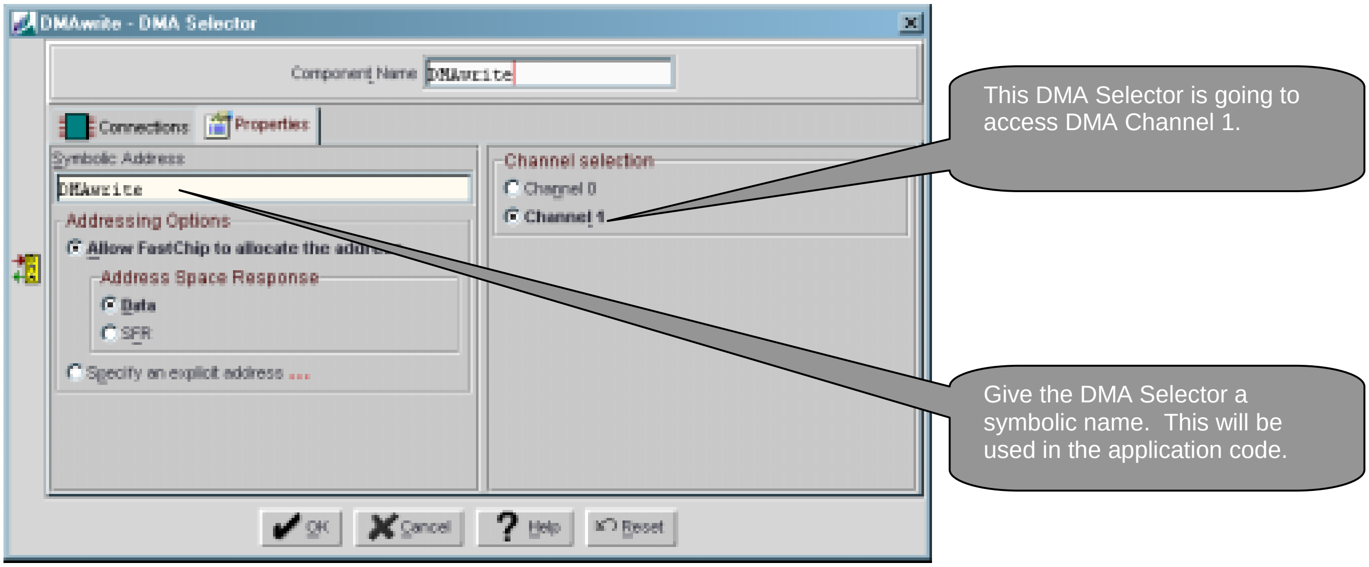
Pick a unique name of the DMA Selector. We have chosen **DMAwrite**.

The **Ack** output of this DMA Selector needs to be connected to the **Req** input of the second DMA Selector (DMAread).

The **Req** input of this DMA Selector needs to be connected to the **Ack** output of the second DMA Selector (DMAread).

Figure A7: Configuring the connections of a DMA Selector for DMA_1

After configuring the connections as shown in Figure A7 click on the **Properties** tab to configure the DMA Selector's properties. Figure A8 illustrates the **Properties** dialog box for a **DMA Selector**.



Component Name: DMAwrite

Symbolic Address: DMAwrite

Addressing Options
● Allow FastChip to allocate the addre
Address Space Response
● Data
○ SFR
○ Specify an explicit address ...

Channel selection
○ Channel 0
● Channel 1

This DMA Selector is going to access DMA Channel 1.

Give the DMA Selector a symbolic name. This will be used in the application code.

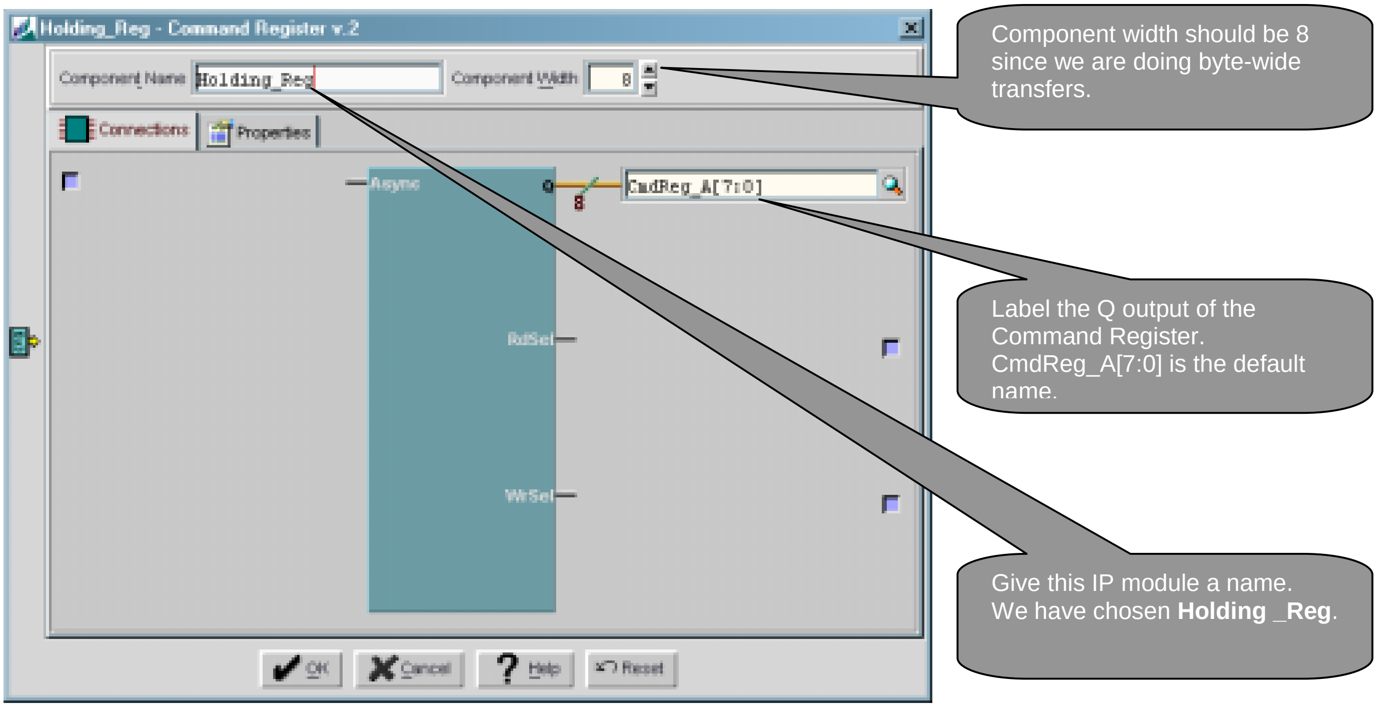
Figure A8: Configuring the properties of a DMA Selector for DMA_1

The DMA portion of this example project has been configured.

Command Register

We will instantiate a **Command Register** to act as a memory-mapped holding register for the DMA transfers. The Command Register can be found in the module library tree under **Peripherals →**

Control. As with the DMA Selectors, drag and drop a Command Register into the CSL window and configure the Command Register as shown in Figures A9 and A10.



Holding_Reg - Command Register v.2

Component Name: Holding_Reg

Component Width: 8

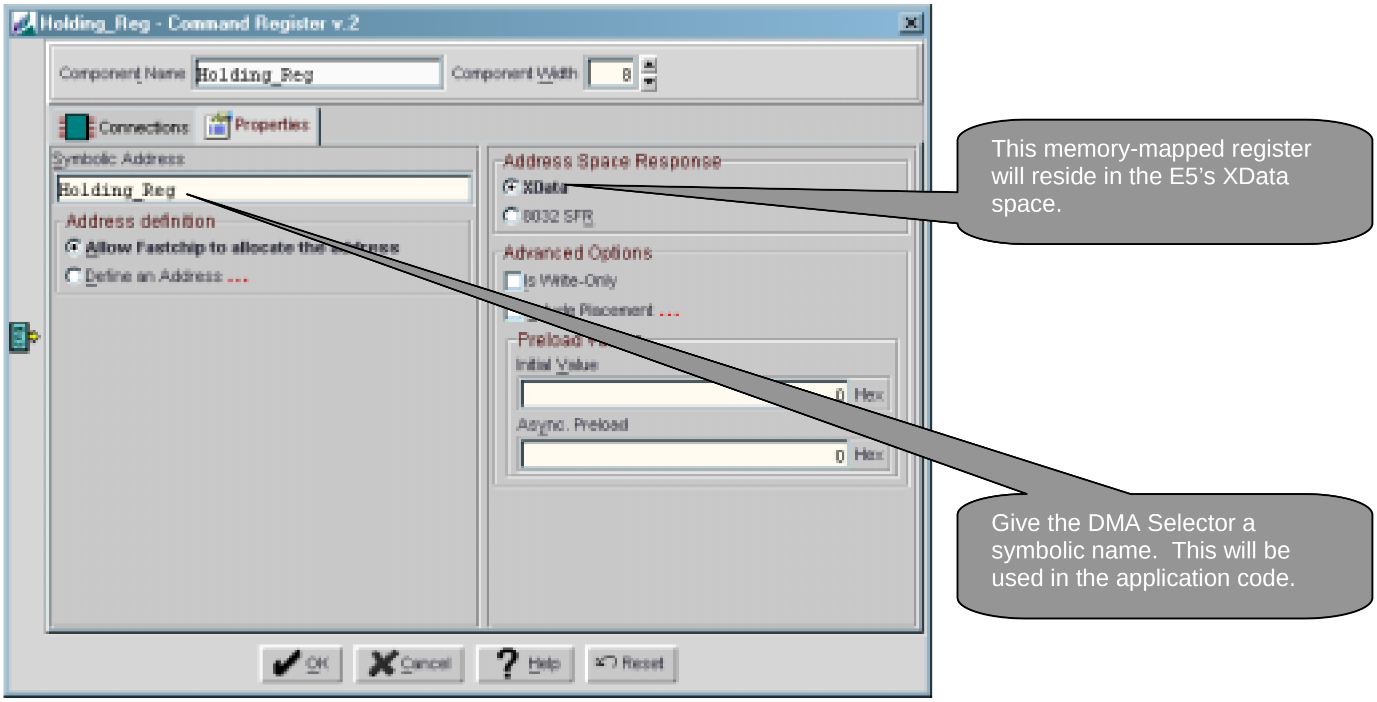
CmdReg_A[7:0]

Component width should be 8 since we are doing byte-wide transfers.

Label the Q output of the Command Register. CmdReg_A[7:0] is the default name.

Give this IP module a name. We have chosen **Holding _Reg**.

Figure A9: Configuring the connections of a Command Register



Holding_Reg - Command Register v.2

Component Name: Holding_Reg

Component Width: 8

Symbolic Address: Holding_Reg

Address definition

● Allow Fastchip to allocate the address

○ Define an Address ...

Address Space Response

● XData

○ 8032 SFR

Advanced Options

□ Is Write-Only

□ ...ole Placement ...

Preload Value

Initial Value: 0 Hex

Async. Preload: 0 Hex

This memory-mapped register will reside in the E5's XData space.

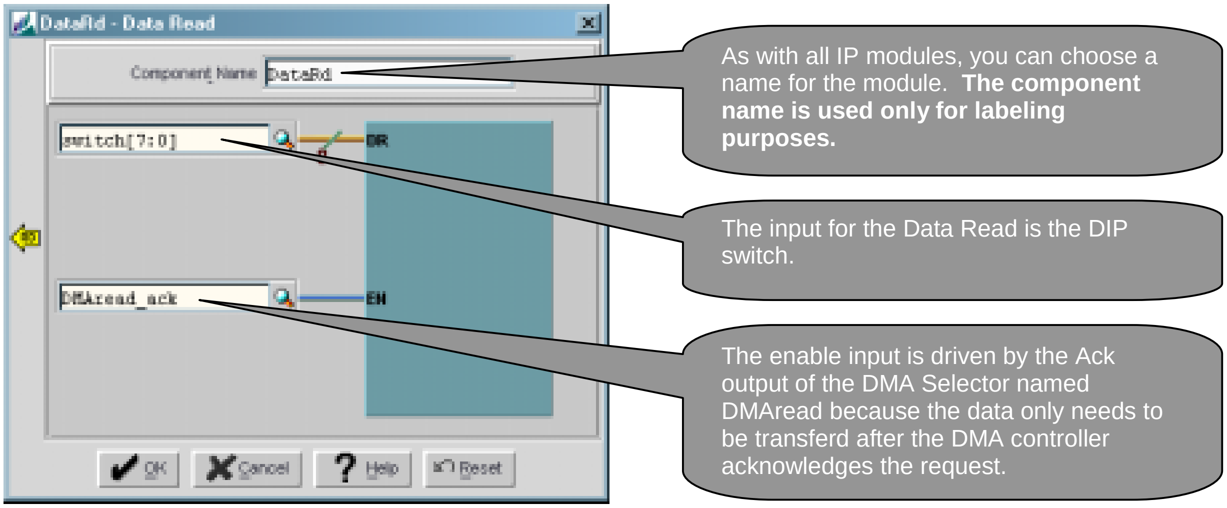
Give the DMA Selector a symbolic name. This will be used in the application code.

Figure A10: Configuring the properties of a Command Register

CSI Bus Connections

Data Read

Since we are getting data from the DIP switch, we need a way to place the data on the CSI bus. The **Data Read** IP module is used exactly for this purpose. It can be found in the module library tree under **CSI Bus → Data Bus.** Figure A11 illustrates the Data Read IP module.



Component Name: DataRd

switch[7:0]

DMAread_ack

As with all IP modules, you can choose a name for the module. **The component name is used only for labeling purposes.**

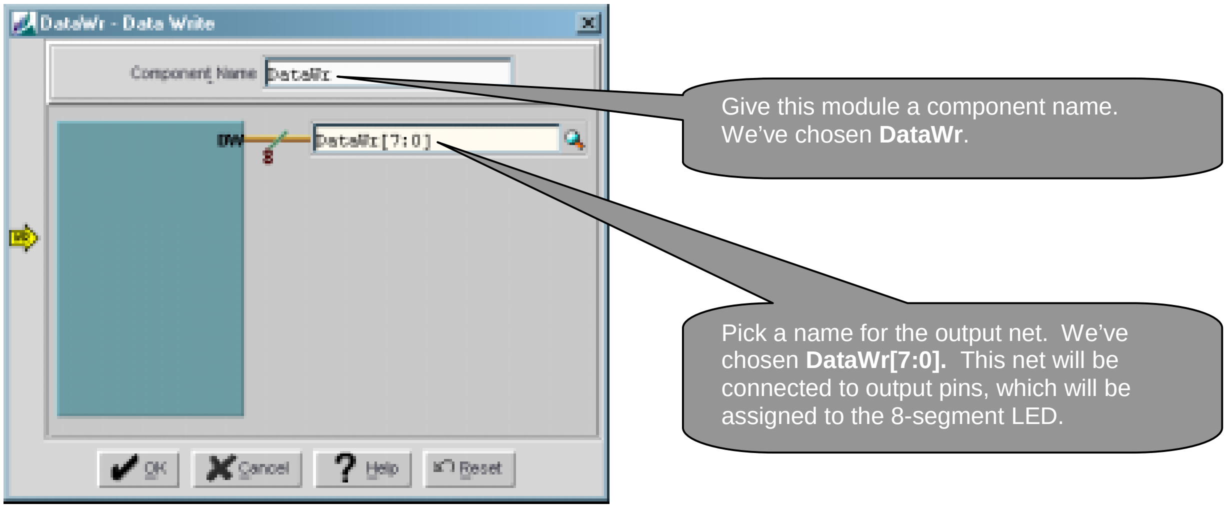
The input for the Data Read is the DIP switch.

The enable input is driven by the Ack output of the DMA Selector named DMAread because the data only needs to be transferd after the DMA controller acknowledges the request.

Figure A11: Data Read

Data Write

The DMA controller (**DMAwrite**) will automatically place the transferred data onto the CSI bus. However, we need a way to get the data from the CSI bus and send it to the 8-segment LED. The **Data Write** module, the opposite of a Data Read module**,** is used for this purpose. Figure A12 shows the Data Write module.



Component Name: DataWr

DataWr[7:0]

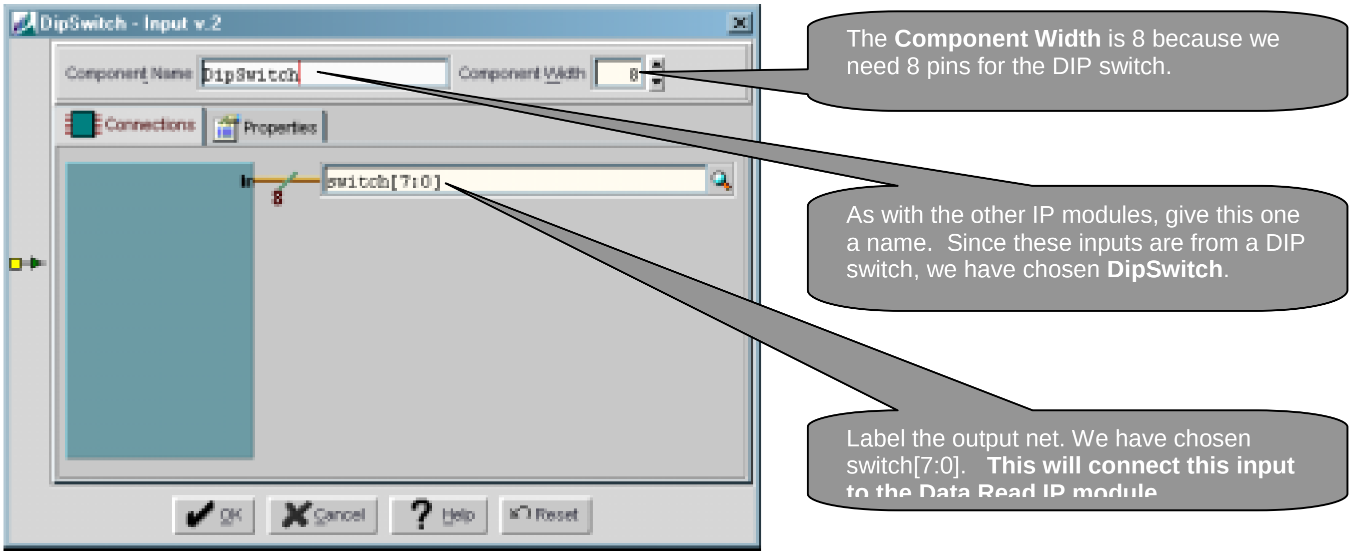
Give this module a component name. We've chosen **DataWr**.

Pick a name for the output net. We've chosen **DataWr[7:0].** This net will be connected to output pins, which will be assigned to the 8-segment LED.

Figure A12: Data Write

Inputs

For this project we need to instantiate 8 input pins. Drag and drop **one input IP module** into the CSL and configure it as shown in Figure A13. The **IP module** can be found in the module library tree under **CSI Bus → I/O**.



DipSwitch - Input v.2

Component Name: DipSwitch

Component Width: 8

switch[7:0]

The **Component Width** is 8 because we need 8 pins for the DIP switch.

As with the other IP modules, give this one a name. Since these inputs are from a DIP switch, we have chosen **DipSwitch**.

Label the output net. We have chosen switch[7:0]. **This will connect this input to the Data Read IP module**

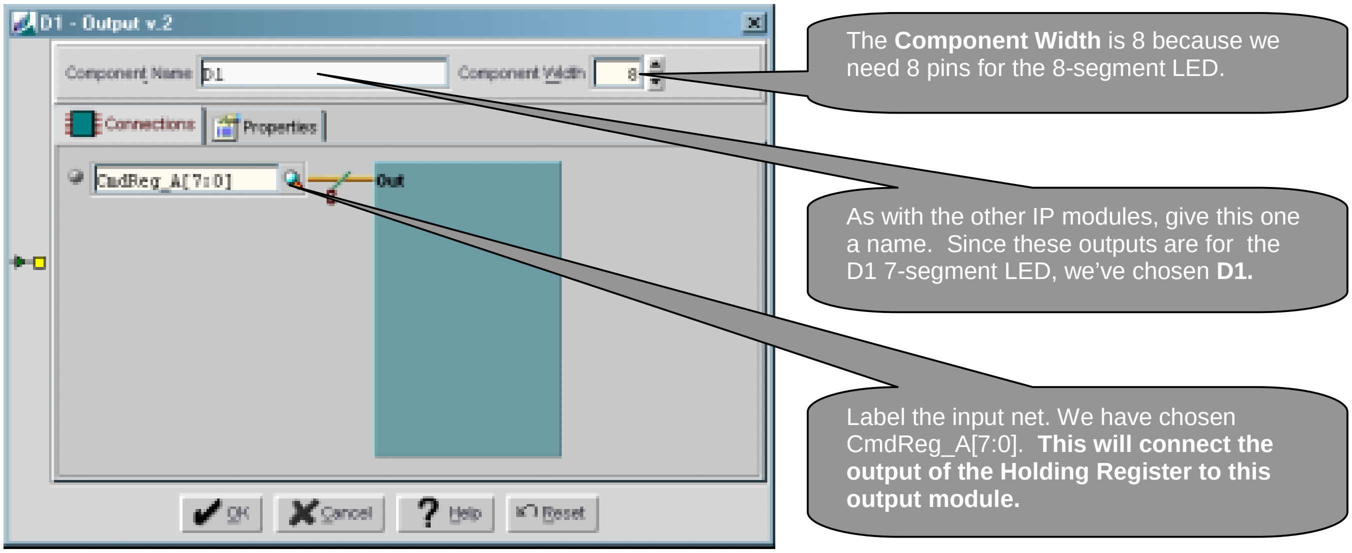
Figure A13: Configuring the connections of an input.

The properties of an input IP module can also be configured, however, for this example project we have left them at their default settings.

Outputs

This project requires 16 output pins to connect to the two 7-segment LED displays. Drag and drop **two output IP modules** into the CSL and configure them as shown in Figures A14 – A16. The **IP module** can be found in the module library tree under **CSI Bus → I/O**.

Output for D1



D1 - Output v.2

Component Name: D1

Component Width: 8

CmdReg_A[7:0]

The **Component Width** is 8 because we need 8 pins for the 8-segment LED.

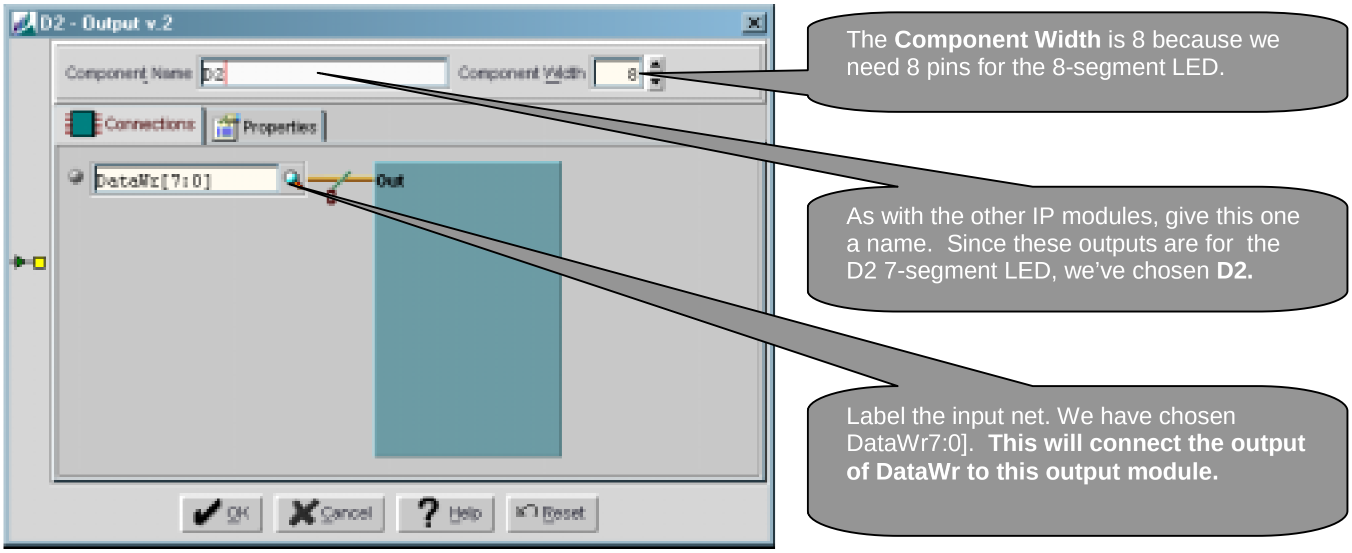
As with the other IP modules, give this one a name. Since these outputs are for the D1 7-segment LED, we've chosen **D1.**

Label the input net. We have chosen CmdReg_A[7:0]. **This will connect the output of the Holding Register to this output module.**

Figure A14: Configuring the connections of an output for D1

For the D1 input, the Properties do not need to be configured. Leave them at their default settings.

Output for D2



D2 - Output v.2

Component Name: D2

Component Width: 8

Connections | Properties

DataWr[7:0] → Out

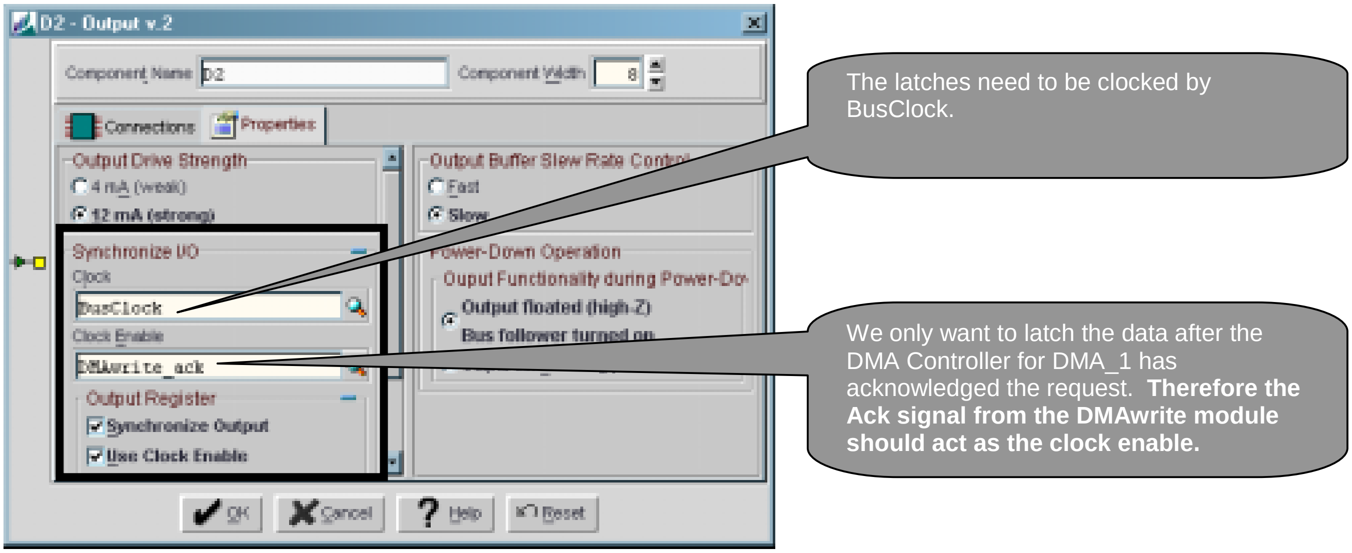
The **Component Width** is 8 because we need 8 pins for the 8-segment LED.

As with the other IP modules, give this one a name. Since these outputs are for the D2 7-segment LED, we've chosen **D2.**

Label the input net. We have chosen DataWr7:0]. **This will connect the output of DataWr to this output module.**

Figure A15: Configuring the connections of an output for D2

For this example project, we would like to latch the data into registers. Configuring the properties of the output **IP module** can do this as shown in Figure A16.



D2 - Output v.2

Component Name: D2

Component Width: 8

Connections | Properties

Output Drive Strength
○ 4 mA (weak)
● 12 mA (strong)

Synchronize I/O
Clock: BusClock
Clock Enable: DMAwrite_ack

Output Register
☑ Synchronize Output
☑ Use Clock Enable

Output Buffer Slew Rate Control
○ Fast
● Slow

Power-Down Operation
Ouput Functionality during Power-Do
● Output floated (high-Z)
Bus follower turned on

The latches need to be clocked by BusClock.

We only want to latch the data after the DMA Controller for DMA_1 has acknowledged the request. **Therefore the Ack signal from the DMAwrite module should act as the clock enable.**

Figure A16: Configuring the properties for D2

Configure I/O Constraints and Memory Interface Unit (MIU)

In order for the example project to function properly, the input and output pads need to be assigned to specific package pins. The FastChip I/O Editor provides an intuitive interface to assign I/O pads to the package pin. Select **I/O Editor** from the **Constraints** menu. Once the I/O window comes up, click Full Screen to expand the window.

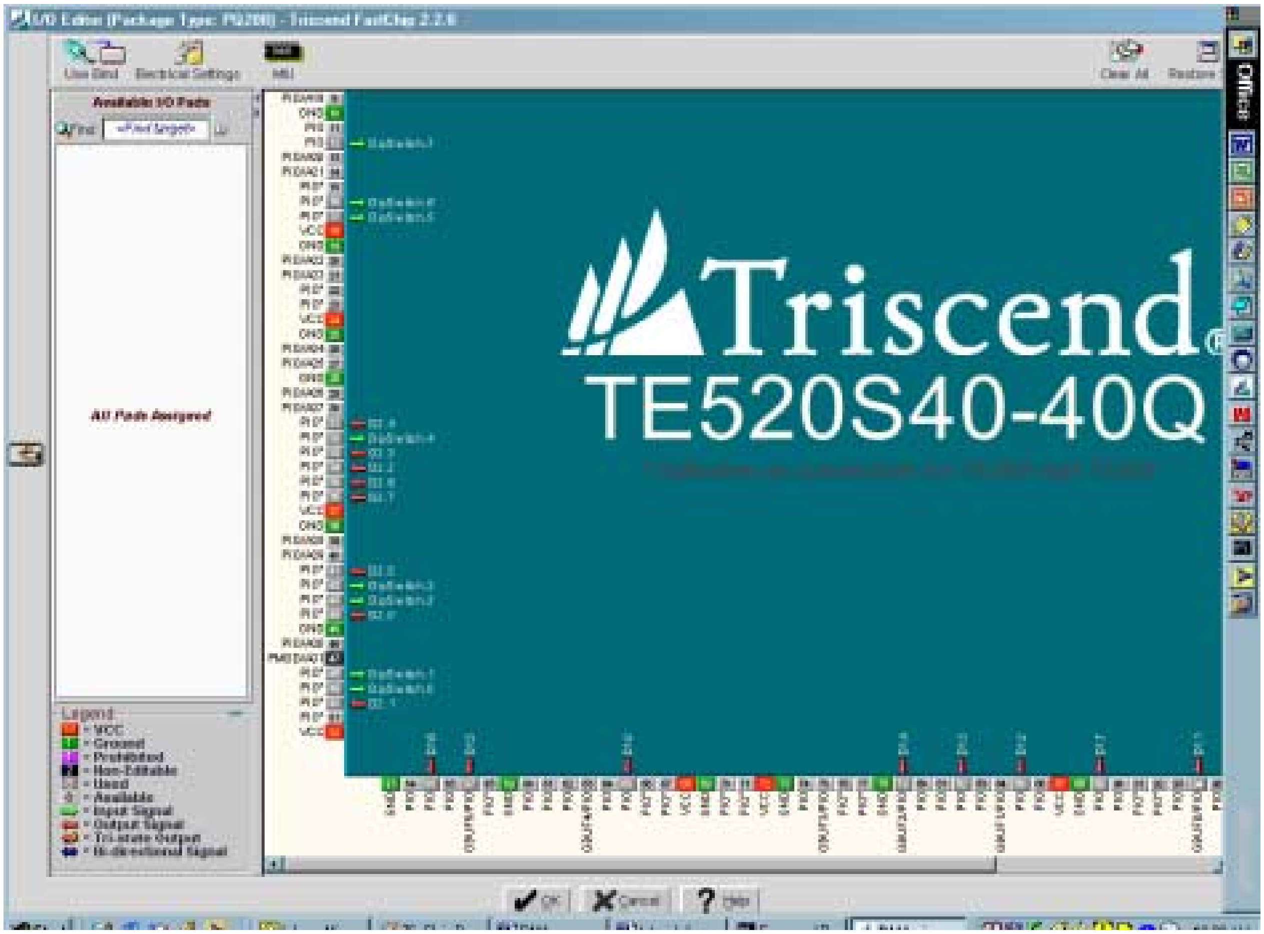


Figure A17: FastChip's I/O Editior with all pads assigned

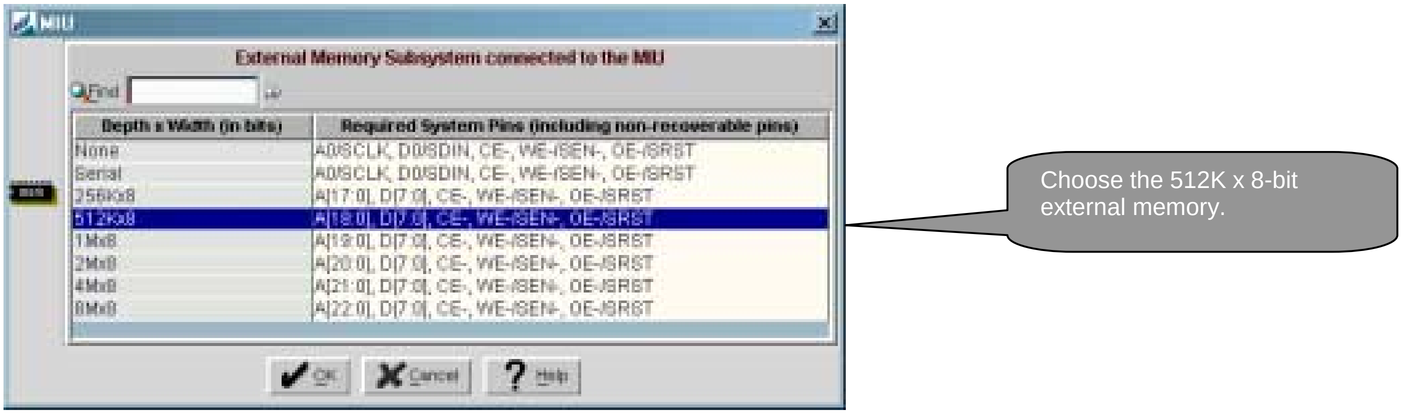
You can view all unassigned I/O pads from the Available I/O Pads area on the left. To assign a pad to a package pin, simply drag the pad into the chip package image window and drop it onto the desired package pin. If you make an error, you can move the pad by dragging and dropping it onto another pin. Assign all pads to their respective package pins according to the following table. Click **OK** after you finished assigning pads.

Module	Pad Name	Pin Number
D1	D1.0	65
	D1.1	94
	D1.2	85
	D1.3	82
	D1.4	79
	D1.5	57
	D1.6	55
	D1.7	89
D2	D2.0	44
	D2.1	50
	D2.2	34
	D2.3	33
	D2.4	31
	D2.5	41
	D2.6	35
	D2.7	36
DIP SWITCH	DipSwitch.7	12
	DipSwitch.6	16
	DipSwitch.5	17
	DipSwitch.4	32
	DipSwitch.3	42
	DipSwitch.2	43
	DipSwitch.1	48
	DipSwitch.0	49

Table A1: I/O Pin Assignment

Configuring the MIU

For this example we are going to use the E5 development board Flash memory to configure the CSoC device. To perform this we need to assign the external memory I/O pins to the device. Click on **MIU** in the I/O Editor window, shown in Figure A17. Select the external 512Kx8-bit external memory, as shown in Figure A18. Note, alternatively the CSoC can be configured directly from the download cable. In this case no external memory I/O pins need to be assigned



MIU

External Memory Subsystem connected to the MIU

Find

Depth x Width (in bits)	Required System Pins (including non-recoverable pins)
None	A0/SCLK, D0/SDIN, CE-, WE-/SEN-, OE-/SRST
Serial	A0/SCLK, D0/SDIN, CE-, WE-/SEN-, OE-/SRST
256Kx8	A[17:0], D[7:0], CE-, WE-/SEN-, OE-/SRST
512Kx8	A[18:0], D[7:0], CE-, WE-/SEN-, OE-/SRST
1Mx8	A[19:0], D[7:0], CE-, WE-/SEN-, OE-/SRST
2Mx8	A[20:0], D[7:0], CE-, WE-/SEN-, OE-/SRST
4Mx8	A[21:0], D[7:0], CE-, WE-/SEN-, OE-/SRST
8Mx8	A[22:0], D[7:0], CE-, WE-/SEN-, OE-/SRST

Choose the 512K x 8-bit external memory.

Figure A18: Configuring the MIU

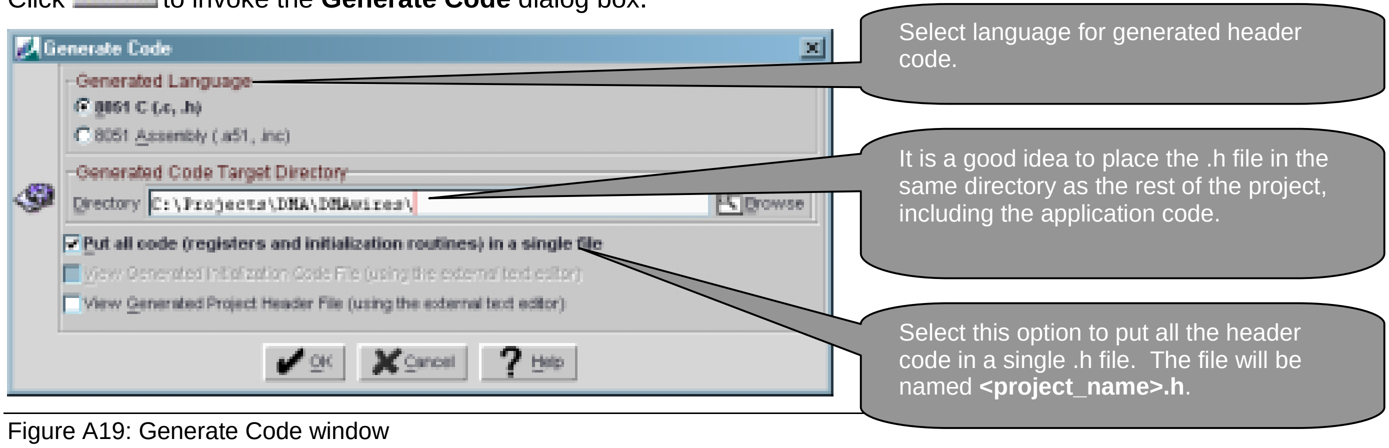
Generate Code for Your Compiler

FastChip can generate C or assembly source code with declarations for the addressable registers in the configurable logic, such as the symbolic address **Holding_Reg** in the **Holding_Reg** command register IP module.

In addition, you can instruct FastChip to generate initialization functions for the dedicated resources. Some of the dedicated resource dialog boxes allow you to specify the desired startup state of the dedicated resources. You can selectively enable the **Generate Code** option to instruct FastChip to include functions for setting up the special function registers (SFR) associated with that dedicated resource in the generated source code.

At this time, no other Dedicated Resources need to be configured, and you can disable the **Generate Code** options in them to reduce the already modest code size of the generated source code.

Click Generate to invoke the **Generate Code** dialog box.



Generate Code

Generated Language

● 8051 C (.c, .h)
○ 8051 Assembly (.a51, .inc)

Generated Code Target Directory

Directory: C:\Projects\DMA\DMAwires\

☑ Put all code (registers and initialization routines) in a single file
□ View Generated Initialization Code File (using the external text editor)
□ View Generated Project Header File (using the external text editor)

Select language for generated header code.

It is a good idea to place the .h file in the same directory as the rest of the project, including the application code.

Select this option to put all the header code in a single .h file. The file will be named **<project_name>.h**.

Figure A19: Generate Code window

If the **Put all code (registers and initialization routines) in a single file** option is selected, FastChip generates only one file:

DMAwires.h <project_name>.h

This file contains necessary macro definitions and all the external address declarations for IP modules. It also contains prototypes for the dedicated resources init functions. This file also contains necessary macro definitions, address declarations for IP modules, and function definitions of the dedicated resources init functions.

For the E5 CSoC, if the project name is more than 8 characters long, the file name is truncated. This is because some of the 3rd party vendor software only support 8.3 notation file names.

Important

Subsequent generate sessions will overwrite the previous ***<project_name>.h*** *files with no warning. Make sure that you do not have any other files in the project directory with this name prior to performing a Generate.*

Bind the Project

In order to realize the logic in your FastChip project on the CSoC silicon, the design content will need to be processed into configuration information for the configurable system logic on the silicon. This process is called **Bind**. This process is analogous to the compile-link-load process when compiling a program or the map-place-route process for creating a gate array or FPGA.

Click Bind. Accept the default selection of **minimum bind effort** and click **OK**.

Bind is a computational-intensive process. FastChip applies several complex computing algorithms to optimize your design and to fit it into the CSL. Typical bind processing for a project like **DMAwires** takes three to five minutes on computers meeting the minimum system requirement. The status bar on the wait dialog box shows you how Bind is progressing.

The final result from Bind is an initialization file called **<project_name>.csl**. You will use this file later to combine it with your software application image to download to the CSoC.

Application Code

The following is the application code for this example project. The header file, **DMAwires.h**, has been included as a comment for your reference.

```
// Include header file created by FastChip

#include "DMAwires.h"


// Header file for routines that convert an 8032 logical

// address to a 32-bit physical address

#include "dmap.h"


void main() {


   unsigned char Update;


   // Call initialization routines created by FastChip.  These functions
```

```
  // set up the dedicated resources, including the DMA Controller.
  // The initialization routines do not setup the Start Address for the
  // DMA transfer, nor do they enable or initialize the channels.
  DMAwires_INIT ();

  // Write an arbitrary value to the 'Holding_Register', which connects to LED D2.
  // If LED D2 equals this value later, then we are not able to transfer data using
  // the DMA controller.
  Holding_Reg = 0x00;

  /* --------------------------------------------------------------------------- */
  /* SET UP DMA CHANNEL 0                                                        */
  /* --------------------------------------------------------------------------- */

*********************
* This section is a copy of the DMA 0 setup code created automatically by
* FastChip.  This code is executed as part of the DMAwired_INIT() routine.
*
* // DMACTRL0_0 (Address 0xff27)
* //    +-----+-----+-----+------+-----+-----+-----+-----+
* //    |  7  |  6  |  5  |   4  |  3  |  2  |  1  |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+======+=====+=====+=====+=====+
* //    | W/R-| PAIR|BLOCK|SFTREQ| CONT| INIT|  EN | CLR |   BIT NAMES
* //    +=====+=====+=====+======+=====+=====+=====+=====+
* //    |  0  |  0  |  0  |   0  |  1  |  0  |  0  |  0  |   REGISTER VALUE
* //    +-----+-----+-----+------+-----+-----+-----+-----+
*
* // DMACTRL0_1 (Address 0xff28)
* //    +-----+-----+-----+-----+-----+------+-----+-----+
* //    |  7  |  6  |  5  |  4  |  3  |   2  |  1  |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+=====+=====+======+=====+=====+
* //    |     |     |     |     |     |CRC_EN|ADRM1|ADRM0|   BIT NAMES
* //    +=====+=====+=====+=====+=====+======+=====+=====+
* //    |  0  |  0  |  0  |  0  |  0  |   0  |  0  |  1  |   REGISTER VALUE
* //    +-----+-----+-----+-----+-----+------+-----+-----+
*
* // DMAEINT0 (Address 0xff29)
* //    +-----+-----+-----+-----+-----+------+-------+-----+
* //    |  7  |  6  |  5  |  4  |  3  |   2  |   1   |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+=====+=====+======+=======+=====+
 * //    |     |     |     |     |     |OVR_EN|INIT_EN|TC_EN|   BIT NAMES
 * //    +=====+=====+=====+=====+=====+======+=======+=====+
 * //    |  0  |  0  |  0  |  0  |  0  |   0  |   0   |  0  |   REGISTER VALUE
 * //    +-----+-----+-----+-----+-----+------+-------+-----+
 *
 * // Set up the DMA Channel 0 control register
 * // -----------------------------------------
 * DMACTRL0_0 = 0x01;                // Clear DMA channel 0 control register
 * DMACTRL0_1 = 0x01;                // Set high-byte settings in control register
 * DMACTRL0_0 = 0x08;                // Set low-byte settings in control register
 *
 *
 * // Set up the transfer count
 * // -------------------------
 * // NOTE:  TRANSFER_COUNT = (BYTE_COUNT - 1)
 * DMASCNT0_0 = 0x00; // Move transfer cnt low-byte of DMA 0 tansfer cnt reg
 * DMASCNT0_1 = 0x00; // Move transfer cnt mid-byte of DMA 0 transfer cnt reg
 * DMASCNT0_2 = 0x00; // Move transfer cnt high-byte of DMA 0 transfer cnt reg
 *
 * // Enable the DMA Channel 0 interrupts
 * // -----------------------------------
 * DMAINT0    = 0x07;  // Clear DMA channel 0 interrupt flags by writing ones to bit
 * DMAEINT0   = 0x00;  // Enable DMA channel 0 interrupts
 *
 * // NOTE:  Must still set up the Starting Address registers and
 * // ====   enable and initialize the channel before starting any transfers.
 * // DMA Start Address is a 32-bit _physical_ address value stored in the following
 * // locations
 * // DMASADR0_0 = Start_Address[7:0]
 * // DMASADR0_1 = Start_Address[15:8]
 * // DMASADR0_2 = Start_Address[23:16]
 * // DMASADR0_3 = Start_Address[31:24]
 *

***********************************************************************************/
   // Convert the 8032's 16-bit logic address pointing to the XDATA
   // RAM buffer into a 32-bit physical address required by the
   // DMA controller.  Using this method allows this code to
   // operate regardless of the initialization method used.
   //
```

```
  // Put logical address in special variables LADDR_1 and LADDR0,
  // as shown below.
  LADDR_1    =  (unsigned char)((unsigned short)&Holding_Reg >> 8);
  LADDR_0    =  (unsigned char)((unsigned short)&Holding_Reg & 0xff);

  // Call the special routine that converts the 16-bit logical
  // address to a 32-bit physical address.  The 32-bit address is
  // provided in variables PADDR_3 through PADDR_0.
  xdata_logical_to_physical();

  // Setup DMA0's start address register
  DMASADR0_0 =  PADDR_0;
  DMASADR0_1 =  PADDR_1;
  DMASADR0_2 =  PADDR_2;
  DMASADR0_3 =  PADDR_3;

  // Enable the DMA channel 0 and initialize the transfer
  DMACTRL0_0 |= 0x06;

  // Enable DMA_READ control register to access DMA channel 0
  // NOTE:  Upper and lower nibbles MUST be identical.
  DMAread = 0x11;

  /* --------------------------------------------------------------------------- */
  /* SET UP DMA CHANNEL 1                                                        */
  /* --------------------------------------------------------------------------- */

**********************
* This section is a copy of the DMA 0 setup code created automatically by
* FastChip.  This code is executed as part of the DMAwired_INIT() routine.
*
* // DMACTRL1_0 (Address 0xff3b)
* //    +-----+-----+-----+------+-----+-----+-----+-----+
* //    |  7  |  6  |  5  |   4  |  3  |  2  |  1  |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+======+=====+=====+=====+=====+
* //    | W/R-| PAIR|BLOCK|SFTREQ| CONT| INIT|  EN | CLR |   BIT NAMES
* //    +=====+=====+=====+======+=====+=====+=====+=====+
* //    |  1  |  0  |  0  |   0  |  1  |  0  |  0  |  0  |   REGISTER VALUE
* //    +-----+-----+-----+------+-----+-----+-----+-----+
*
```

```
* // DMACTRL1_1 (Address 0xff3c)
* //    +-----+-----+-----+-----+-----+------+-----+-----+
* //    |  7  |  6  |  5  |  4  |  3  |   2  |  1  |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+=====+=====+======+=====+=====+
* //    |     |     |     |     |     |CRC_EN|ADRM1|ADRM0|   BIT NAMES
* //    +=====+=====+=====+=====+=====+======+=====+=====+
* //    |  0  |  0  |  0  |  0  |  0  |   0  |  0  |  1  |   REGISTER VALUE
* //    +-----+-----+-----+-----+-----+------+-----+-----+
*
* // DMAEINT1 (Address 0xff3d)
* //    +-----+-----+-----+-----+-----+------+-------+-----+
* //    |  7  |  6  |  5  |  4  |  3  |   2  |   1   |  0  |   BIT LOCATIONS
* //    +=====+=====+=====+=====+=====+======+=======+=====+
* //    |     |     |     |     |     |OVR_EN|INIT_EN|TC_EN|   BIT NAMES
* //    +=====+=====+=====+=====+=====+======+=======+=====+
* //    |  0  |  0  |  0  |  0  |  0  |   0  |   0   |  0  |   REGISTER VALUE
* //    +-----+-----+-----+-----+-----+------+-------+-----+
*
* // Set up the DMA Channel 1 control register
* // -----------------------------------------
*     DMACTRL1_0 = 0x01; // Clear DMA channel 1 control register
*     DMACTRL1_1 = 0x01; // Set high-byte settings in control register
*     DMACTRL1_0 = 0x88; // Set low-byte settings in control register
*
* // Set up the transfer count
* // -------------------------
* // NOTE:  TRANSFER_COUNT = (BYTE_COUNT - 1)
*     DMASCNT1_0 = 0x00; // Move transfer cnt low-byte of DMA 1 transfer cnt reg
*     DMASCNT1_1 = 0x00; // Move transfer cnt mid-byte of DMA 1 transfer cnt reg
*     DMASCNT1_2 = 0x00; // Move transfer cnt high-byte of DMA 1 transfer cnt reg
*
* // Enable the DMA Channel 1 interrupts
* // -----------------------------------
*     DMAINT1    = 0x07; // Clear DMA_1 interrupt flags by writing ones to bit
*     DMAEINT1   = 0x00; // Enable DMA channel 1 interrupts
*
* // NOTE:  Must still set up the Starting Address registers and
* // ====   enable and initialize the channel before starting any transfers.
* // DMA Start Address is a 32-bit _physical_ addr value stored in the following
* // locations:
 * // DMASADR1_0 = Start_Address[7:0]
 * // DMASADR1_1 = Start_Address[15:8]
 * // DMASADR1_2 = Start_Address[23:16]
 * // DMASADR1_3 = Start_Address[31:24]
 *
 * // DMA Channel 1 grabs value from 'Holding_Register' and displays it on LED D1.
 * // Load the start address of DMA Channel 1.
 * DMASADR1_3 = 0x88;
 * DMASADR1_2 = 0x10;
 * DMASADR1_1 = 0xef;
 * DMASADR1_0 = 0xfc;
********************************************************************************* */
   // Setup DMA0's start address register
   // Since the start address for DMA_1 is also the 'Holding Register', the starting
   // address for DMA_1 will be the same as the starting address for DMA_0.  If they
   // had different starting adddresses, then you would have to use the same routine
   // that was used for DMA_0.
   DMASADR1_0 =  PADDR_0;
   DMASADR1_1 =  PADDR_1;
   DMASADR1_2 =  PADDR_2;
   DMASADR1_3 =  PADDR_3;

   // Enable the DMA channel 1 and intialize the transfer
   DMACTRL1_0 |= 0x06;

   // Enable DMA_WRITE control register to access DMA Channel 1
   DMAwrite = 0x33;

   // Start the fun by performing a software DMA request on DMA Channel 0
   DMACTRL0_0 |= 0x10;

   // An embedded system program never ends
   while (1) {
       Update++;
   }
```

Using the BFM with the DMA

This section describes how to use the DMA functionality of the Bus Functional Model. For a thorough description of the BFM please refer to **Application Note AN32**. Code for the testbench.v file follows.

```
`timescale 1ns/10ps

/*************************************************************
 *
 *                  Triscend Corporation
 *
 *                   Fast Chip 2.0 Project
 *
 *************************************************************
 *
 *
 *     Module:       CSL RTL Testbench
 *
 *
 *     Description:
 *
 *       This is the testbench for interfacing your CSL design
 *       to the Bus Functional Model, for the purposes of testing
 *       it before needing to download it onto the hardware.
 *
 ************************************************************/

// (Optional) To globally initialize flip-flops and latches:
// 1. compile TriscendV.v with the TRDEF_GLOBAL flag defined
//    (+define+TRDEF_GLOBAL command line option can be used)
// 2. uncomment the `define below or compile with +define+TRDEF_GLOBAL
// 3. compile TriscendGlobal.v
// 4. specify top-level modules as: testbench, TRGLOBAL

`define TRDEF_GLOBAL
`define BUS_CLOCK_PERIOD 20
`define VERBOSE 1

module testbench ();
```

Setting Verbose to 1 will give more information during simulation which will help during the debugging process.

```
// CSIBUS Declaration
wire [127:0] CSIBUS;

// Sideband for interrupts
wire [127:0]  SIDEBAND;

// include the file with task declarations and logic
`include "saddressmap.v"
`include "bfm_tasks.v"

// my declarations
 wire [7:0] led1;
 wire [7:0] led2;
 reg [7:0] cnt;
 integer adr0, adr1;

// Your csl model is instantiated here
// `include "csl.v"
 DMAwires csl (.CSIBUS ( CSIBUS ), .\DipSwitch. (8'h55), .\D1. (led1), .\D2. (led2));
```

Simulates the input from the DIP switch. In this example we've used the value 0x55 (01100110) as the input.

Normally the csl model is instantiated in a file called **csl.v**. For this example we've instantiated the csl model directly in **testbench.v**.

```
// put task calls and testbench logic be
initial
  begin
    @(posedge bfm_brstN);
    bfm_idle(5);

    // reset and set up DMA channels 0 and 1
     dma_reset (0);
     dma_reset (1);
     dma_setup (0, 0, 32'h0010_effd, 16'h0001, 2'b00);
     dma_setup (1, 32'h0010_effd, 0, 16'h0001, 2'b00);
```

Sets up the basic DMA transfer parameters – source address, destination address, transfer byte count, transaction size – for a specified channel. For **I/O → memory** transactions ignore the source address. For **memory → I/O** transactions ignore the destination address.

```
   // enable DMAread and DMAwrite.  DMAread access DMA channel 0
   // and DMAwrite accesses DMA channel1
     cpu_write8 (`DMAread, 8'h11);
     cpu_write8 (`DMAwrite, 8'h33);

   // find the base address for DMA0 and DMA1
     adr0 = bfm_dma_base_address(0);
     adr1 = bfm_dma_base_address(1);
     cnt=0;
     while (cnt<4)
     begin

     // initialize an I/O to memory DMA transfer
       dma_io2mem_init (0);
       cpu_write8 (adr0, 8'h56);  // issue a SW request

     // initialize a memory to I/O DMA transfer
       dma_mem2io_init (1);
       cpu_write8 (adr1, 8'h16);  // issue a SW request

     // wait for DMA interrupts
       dma_wait (0, 1);
       dma_wait (1, 1);

     // if everything works properly LED1 and LED2 should equal the DipSwitch
       $display ("LED1 = %h", led1);
       $display ("LED2 = %h", led2);

       cnt = cnt+1;

     end
      $finish;
    end

endmodule // testbench
```

Appendix B: Example Design Project – DMA Grab

Before Getting Started

Before starting this tutorial make sure that you have the following items. Please refer to the *Triscend E5 Development Board* for information on all the items included with the Triscend E5 Evaluation Kit

- The latest version of FastChip properly installed on your computer.
- Keil uVision2 properly installed on your computer.
- A Triscend E5 Evaluation Board with a download cable.

Scope

This example design project introduces you to the basic operations and the design flow of the Triscend FastChip development system. It guides you through a sample project using the E5's DMA controllers.

It will provide an introduction to hardware design with CSoC devices, specifically utilizing the DMA controller modules and some of their main features.

The audience for this example design project is an intermediate user. First time users may first want to review the FastChip online tutorial. The tutorial is available by opening FastChip, then making the menu selection **Help → Triscend CSoC Learning Center (Tutorial)**.

In this tutorial you will learn how to:

- Add IP modules from the FastChip library, including the DMA Selectors.
- Connect the signals between the IP modules.
- Set up CSoC dedicated resources.

Design Overview

This FastChip example project uses the DMA to transfer data from a counter to the LED displays. Information from a counter is displayed on an 7-segment LED. Using the DMAs the CSoC sends the same information to the second LED display. When everything works properly, the two LEDs will match.

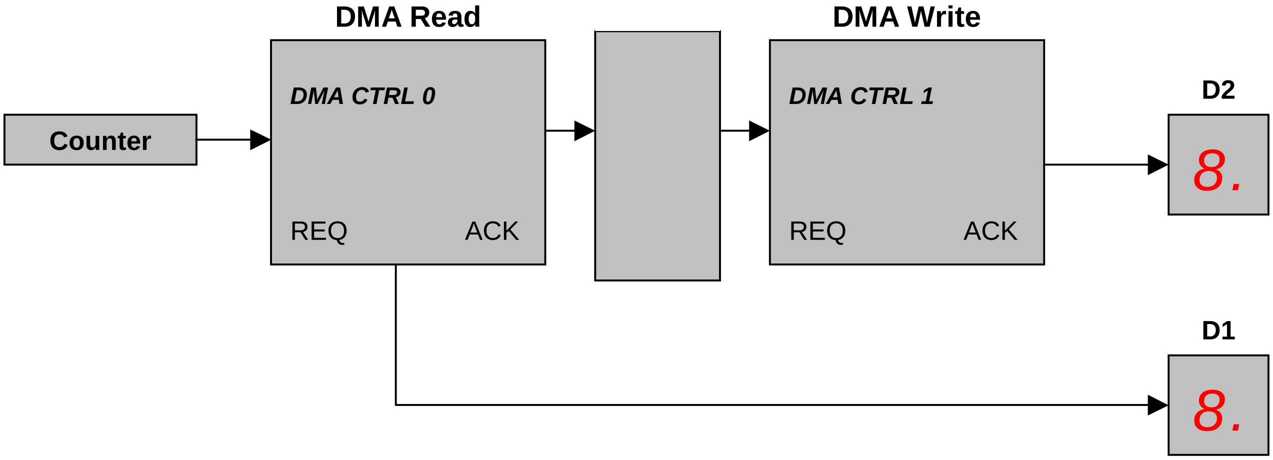


Figure B1: Block Diagram

Configuring the DMA Channels

This project makes use of both DMA channels, DMA_0 and DMA_1. FastChip's interface to the E5's dedicated resources is shown in Figure B2.

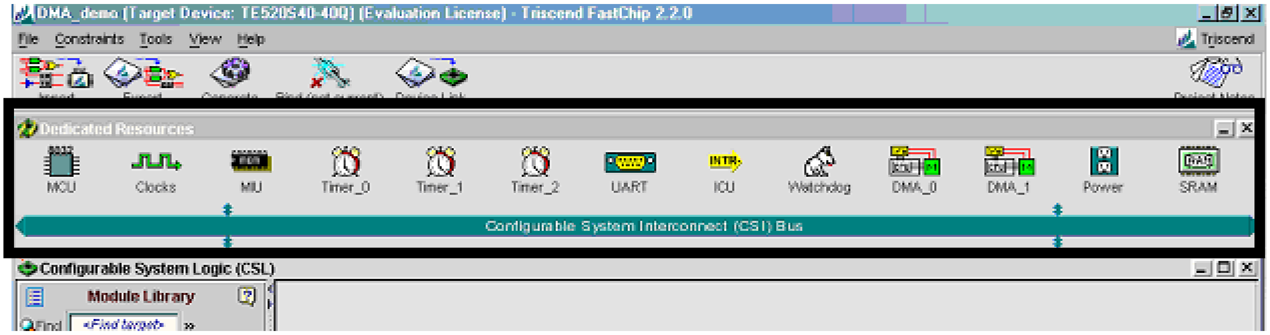

DMA_demo (Target Device: TE520S40-40Q) (Evaluation License) - Triscend FastChip 2.2.0
File
Constraints
Tools
View
Help
Triscend
Dedicated Resources
MCU
Clocks
MIU
Timer_0
Timer_1
Timer_2
UART
ICU
Watchdog
DMA_0
DMA_1
Power
SRAM
Configurable System Interconnect (CSI) Bus
Configurable System Logic (CSL)
Module Library
Find
<Find target>

Figure B2: Dedicated resources window in FastChip.

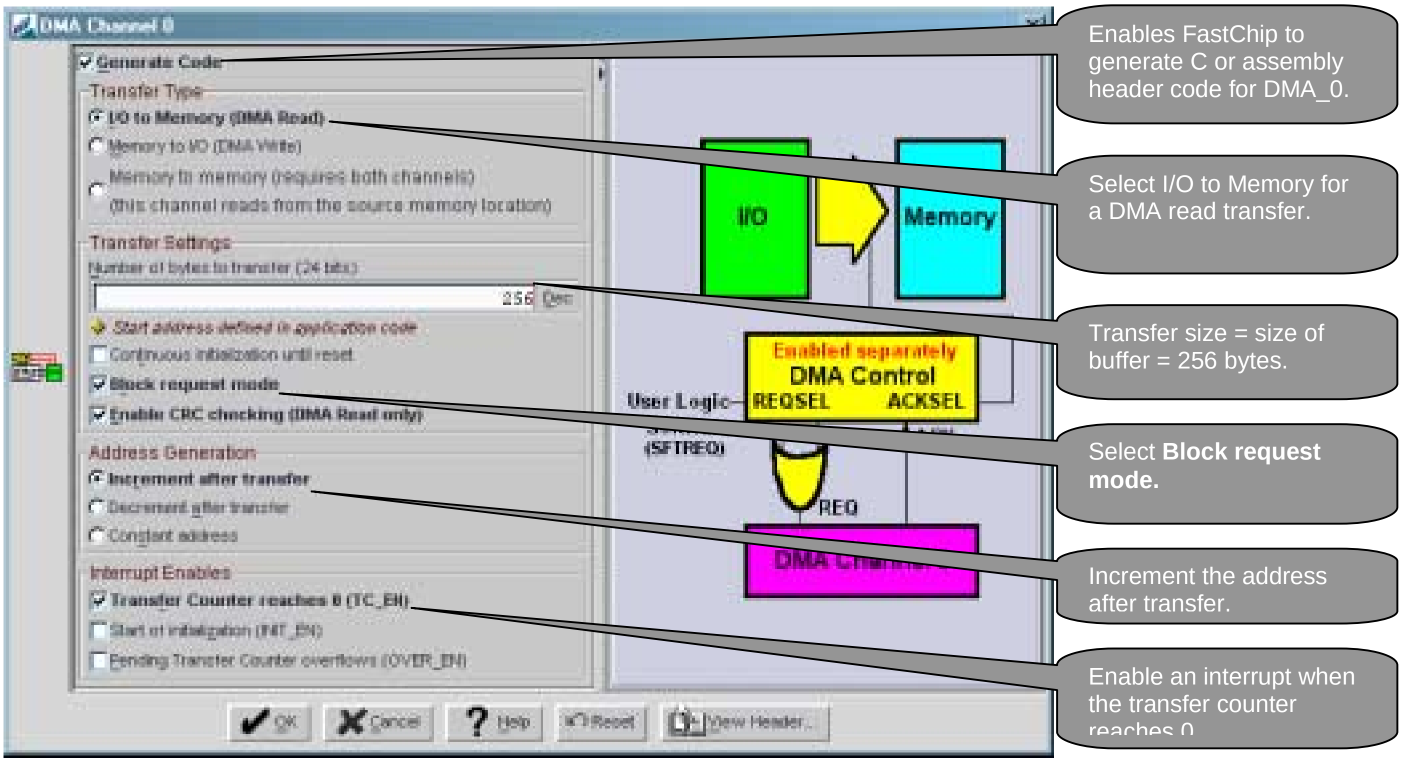
Configuring DMA_0

For this project, DMA_0 needs to be configured for **I/O to Memory** transfers. In addition, DMA_0 needs to be configured for **block request mode**. This option can be set in the **Transfer Settings** section (see Figure B3).

Since the source of the DMA transfer is a 256-byte buffer, we need to increment the starting address after transfer. This option can be set in the **Address Generation** section (see Figure B3).

Unlike the project described in Appendix A, this example project will make use of the Interrupt Enables. Specifically, we will enable an interrupt when the **Transfer Counter reaches 0**. This option can be set in the **Interrupt Enables** section (see Figure B3).

For this project, DMA_0 should be configured as shown in Figure B3.



DMA Channel 0

☑ Generate Code

Transfer Type
- ● I/O to Memory (DMA Read)
- ○ Memory to I/O (DMA Write)
- ○ Memory to memory (requires both channels) (this channel reads from the source memory location)

Transfer Settings

Number of bytes to transfer (24 bits): 256 Dec

Start address defined in application code

- □ Continuous initialization until reset
- ☑ Block request mode
- ☑ Enable CRC checking (DMA Read only)

Address Generation
- ● Increment after transfer
- ○ Decrement after transfer
- ○ Constant address

Interrupt Enables
- ☑ Transfer Counter reaches 0 (TC_EN)
- □ Start of initialization (INIT_EN)
- □ Pending Transfer Counter overflows (OVER_EN)

Enables FastChip to generate C or assembly header code for DMA_0.

Select I/O to Memory for a DMA read transfer.

Transfer size = size of buffer = 256 bytes.

Select **Block request mode.**

Increment the address after transfer.

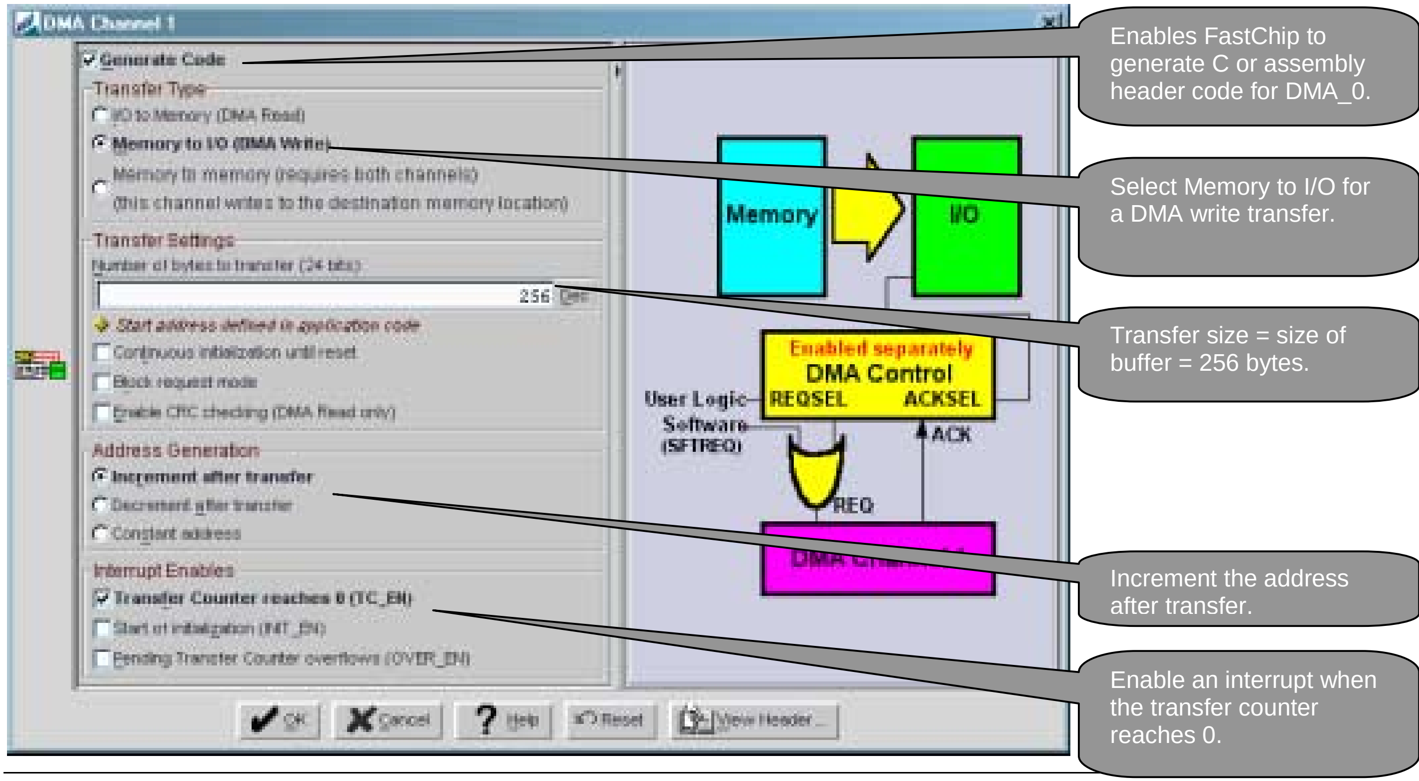
Enable an interrupt when the transfer counter reaches 0

Figure B3: Configuring DMA_0

Checking the **Generate Code** box initiates a header code generation session. All the DMA_0's registers will be set appropriately depending on the options selected in the configuration window. It is important to note that **the start address must be defined in the application code**.

Configuring DMA_1

DMA Channel 1 should be configured similar to DMA_0. DMA_1 needs to be configured for **Memory to I/O** transfers and must be set to single request mode (i.e. not block request mode). See Figure A4 to see how to configure DMA_1.



DMA Channel 1

☑ Generate Code

Transfer Type
○ I/O to Memory (DMA Read)
● Memory to I/O (DMA Write)
○ Memory to memory (requires both channels) (this channel writes to the destination memory location)

Transfer Settings
Number of bytes to transfer (24 bits)
256 Dec
Start address defined in application code
□ Continuous initialization until reset
□ Block request mode
□ Enable CRC checking (DMA Read only)

Address Generation
● Increment after transfer
○ Decrement after transfer
○ Constant address

Interrupt Enables
☑ Transfer Counter reaches 0 (TC_EN)
□ Start of initialization (INIT_EN)
□ Pending Transfer Counter overflows (OVER_EN)

Enables FastChip to generate C or assembly header code for DMA_0.

Select Memory to I/O for a DMA write transfer.

Transfer size = size of buffer = 256 bytes.

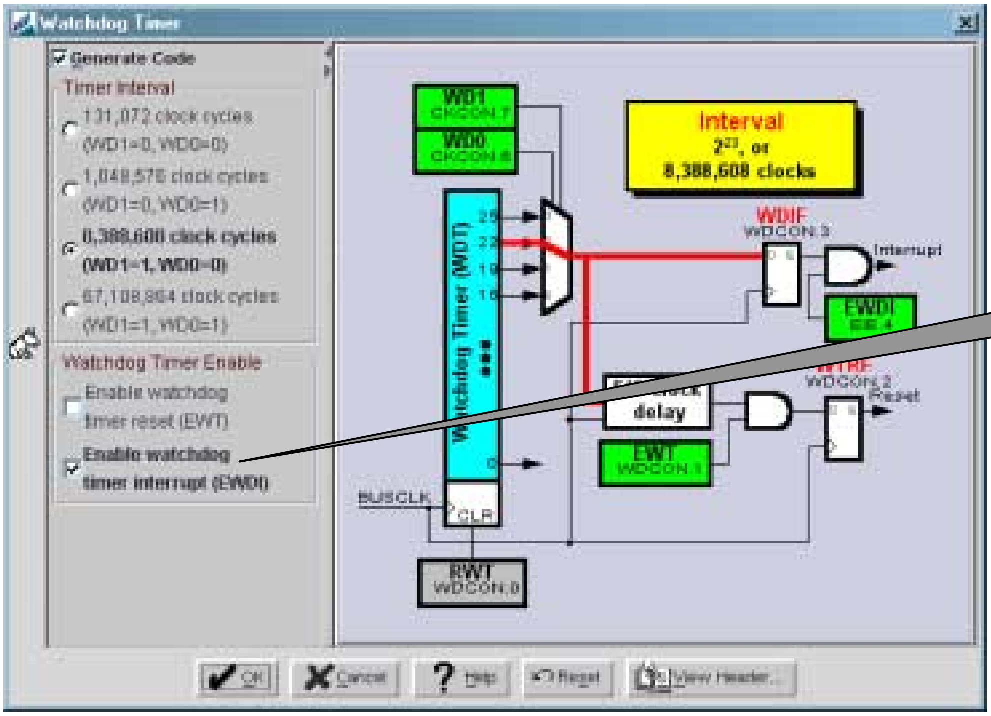
Increment the address after transfer.

Enable an interrupt when the transfer counter reaches 0.

Figure B4: Configuring DMA_1

Configuring the Watchdog Timer (WDT)

The WDT is used by the application code to initiate software requests for DMA_1. For this project, configure the WDT as shown in Figure B5. The WDT can be found in the Dedicated Resources section of FastChip as shown in Figure B2.



Watchdog Timer

☑ Generate Code

Timer Interval

○ 131,072 clock cycles (WD1=0, WD0=0)

○ 1,048,576 clock cycles (WD1=0, WD0=1)

● 8,388,608 clock cycles (WD1=1, WD0=0)

○ 67,108,864 clock cycles (WD1=1, WD0=1)

Watchdog Timer Enable

□ Enable watchdog timer reset (EWT)

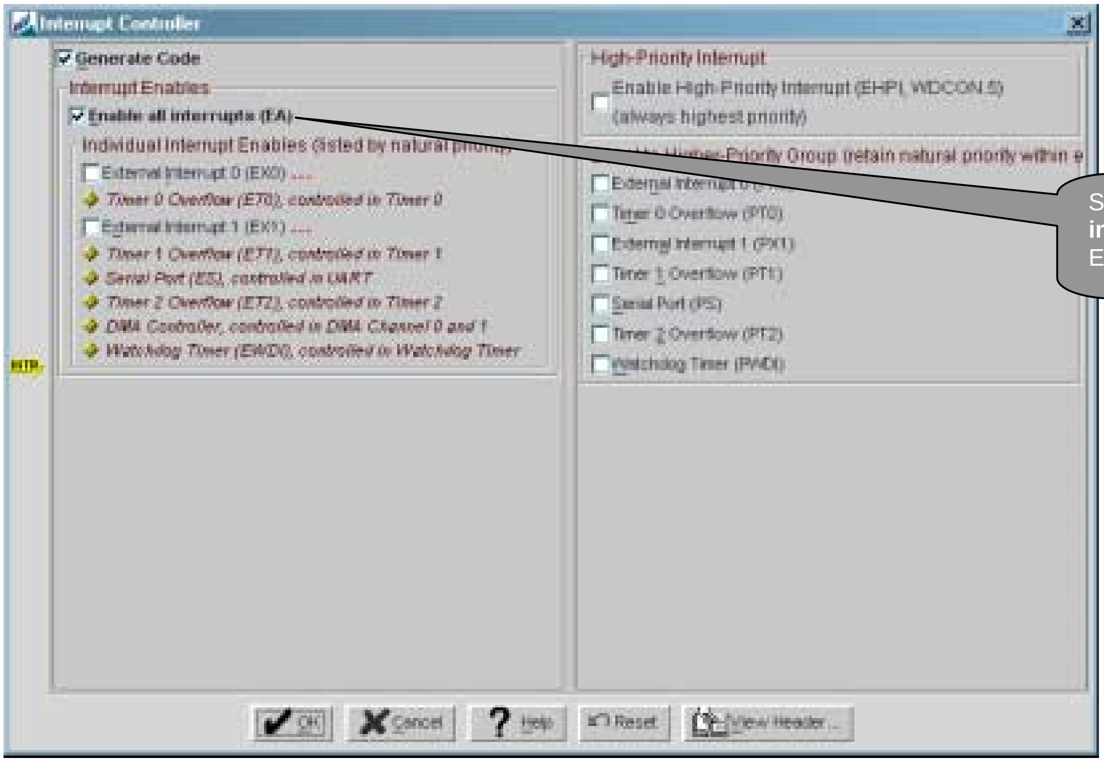
☑ Enable watchdog timer interrupt (EWDI)

Interval 2^{23}, or 8,388,608 clocks

Enable the watchdog timer interrupt.

Figure B5 Configuring the Watch Dog Timer (WDT)

In order for the watchdog timer interrupt to be recognized, all the E5's interrupt system must be enabled. This can be accomplished by configuring the **Interrupt Controller Unit (ICU)** in the Dedicated Resources window of FastChip (Figure B2).



Interrupt Controller

☑ Generate Code

Interrupt Enables

☑ Enable all interrupts (EA)

Individual Interrupt Enables (listed by natural priority)

□ External Interrupt 0 (EX0) ....

- *Timer 0 Overflow (ET0), controlled in Timer 0*

□ External Interrupt 1 (EX1) ....

- *Timer 1 Overflow (ET1), controlled in Timer 1*
- *Serial Port (ES), controlled in UART*
- *Timer 2 Overflow (ET2), controlled in Timer 2*
- *DMA Controller, controlled in DMA Channel 0 and 1*
- *Watchdog Timer (EWDI), controlled in Watchdog Timer*

High-Priority Interrupt

□ Enable High-Priority Interrupt (EHPI, WDCON.5) (always highest priority)

... Higher-Priority Group (retain natural priority within e

□ External Interrupt 0 ...

□ Timer 0 Overflow (PT0)

□ External Interrupt 1 (PX1)

□ Timer 1 Overflow (PT1)

□ Serial Port (PS)

□ Timer 2 Overflow (PT2)

□ Watchdog Timer (PWDI)

Selecting **Enable all interrupts** enables the E5 interrupt system.

Figure B6: Configuring the ICU

Selecting and Configuring IP Modules

For this project, five different modules are instantiated directly from the Triscend IP module library. The following modules will be used in this example project:

- DMA Selectors – used for interaction between the CSL and the DMA channels.
- Command Register – used for the holding register.
- Data Write – write data from the CSI bus' point of view.
- Data Read – read data from the CSI bus' point of view.
- Outputs – used to output data to the 7-segment displays.

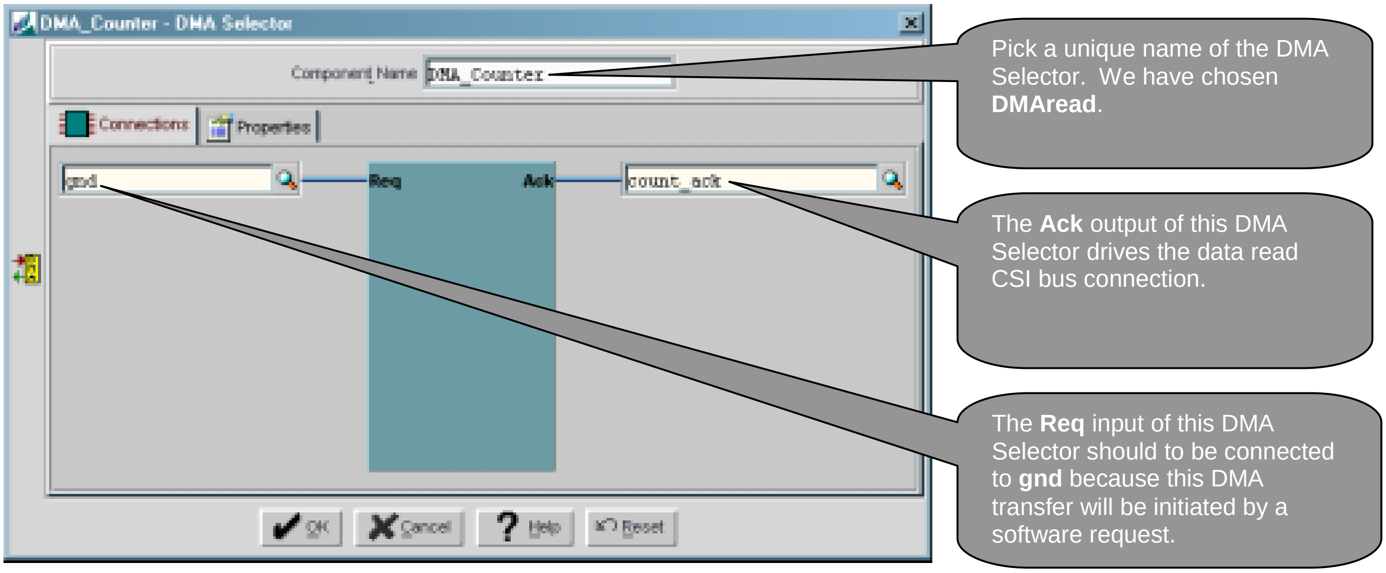
DMA Selectors

Two DMA Selectors are need for this project – one for DMA Channel 0 (DMA_0) and the other for DMA Channel 1 (DMA_1). The modules can be found in the module library tree under **CSI Bus → Selectors**.

Each IP module has a **Component Name** which can the designer can choose. **The Component Name does not affect the functionality of the design in any way. It is used only to label the modules.** We will name the two selectors **DMA_Counter** and **DMA_Write**.

DMA_Counter

Drag and drop the **DMA Selector** into the CSL window and click the module to access **Connections** and **Properties** dialog boxes. Figure B7 illustrates the **Connections** dialog box for a **DMA Selector**.



DMA_Counter - DMA Selector

Component Name: DMA_Counter

gnd — Req

Ack — count_ack

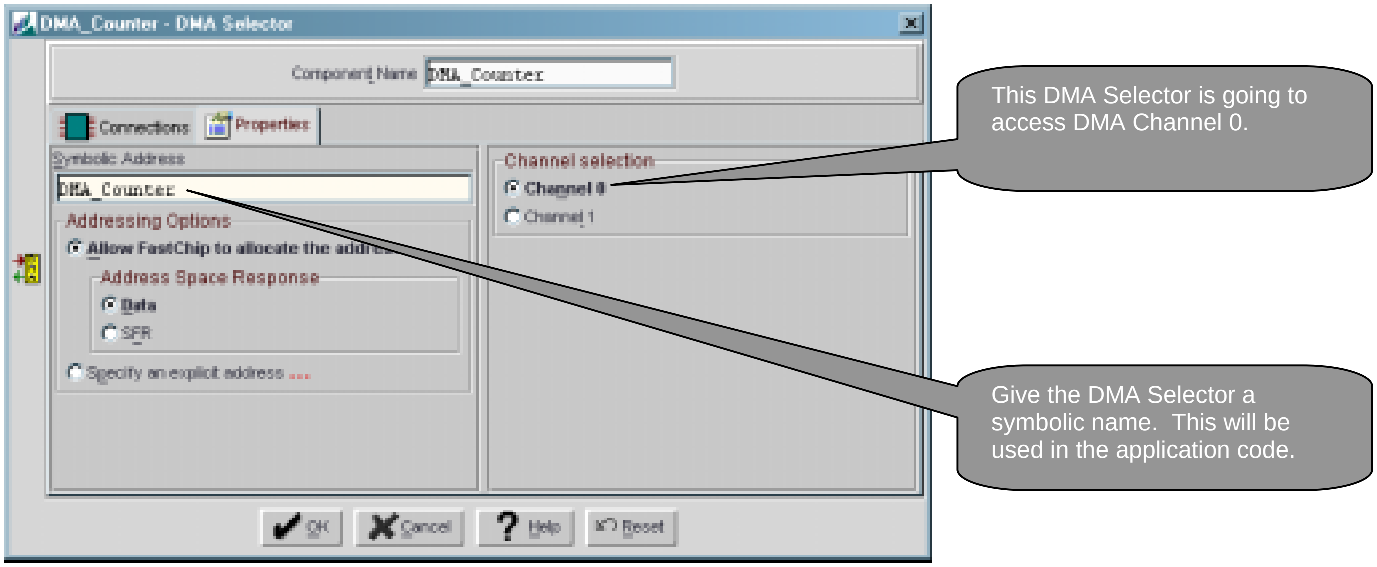
Pick a unique name of the DMA Selector. We have chosen **DMAread**.

The **Ack** output of this DMA Selector drives the data read CSI bus connection.

The **Req** input of this DMA Selector should to be connected to **gnd** because this DMA transfer will be initiated by a software request.

Figure B7: Configuring the connections of a DMA Selector for DMA_0

After configuring the connections as shown in Figure B7 click on the **Properties** tab to configure the DMA Selector's properties. Figure B8 illustrates the **Properties** dialog box for a **DMA Selector**.


DMA_Counter - DMA Selector

Component Name: DMA_Counter

Symbolic Address: DMA_Counter

Addressing Options
● Allow FastChip to allocate the addre...
Address Space Response
● Data
○ SFR
○ Specify an explicit address ...

Channel selection
● Channel 0
○ Channel 1

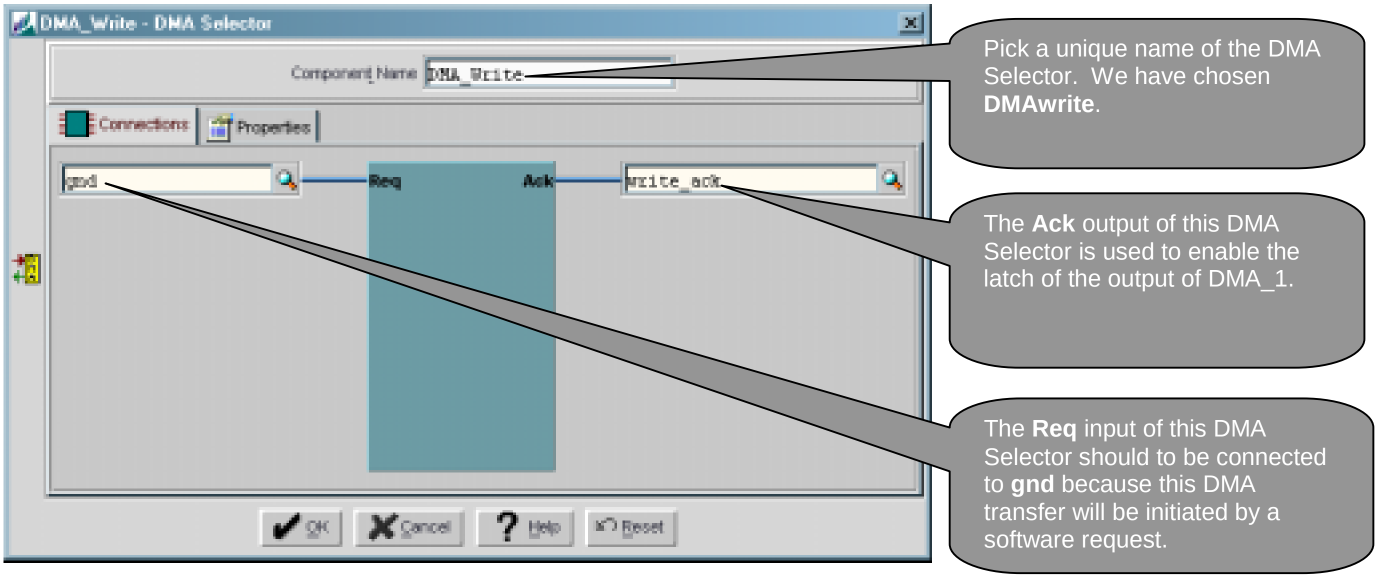
This DMA Selector is going to access DMA Channel 0.

Give the DMA Selector a symbolic name. This will be used in the application code.

Figure B8: Configuring the properties of a DMA Selector for DMA_0

DMA_Write

Drag and drop another **DMA Selector** into the CSL window. As before, click the module to access **Connections** and **Properties** dialog boxes. Figure B9 illustrates the **Connections** dialog box for a **DMA Selector**.


DMA_Write - DMA Selector

Component Name: DMA_Write

gnd — Req

Ack — write_ack

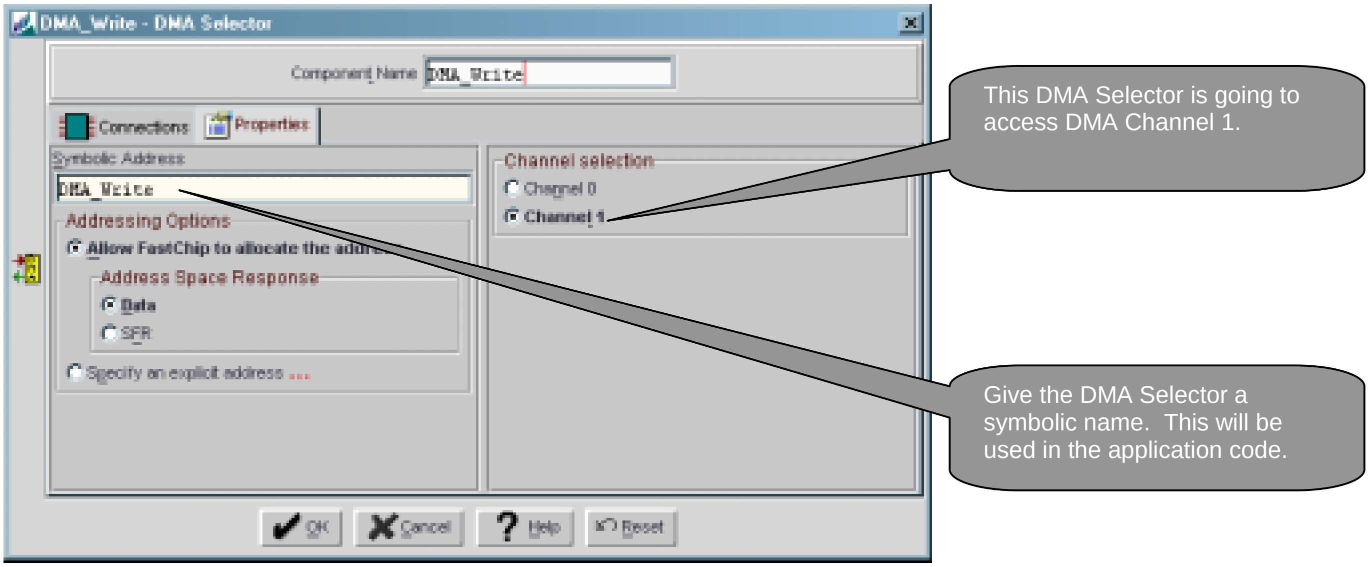
Pick a unique name of the DMA Selector. We have chosen **DMAwrite**.

The **Ack** output of this DMA Selector is used to enable the latch of the output of DMA_1.

The **Req** input of this DMA Selector should be connected to **gnd** because this DMA transfer will be initiated by a software request.

Figure B9: Configuring the connections of a DMA Selector for DMA_1

After configuring the connections as shown in Figure B9 click on the **Properties** tab to configure the DMA Selector’s properties. Figure B10 illustrates the **Properties** dialog box for a **DMA Selector**.



DMA_Write - DMA Selector

Component Name: DMA_Write

Symbolic Address: DMA_Write

Addressing Options

● Allow FastChip to allocate the addr

Address Space Response

● Data

○ SFR

○ Specify an explicit address ...

Channel selection

○ Channel 0

● Channel 1

This DMA Selector is going to access DMA Channel 1.

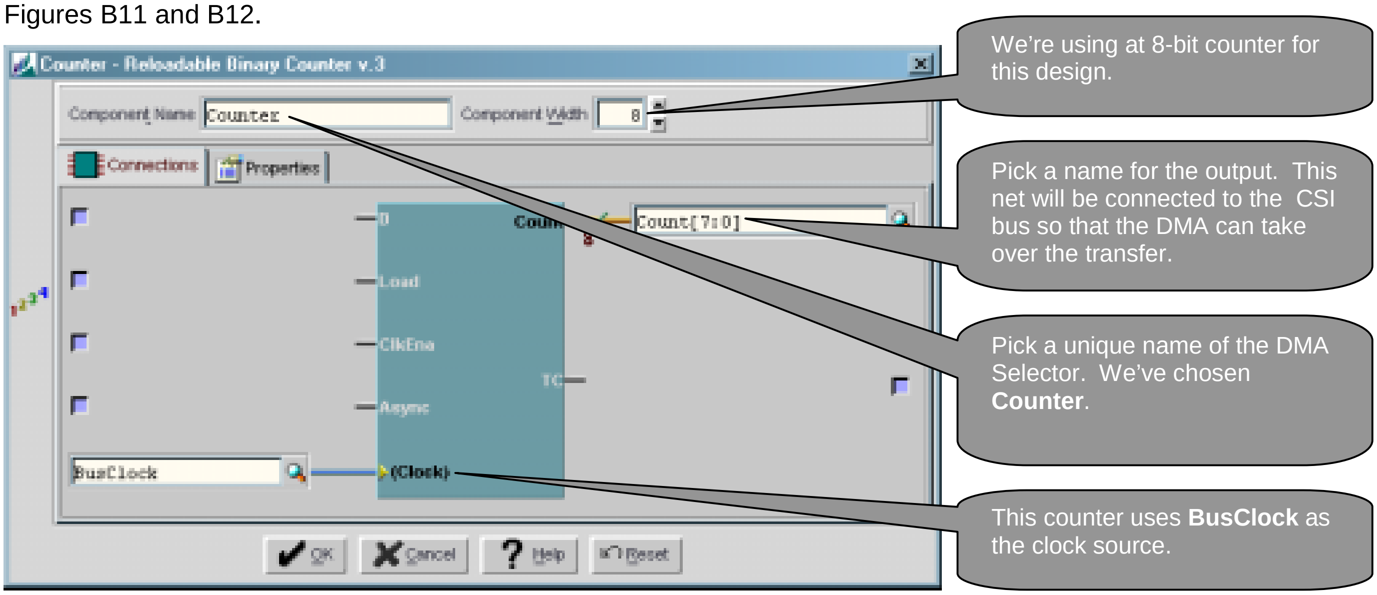
Give the DMA Selector a symbolic name. This will be used in the application code.

Figure B10: Configuring the properties of a DMA Selector for DMA_1

The DMA portion of this example project has been configured.

Counter

A counter will serve as the data source for this example project. Configure the counter as shown in Figures B11 and B12.



Counter - Reloadable Binary Counter v.3

Component Name: Counter

Component Width: 8

BusClock

Count[7:0]

We're using at 8-bit counter for this design.

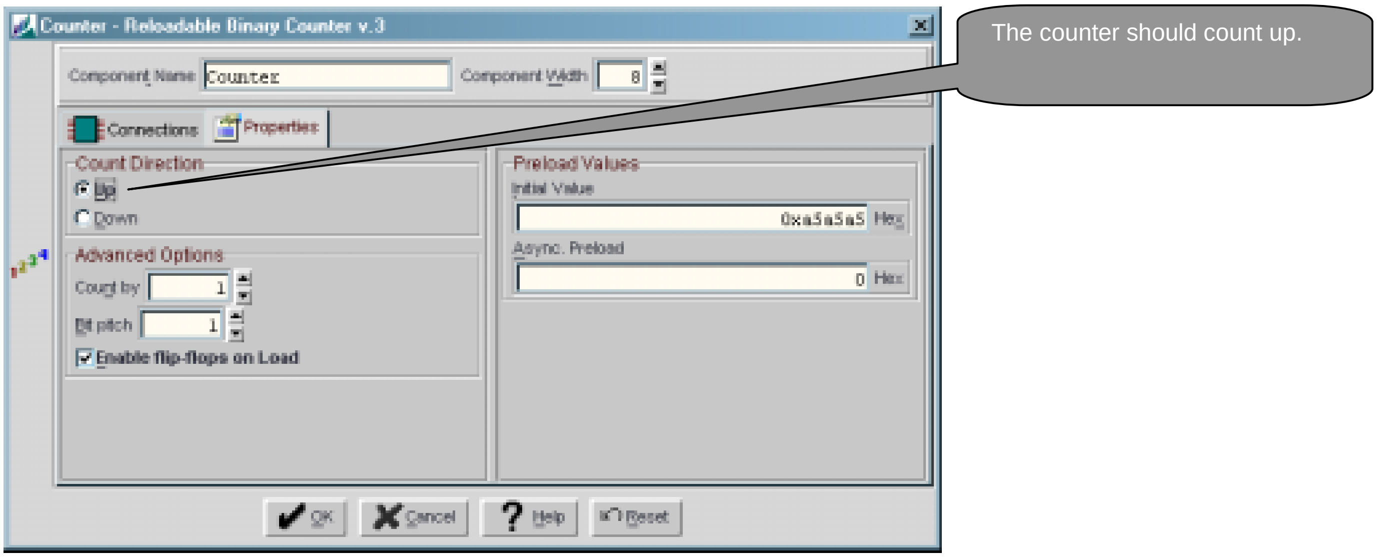
Pick a name for the output. This net will be connected to the CSI bus so that the DMA can take over the transfer.

Pick a unique name of the DMA Selector. We've chosen **Counter**.

This counter uses **BusClock** as the clock source.

Figure B11: Configuring the Counter module

After configuring the connections of the counter, we will need to set the properties as shown in Figure B12.



Counter - Reloadable Binary Counter v.3

Component Name: Counter
Component Width: 8

Count Direction
● Up
○ Down

Advanced Options
Count by: 1
Bit pitch: 1
☑ Enable flip-flops on Load

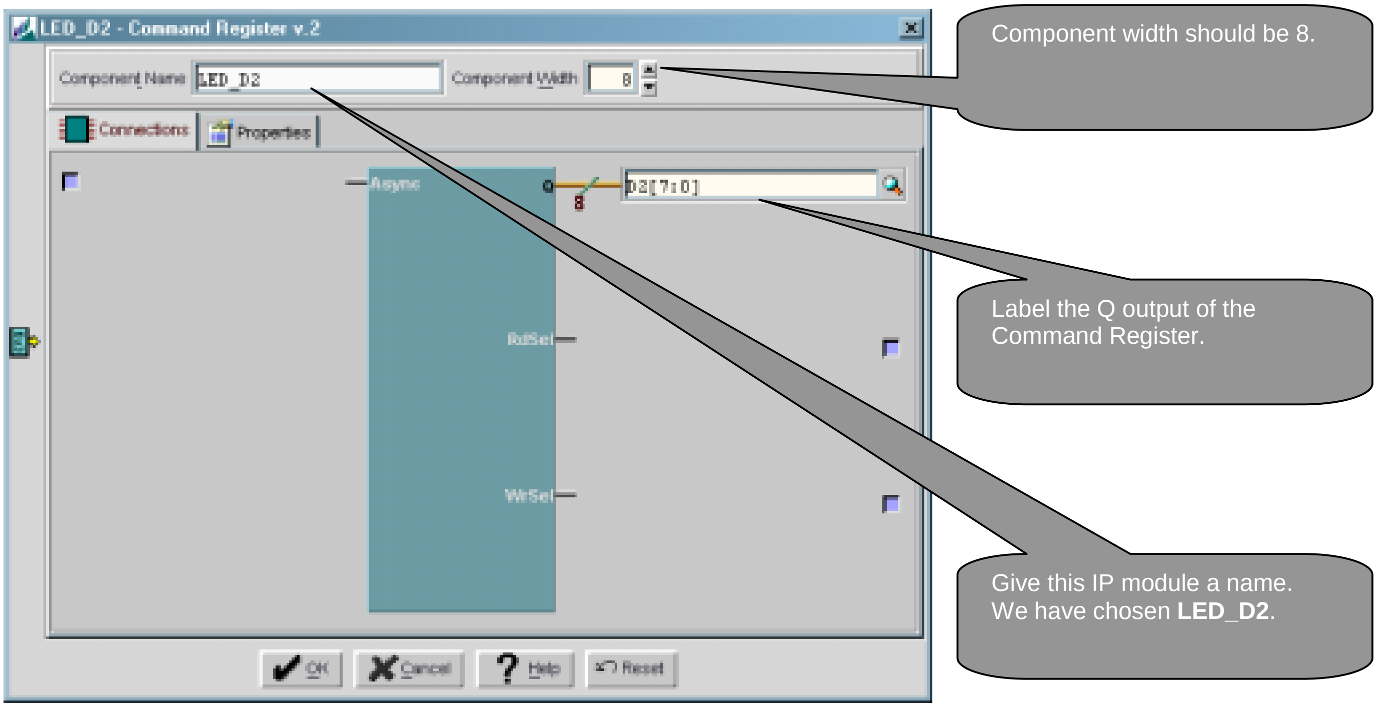
Preload Values
Initial Value: 0xa5a5a5 Hex
Async. Preload: 0 Hex

The counter should count up.

Figure B12: Setting the Properties of the Counter module

Command Register

We will instantiate a **Command Register** to act as a memory-mapped holding register verifying the DMA transfers. The Command Register can be found in the module library tree under **Peripherals → Control.** As with the DMA Selectors, drag and drop a Command Register into the CSL window and configure the Command Register as shown in Figures B13 and B14.



LED_D2 - Command Register v.2

Component Name: LED_D2
Component Width: 8

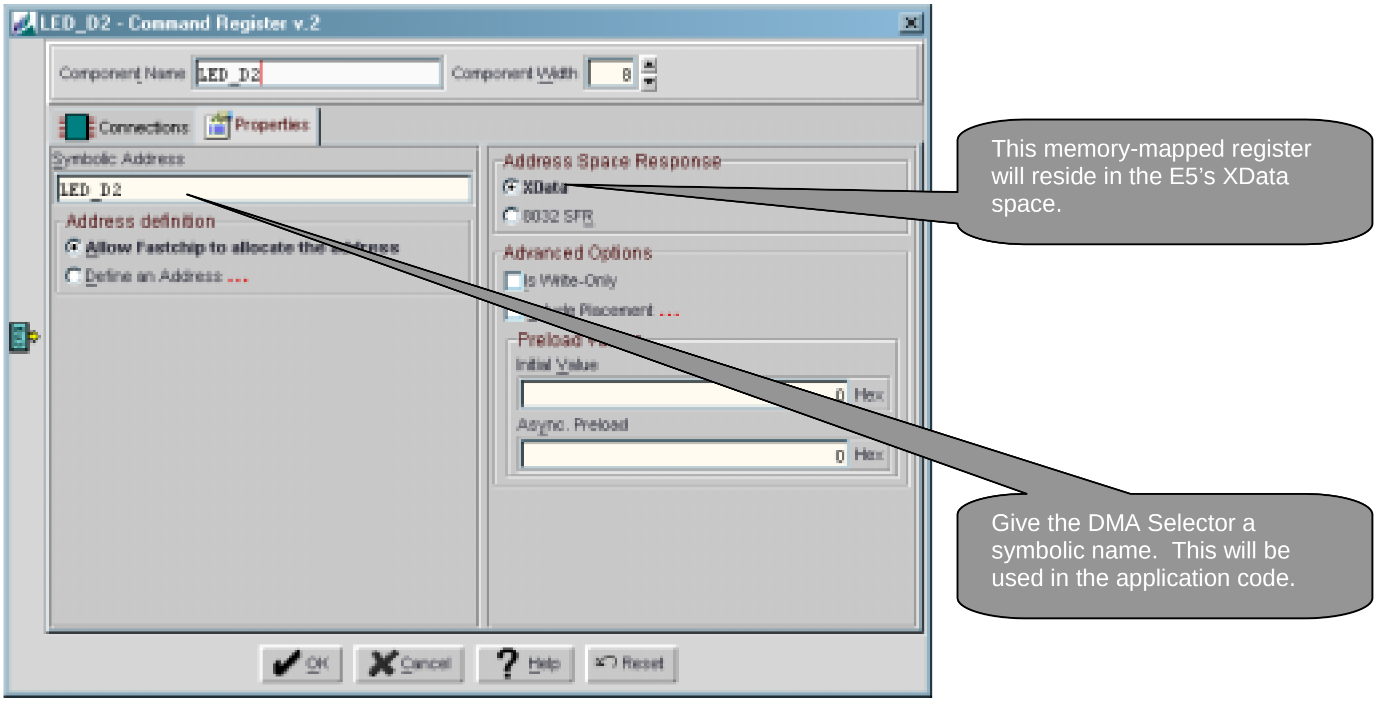
Q: D2[7:0]

Component width should be 8.

Label the Q output of the Command Register.

Give this IP module a name. We have chosen **LED_D2**.

Figure B13: Configuring the connections of a Command Register



Component Name: LED_D2
Component Width: 8

Symbolic Address: LED_D2

Address definition
● Allow Fastchip to allocate the address
○ Define an Address ...

Address Space Response
● XData
○ 8032 SFR

Advanced Options
□ Is Write-Only
□ Absolute Placement ...

Preload Values
Initial Value: 0 Hex
Async. Preload: 0 Hex

This memory-mapped register will reside in the E5's XData space.

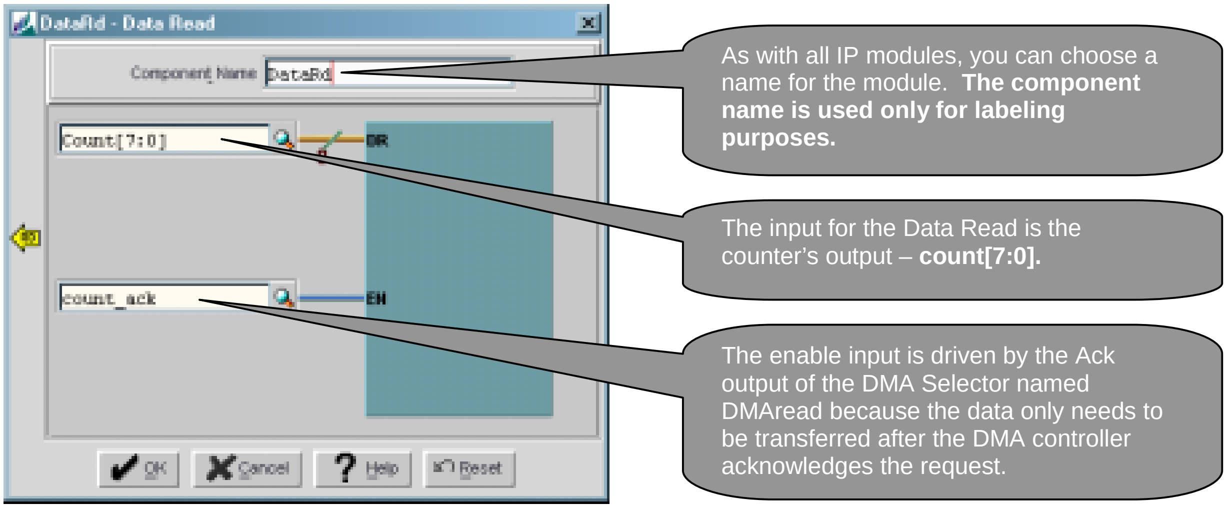
Give the DMA Selector a symbolic name. This will be used in the application code.

Figure B14: Configuring the properties of a Command Register

CSI Bus Connections

Data Read

Since we are getting data from the counter, we need a way to place the data on the CSI bus. The **Data Read** IP module is used exactly for this purpose. It can be found in the module library tree under **CSI Bus → Data Bus.** Figure B15 illustrates the Data Read IP module.



Component Name: DataRd

Count[7:0]

count_ack

As with all IP modules, you can choose a name for the module. **The component name is used only for labeling purposes.**

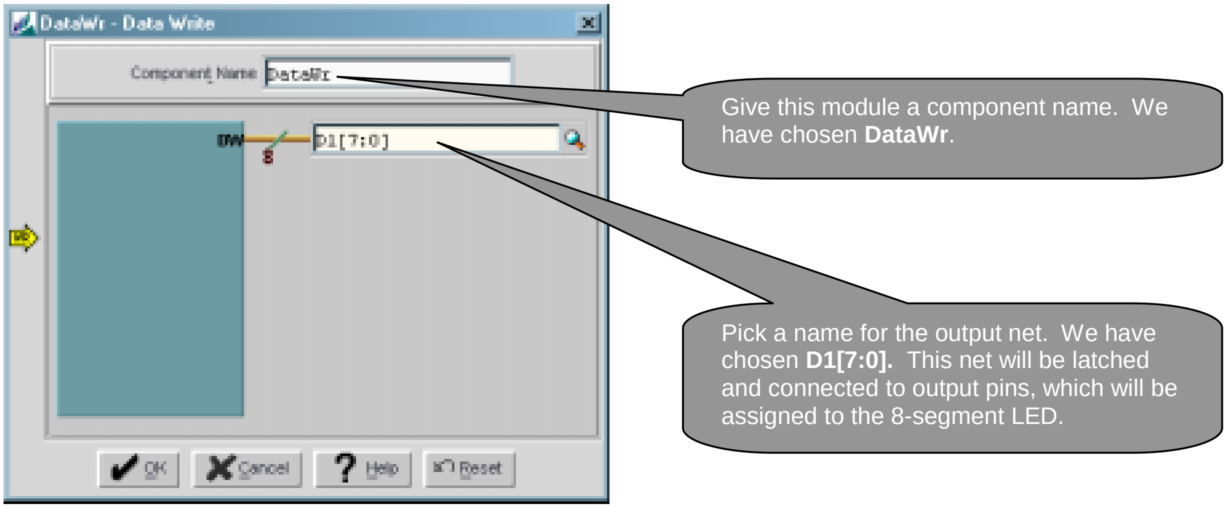
The input for the Data Read is the counter's output – **count[7:0].**

The enable input is driven by the Ack output of the DMA Selector named DMAread because the data only needs to be transferred after the DMA controller acknowledges the request.

Figure B15: Data Read

Data Write

The DMA controller (**DMAwrite**) will automatically place the transferred data onto the CSI bus. However, we need a way to get the data from the CSI bus and send it to the 8-segment LED. The **Data Write** module, the opposite of a Data Read module**,** is used for this purpose. Figure B16 shows the Data Write module.



DataWr - Data Write

Component Name: DataWr

DW — 8 — D1[7:0]

Give this module a component name. We have chosen **DataWr**.

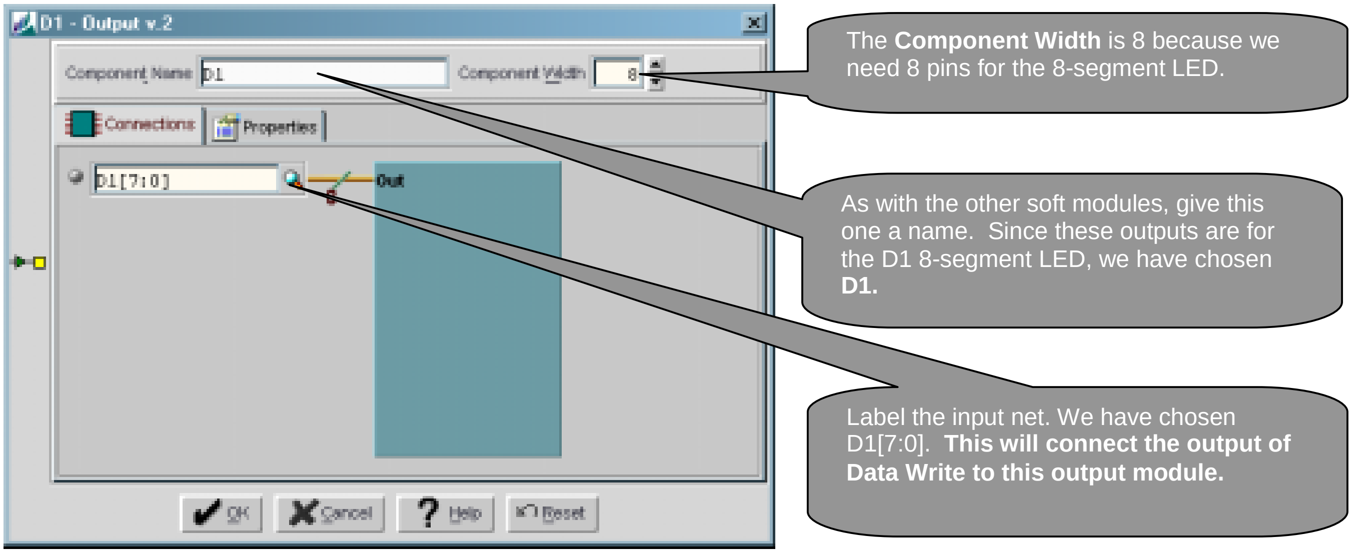
Pick a name for the output net. We have chosen **D1[7:0].** This net will be latched and connected to output pins, which will be assigned to the 8-segment LED.

Figure B16: Data Write

Outputs

This project requires 16 output pins to connect to the 2 8-segment LEDS. Drag and drop **two output IP modules** into the CSL and configure them as shown in Figures B17 – B19. The IP module can be found in the module library tree under **CSI Bus → I/O**.

Output for D1



D1 - Output v.2

Component Name: D1

Component Width: 8

D1[7:0] — 8 — Out

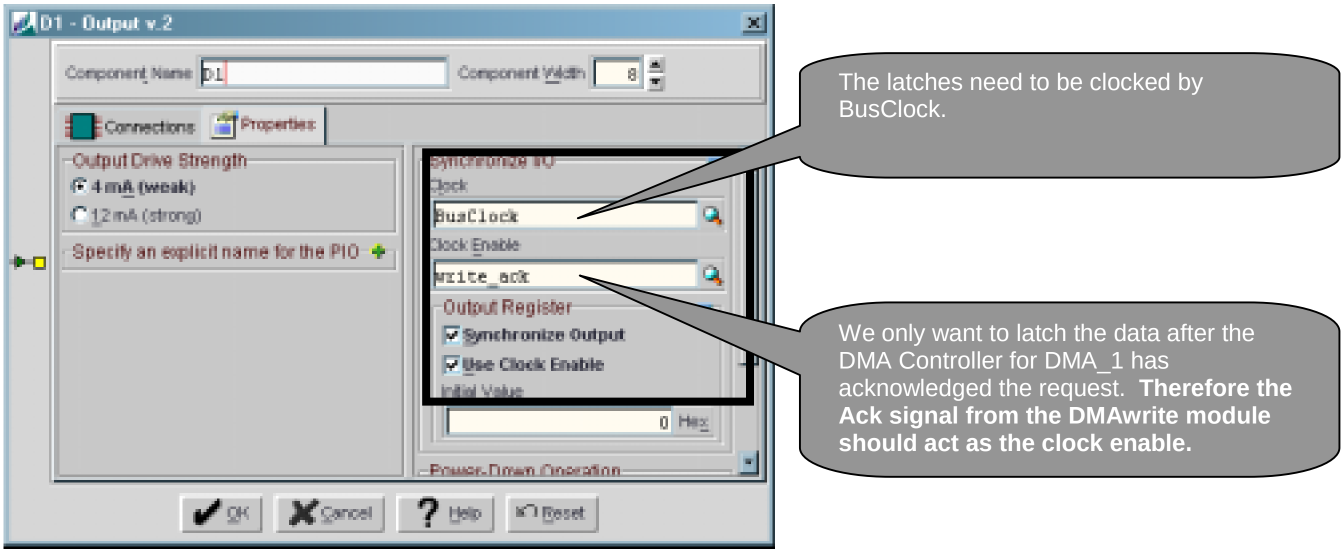
The **Component Width** is 8 because we need 8 pins for the 8-segment LED.

As with the other soft modules, give this one a name. Since these outputs are for the D1 8-segment LED, we have chosen **D1.**

Label the input net. We have chosen D1[7:0]. **This will connect the output of Data Write to this output module.**

Figure B17: Configuring the connections of an output for D1

For this example project, we would like to latch the data into registers. Please refer to Figure B18 to learn how to configure the module.



Component Name: D1

Component Width: 8

Output Drive Strength
● 4 mA (weak)
○ 12 mA (strong)

Specify an explicit name for the PIO

Synchronize I/O

Clock: BusClock

Clock Enable: write_ack

Output Register
☑ Synchronize Output
☑ Use Clock Enable

Initial Value: 0 Hex

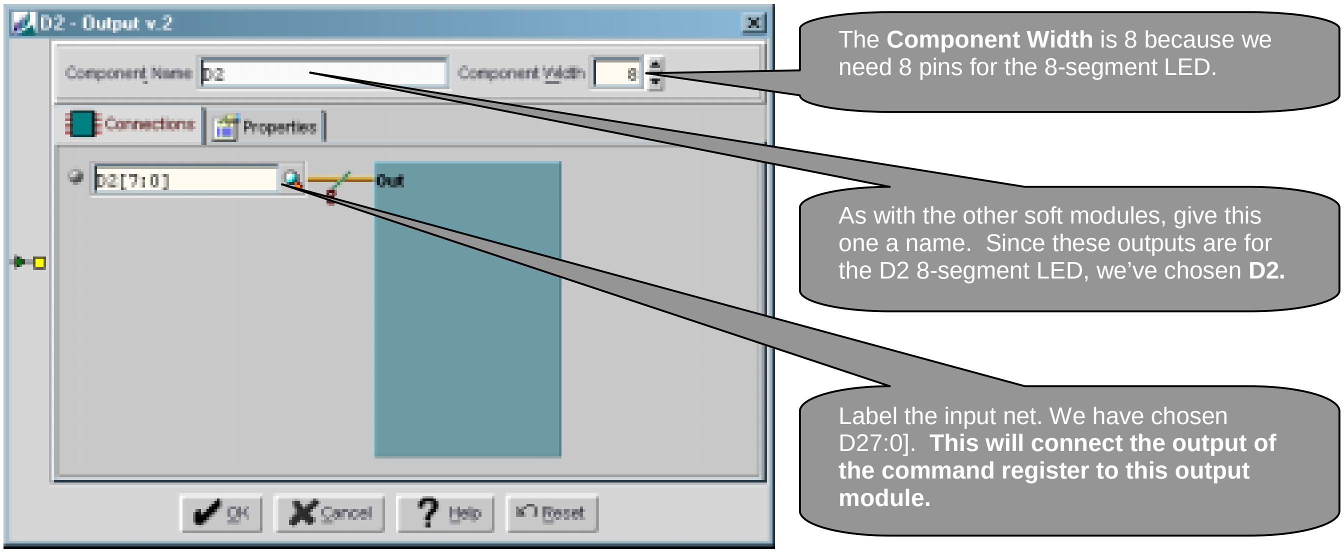
Power-Down Operation

The latches need to be clocked by BusClock.

We only want to latch the data after the DMA Controller for DMA_1 has acknowledged the request. **Therefore the Ack signal from the DMAwrite module should act as the clock enable.**

Figure B18: Configuring the properties for D1

Output for D2



Component Name: D2

Component Width: 8

D2[7:0]

The **Component Width** is 8 because we need 8 pins for the 8-segment LED.

As with the other soft modules, give this one a name. Since these outputs are for the D2 8-segment LED, we've chosen **D2.**

Label the input net. We have chosen D27:0]. **This will connect the output of the command register to this output module.**

Figure A15: Configuring the connections of an output for D2

We do not need to modify the properties of D2 because no special features are need (such as latching data).

To complete this example project follow the steps detailed in Appendix A **Configure I/O Constraints and Memory Interface Unit (MIU).**

Application Code

The following is the application code for this example project.

```
// Header file generated by FastChip
#include "dmagrab.h"

// Header file for routines that convert an 8032 logical
// address to a 32-bit physical address
#include "dmap.h"

/*=========================================================================
| CONSTANTS
 ========================================================================*/
// The size of the buffer created in XDATA RAM is
// configurable.  Set BUFFERSIZE as desired.
#define BUFFERSIZE 256

/*=========================================================================
| DECLARE VARIABLES
 ========================================================================*/
// Create a buffer in XDATA RAM and a pointer to it
unsigned char xdata MyBuffer[BUFFERSIZE];
unsigned char xdata * xdata Buffer_Ptr;

// The 'captured_flag', when non-zero, indicates that
// valid data is captured in MyBuffer.  0 = No data captured in buffer
// 0xff = Data captured
unsigned char data captured_flag = 0;

/*=========================================================================
| ROUTINES
 ========================================================================*/

/*********************************************************
 ** MAIN ROUTINE                                        **
 *********************************************************/
void main(void) {

   // Execute the initialization file created by FastChip.
   // This sets up any dedicated resources, including the
   // the DMA controller.
   DMAgrab_INIT();

   // DMA 0 is configured for block request mode, incrementing
   // addresses, no auto-initialize, CRC checking, 256 byte transfer.

   // Convert the 8032's 16-bit logic address pointing to the XDATA
   // RAM buffer into a 32-bit physical address required by the
   // DMA controller.  Using this method allows this code to
   // operate regardless of the initialization method used.
   //
   // Put logical address in special variables LADDR_1 and LADDR0,
   // as shown below.
   LADDR_1    =  (unsigned char)((unsigned short)&MyBuffer >> 8);
   LADDR_0    =  (unsigned char)((unsigned short)&MyBuffer & 0xff);

   // Then, call the special routine that converts the 16-bit logical
   // address to a 32-bit physical address.  The 32-bit address is
   // provided in variables PADDR_3 down to PADDR_0.
   xdata_logical_to_physical();

   // Setup DMA0's start address register
   DMASADR0_0 =  PADDR_0;
   DMASADR0_1 =  PADDR_1;
   DMASADR0_2 =  PADDR_2;
   DMASADR0_3 =  PADDR_3;

   // Load DMA0's length count.  Note, the value loaded is the buffer
   // size minus 1.
   DMASCNT0_0 = ((BUFFERSIZE-1) & 0xff);
   DMASCNT0_1 = (((BUFFERSIZE-1)>>8) & 0xff);
   DMASCNT0_2 = (((BUFFERSIZE-1)>>16) & 0xff);

   // Enable the DMA channel 0 and initialize the transfer
   DMACTRL0_0 |= 0x06;

   // Enable 'DMAcounter' control register to access DMA channel 0
   // NOTE:  Upper and lower nibbles MUST be identical.
   DMA_Counter = 0x11;
```

```
   // DMA_1 is configured for single transfer request mode,
   // incrementing addresses, no auto-initialize, 265 byte
   // transfer.

   // Setup DMA_1's start address register to the same location used
   // by DMA_0, as shown earlier.
   DMASADR1_0 =  PADDR_0;
   DMASADR1_1 =  PADDR_1;
   DMASADR1_2 =  PADDR_2;
   DMASADR1_3 =  PADDR_3;

   // Load DMA_1's length count
   DMASCNT1_0 = ((BUFFERSIZE-1) & 0xff);
   DMASCNT1_1 = (((BUFFERSIZE-1)>>8) & 0xff);
   DMASCNT1_2 = (((BUFFERSIZE-1)>>16) & 0xff);

   // Enable the DMA channel 1 and initialize the transfer
   DMACTRL1_0 |= 0x06;

   // Enable DMAwrite control register to access DMA Channel 1
   DMA_Write = 0x33;

   // Start the fun by performing a software DMA request on DMA Channel 0
   DMACTRL0_0 |= 0x10;

   // Set Buffer_Ptr to the address of the first location in the XDATA RAM
   // buffer.  This is used during verification.  If everything
   // operates correctly, then LED D2 equals LED D1 on the Triscend E5
   // Development Board.
   Buffer_Ptr = &MyBuffer;

   while (1) {
       // An embedded program never ends

       // The 8032 MCU is fully functional while the DMA
       //   controller is operating.  Add your own function
       //   here.
   }
}
```

```
/*********************************************************
 ** WATCHDOG INTERRUPT SERVICE ROUTINE **
 *********************************************************/

void WDT_ISR(void) interrupt 12 {

   if (captured_flag) {
       // Display current value from buffer located in
       // internal XDATA RAM, for verification purposes.
       LED_D2 = *Buffer_Ptr++;

       // Transfer one byte from XDATA RAM using DMA_1.
       // Set a software DMA request.
       DMACTRL1_0 |= 0x10;
   }

   // Clear watchdog timer interrupt flag (WDIF).  This
   // is a protected bit.  Open the Timed Acces (TA)
   // register by writing 0xAA followed by 0x55.
   TA = 0xAA;
   TA = 0x55;
   WDIF = 0;
}

/*********************************************************
 ** DMA CONTROLLER INTERRUPT SERVICE ROUTINE **
 *********************************************************/

void DMA_ISR(void) interrupt 7 {

   /* If DMA 0 is finished capturing data, set the
      'capture_flag' and re-initialize DMA 1.  Clear
      the DMA 0 terminal count interrupt when done.

      If DMA 1 is finished displaying data, clear the
      'capture_flag', re-initialize DMA 0, then set
      a software DMA request to capture more data using
      DMA 0.  Clear the DMA 1 terminal count interrupt
      when done.   */
   // DMA 1 Complete:  Check TC bit in DMA1 status register
   if (DMAINT1 & 0x01) {

       // Clear captured flag
       captured_flag = 0;

       // Re-initialize DMA 0 transfer and set software request
       DMACTRL0_0 |= 0x14;

       // Clear TC bit by writing a '1' to the bit location
       DMAINT1 |= 0x01;
   }

   // DMA 0 Complete:  Check TC bit in DMA0 status register
   if (DMAINT0 & 0x01) {

       // Reset Buffer_Ptr, used for verification purposes
       Buffer_Ptr = &MyBuffer;

       // Re-initialize DMA 1 transfer
       DMACTRL1_0 |= 0x04;

       // Data is captured in MyBuffer, set 'captured_flag'
       captured_flag = 0xff;

       // Clear TC bit by writing a '1' to the bit location
       DMAINT0 |= 0x01;
   }
}
```

Revision History

Revision	Date	Comment
1.0	25-JUN-2001	Initial release

Triscend

Triscend Corporation
301 North Whisman Road
Mountain View, CA 94043-3969
USA

Tel: 1-650-968-8668
Fax: 1-650-934-9393

Support: **SupportCenter@triscend.com**
Web: **www.triscend.com/salessupport**