```
19        { 0x25, 0x66 }, // "4"
20        { 0x2E, 0x6D }, // "5"
21        { 0x36, 0x7D }, // "6"
22        { 0x3D, 0x07 }, // "7"
23        { 0x3E, 0x7F }, // "8"
24        { 0x46, 0x6F }, // "9"
25        { 0x45, 0x3F }  // "0"
26    };
27
28    static void displayPs2Data() interrupt 0 using 0
29    {
30        unsigned int i;
31        unsigned char c;
32
33        c = rcvData;   // get the keyboard scan code
34
35        // search the translation table for the scan code
36        for(i=0; i<sizeof(ps2XlateTbl)/sizeof(ps2XlateEntry); i++)
37            if(ps2XlateTbl[i].ps2Data == c)
38            { // found a matching scan code in the table
39                ledPort = ps2XlateTbl[i].led; // display digit
40                return;
41            }
42
43        // no matching scan code was found, so display "E"
44        ledPort = ERROR;
45    }
```

Once you set the compiler and linker options as you did in the previous chapter, you can compile and link the ***Chap31*** Keil project. Then re-enter the FastChip project window and bind your design. Download the keyboard interface circuitry and the 8032 program in the Chap31.HEX file to your CSoC Board. Finally, use dScope to establish a debugging link to the CSoC Board and then reset and execute the application program. At this point, you should be able to type on the numeric keys of a keyboard attached to the PS/2 port of your CSoC Board and see the numbers appear on the LED digit.

Design 3.2 - PS/2 Keyboard Interface Using DMA

In the previous section you built a keyboard interface that interrupts the 8032 MCU program flow whenever a key is pressed. Keystrokes don't arrive at a very high rate so the MCU isn't overly burdened by processing the interrupts, but this isn't true for all data sources. For example, the UART could receive bursts of data and interrupt the 8032 thousands of times per second. Then the UART is idle until another burst arrives. Programming the 8032 to handle the rapid bursts might be impossible, so it is better to buffer the bursts and then let the MCU process the buffer contents at regular intervals.

You could use a FIFO for this, but a FIFO that buffers more than 1000 bytes won't fit in the CSL. You can build much larger buffers using the internal CSoC SRAM or the external SRAM on the CSoC Board. One of the CSoC DMA controllers can manage the transfer of bytes from the data source to the SRAM and then interrupt the MCU when the buffer is full. In this section you will rebuild the PS/2 keyboard interface using a DMA controller to place ten scan codes from the keyboard into a buffer in SRAM. Then the 8032 MCU will read the buffer and display the ten keys that were pressed on the LED digit. While this design is overkill for the application, a keyboard is a convenient data source because you can control the arrival of keystrokes and easily see what the CSoC is doing.

The schematic for the modified keyboard interface circuit is shown in Figure 18. Falling edges of the **ps2_clock** signal strobe keyboard scan code bits from **ps2_data** into the **ps2_data_sreg** shift register. Each rising edge of **ps2_clock** sets the **rcv_active** flip-flop which indicates the receiver circuit is gathering data. The **bit_timer** counter is also cleared whenever **ps2_clock** is low. Once **ps2_clock** stays high at the end of the scan code transmission, the 25 MHz BusClock will have sufficient time to increment the counter until bit **bit_timer[11]** goes high. This takes $2^{11} \div 25$ MHz = 82 μs which is slightly longer than the 75 μs **ps2_clock** period. Once **rcv_active** and **bit_time[11]** are both set, this drives the **rcv_dma_comb** signal high which 1) clears the **rcv_active** flip-flop indicating the receiver is no longer active, and 2) sets the **rcv_dmareq** flip-flop. The scan code in the shift register is loaded into the **rcvData** register when the **rcv_dmareq** flip-flop is set, and a request to store the scan code in the buffer in SRAM is made to a DMA controller. The **rcv_dmareq** flip-flop is cleared on the very next clock cycle when **bit_time[0]** goes high. This is necessary because the DMA controller will log a request for every **BusClock** cycle that the **rcv_dmareq** signal is high. The DMA controller does a transfer for every request it logs. Because there is only one scan code to store in the buffer, the **rcv_dmareq** signal should only be high for one clock cycle.

When the DMA controller gets access to the CSI bus, it raises the **rcv_dmaack** signal to acknowledge the DMA request. This gates the scan code onto the CSI data bus through the **rcvDataRd** buffer. The scan code travels over the CSI data bus to either the internal or external SRAM. The DMA controller provides the address of the SRAM location where the scan code is stored. Then the memory address pointer is incremented and the counter that records the number of remaining transfers is decremented. The buffer is full when the transfer counter reaches zero, and the MCU is interrupted to process the buffer contents.

©2000 by XESS Corp.

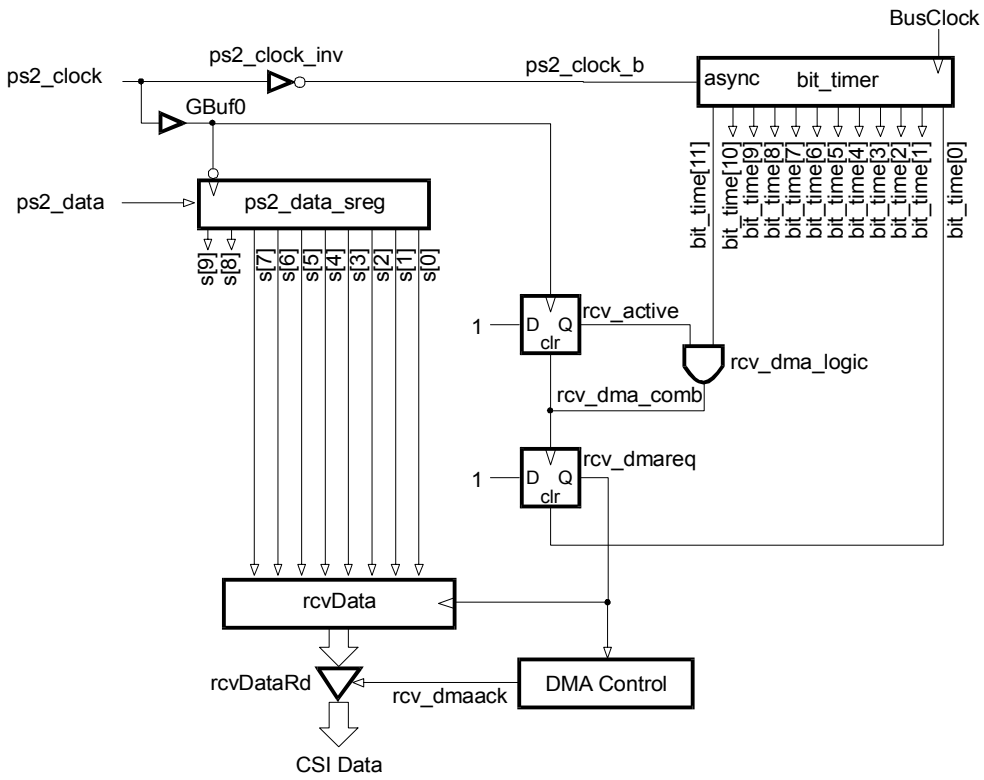
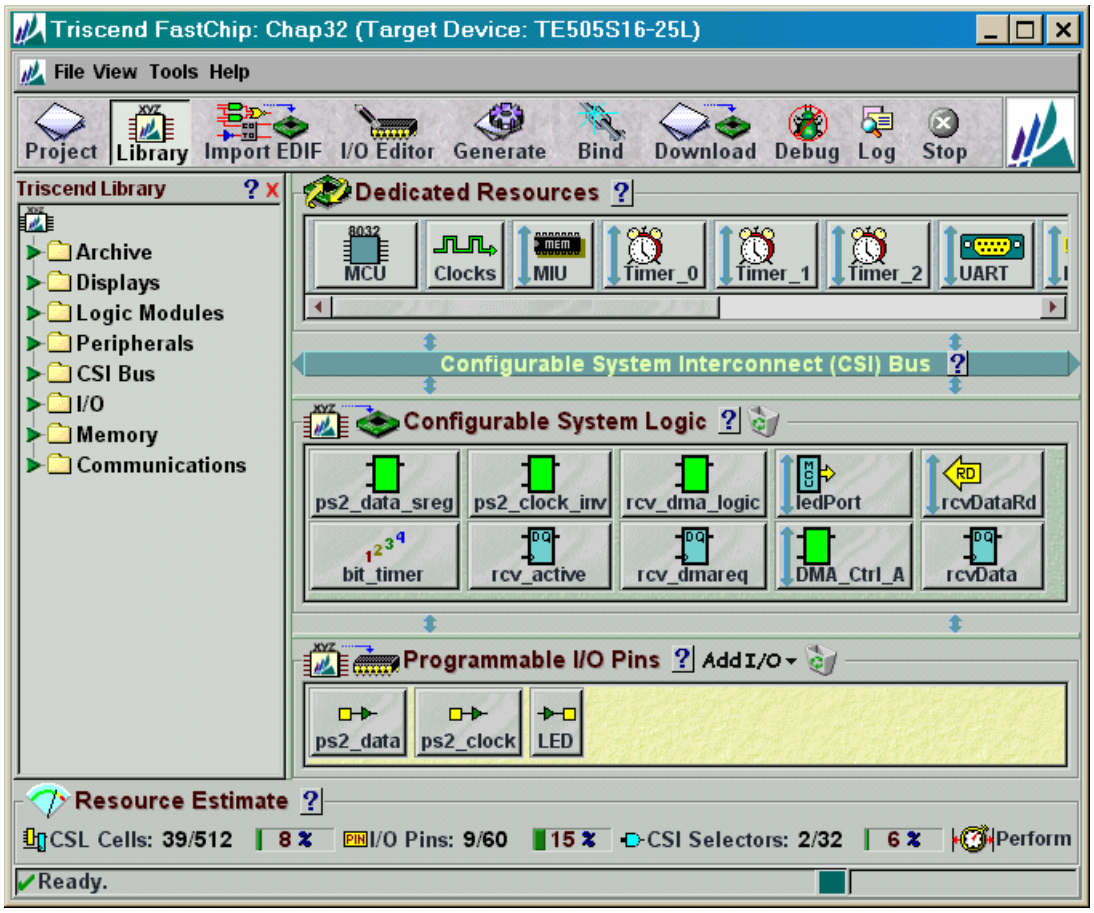
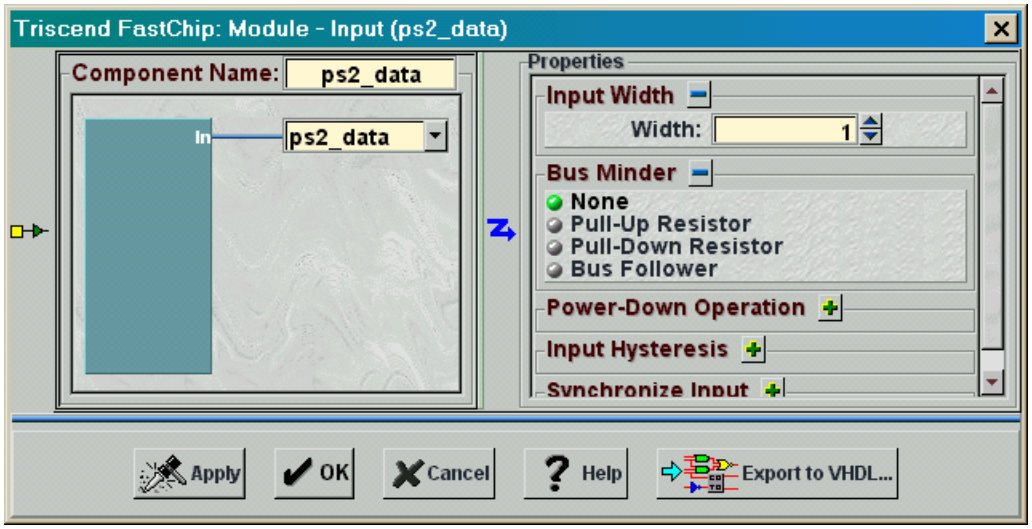


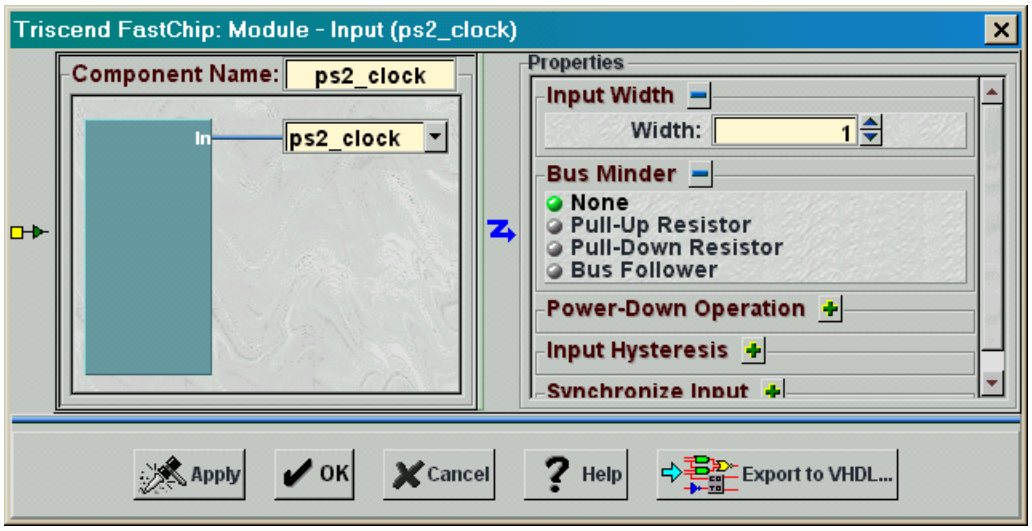
Figure 18: Schematic of a PS/2 keyboard interface circuit that uses DMA.

The ***Chap31*** FastChip project for the keyboard interface contains the modules shown below. The module and signal names in the FastChip project follow those used in Figure 18.

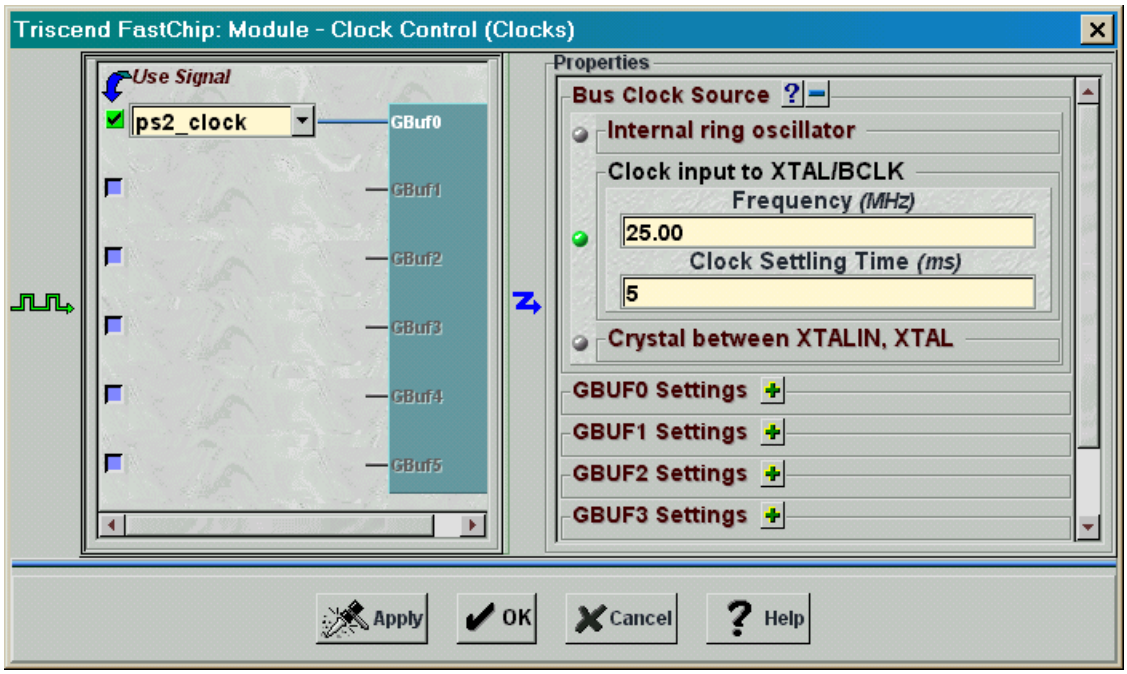


The PS/2 keyboard clock and data signals are brought in through standard input ports as follows:

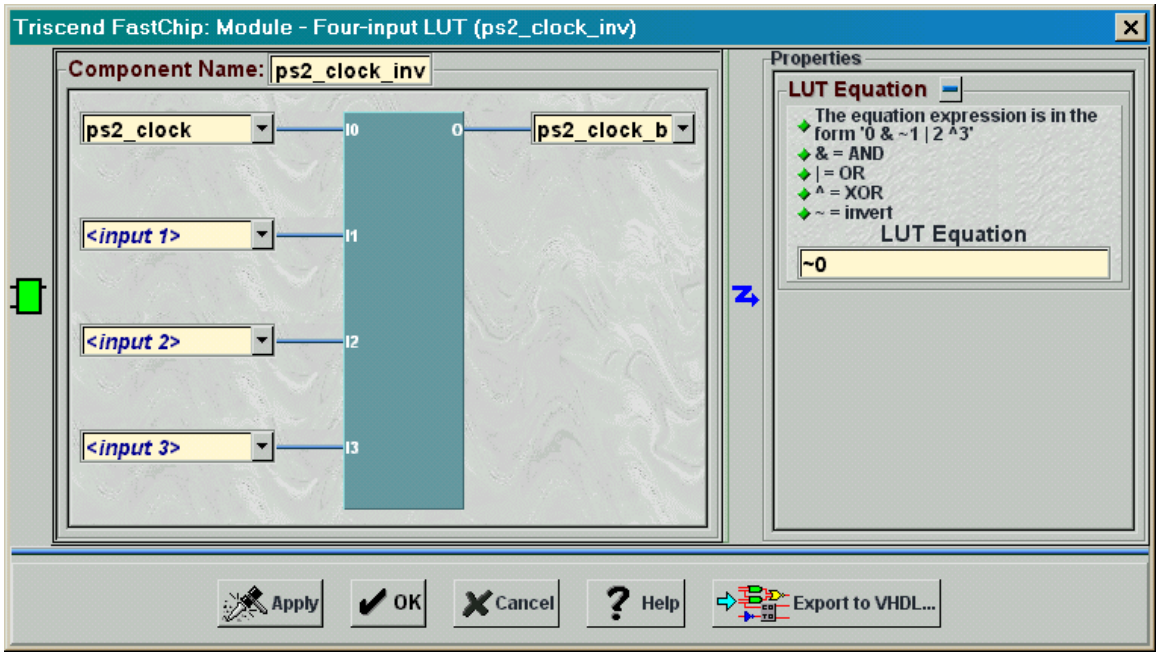




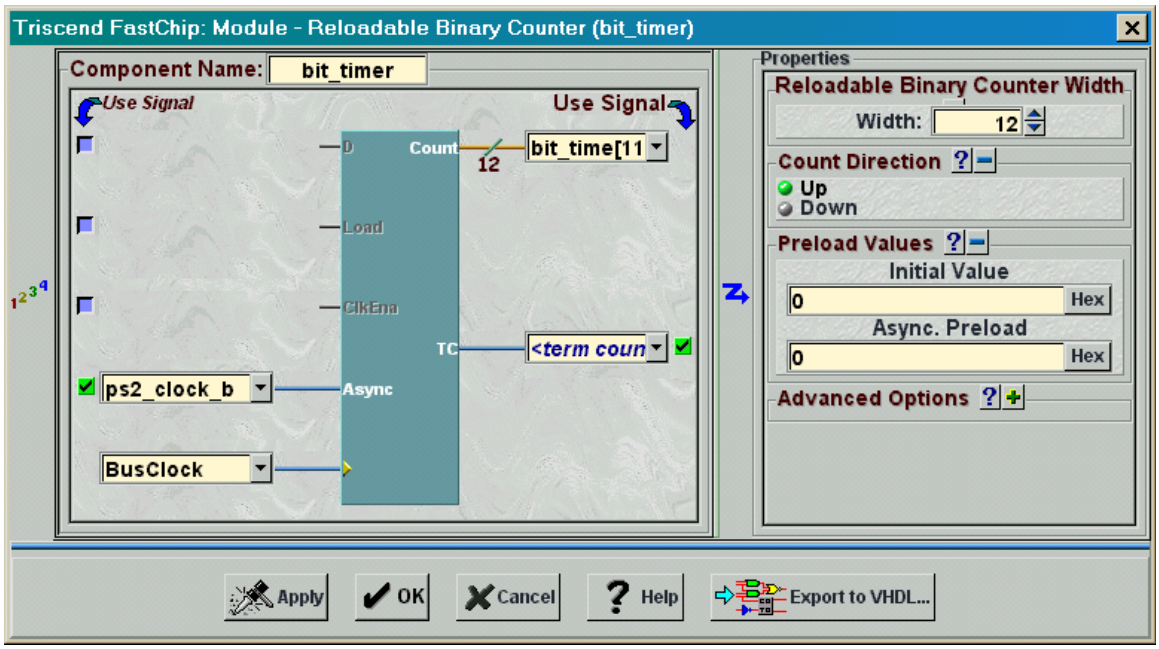
The **ps2_clock** signal is passed through global buffer **GBuf0** so it arrives at the flip-flop clock inputs with minimal skew. The external 25 MHz oscillator is also selected as the source for the **BusClock**.



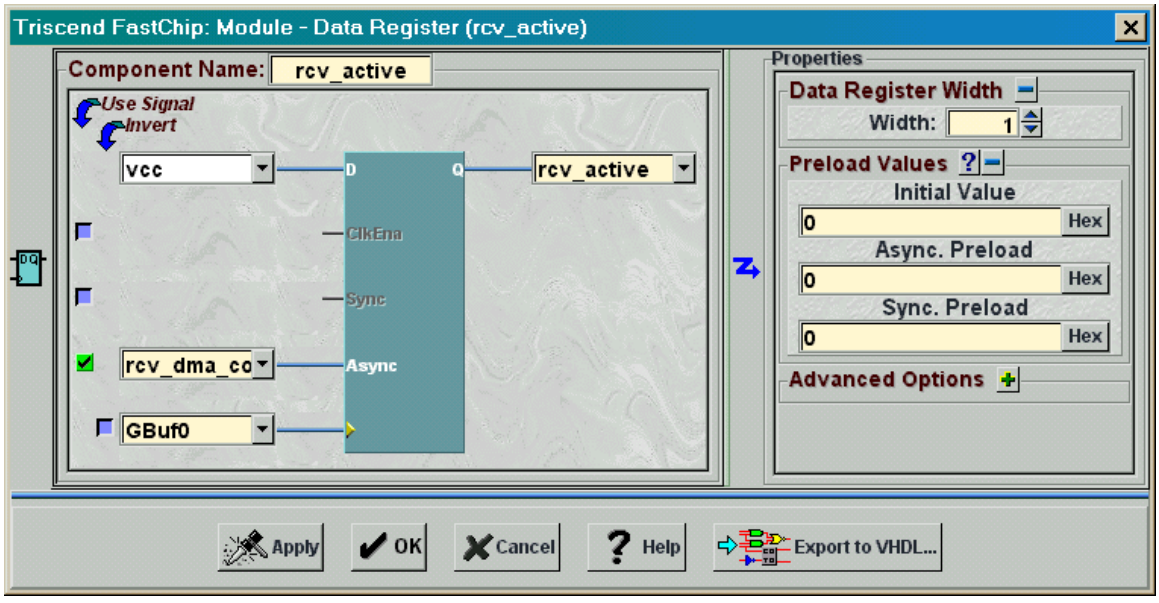
The PS/2 clock is also passed through an inverter to create the **ps2_clock_b** signal.



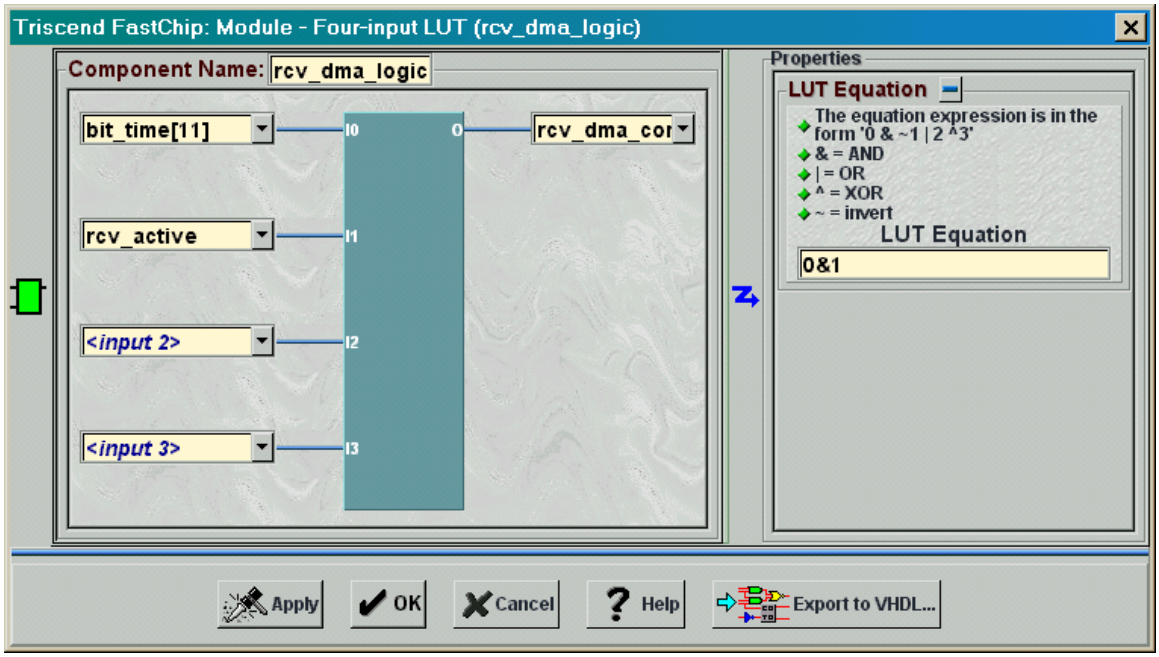
The **ps2_clock_b** signal is used to asynchronously clear the 12-bit timer (**bit_timer**) whenever the PS/2 clock is high . Otherwise, the 25 MHz BusClock increments **bit_timer**.



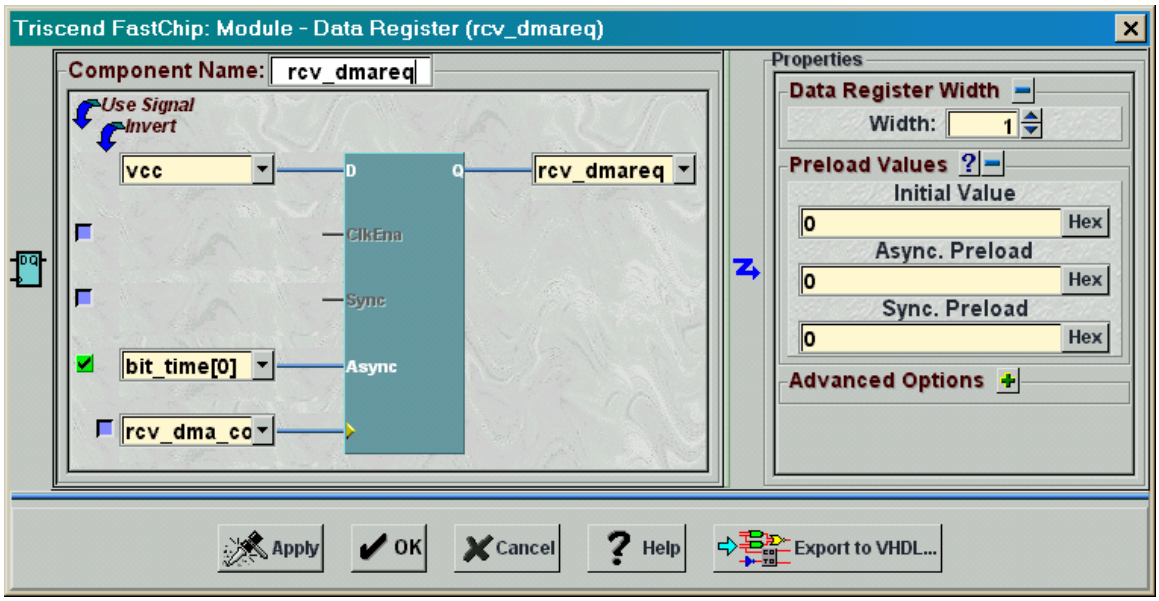
The **rcv_active** module is a flip-flop that is set by any rising edge of the PS/2 clock and cleared by a high level on the **rcv_dma_comb** signal.



The **rcv_dma_logic** LUT generates the **rcv_dma_comb** signal whenever the **bit_time[11]** and **rcv_active** signals are both high.

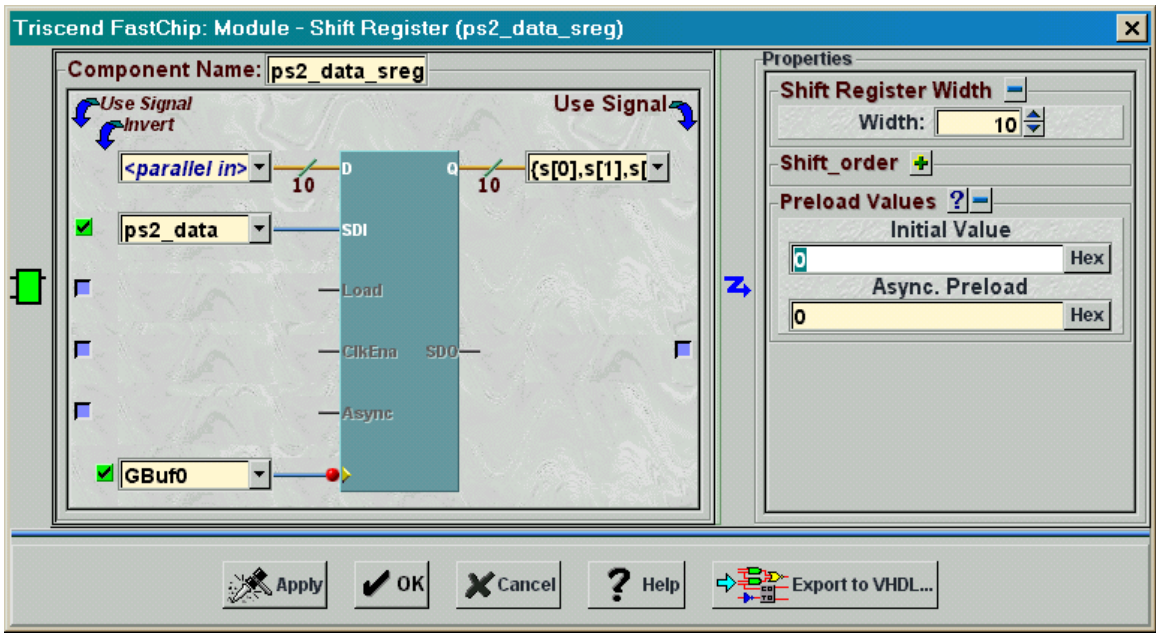


A rising edge on **rcv_dma_comb** sets the **rcv_dmareq** flip-flop. This requests a memory transfer operation from the DMA controller. The **rcv_dmareq** flip-flop is cleared by loading it with a zero when the **bit_time[0]** signal goes high on the next cycle of **BusClock**.

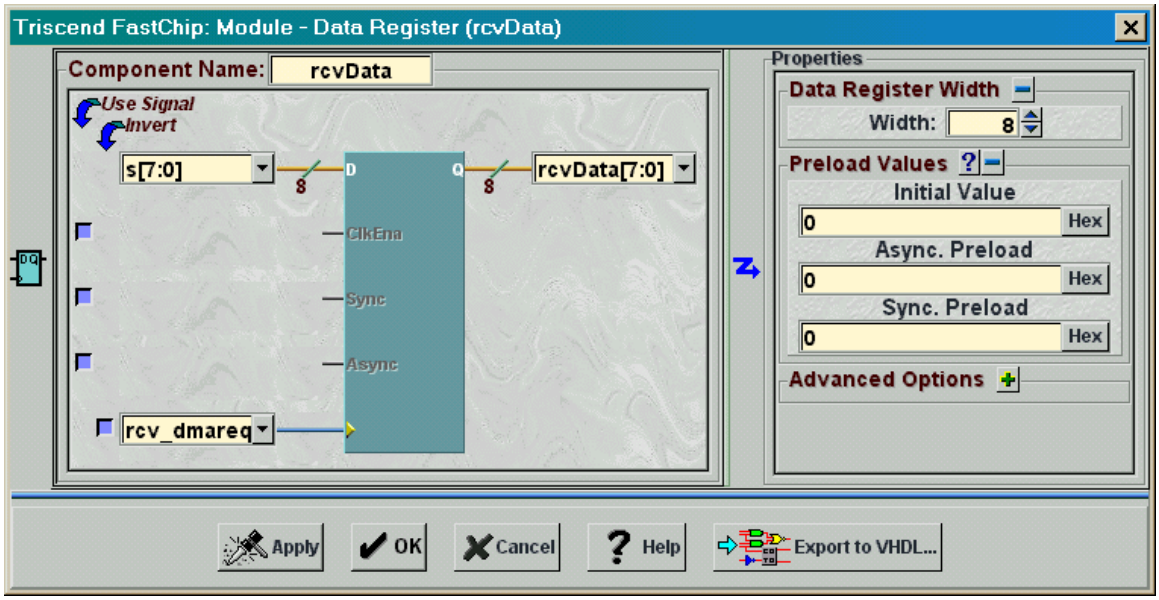


The PS/2 keyboard scan code enters a 10-bit shift register on the falling edges of the **GBuf0** signal (i.e. the PS/2 clock). The most-significant bit of **ps2_data_sreg** carries the least-significant bit of the keyboard data (**s[0]**) so the order of the signal

assignments typed into the <parallel out> field is reversed:
`{s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9]}`.

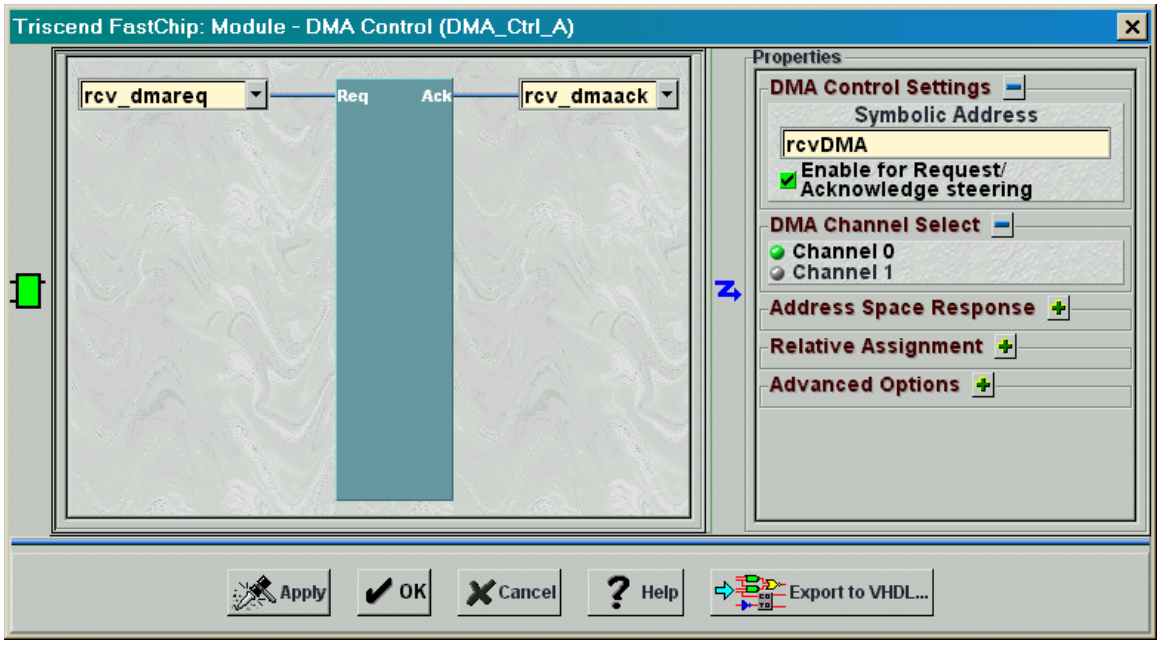


A rising edge on the **rcv_dmareq** signal clocks bits **s[7:0]** of of the scan code into the **rcvData** module. (Bits **s[9:8]** hold the parity and stop bit data which aren't needed by the MCU.)

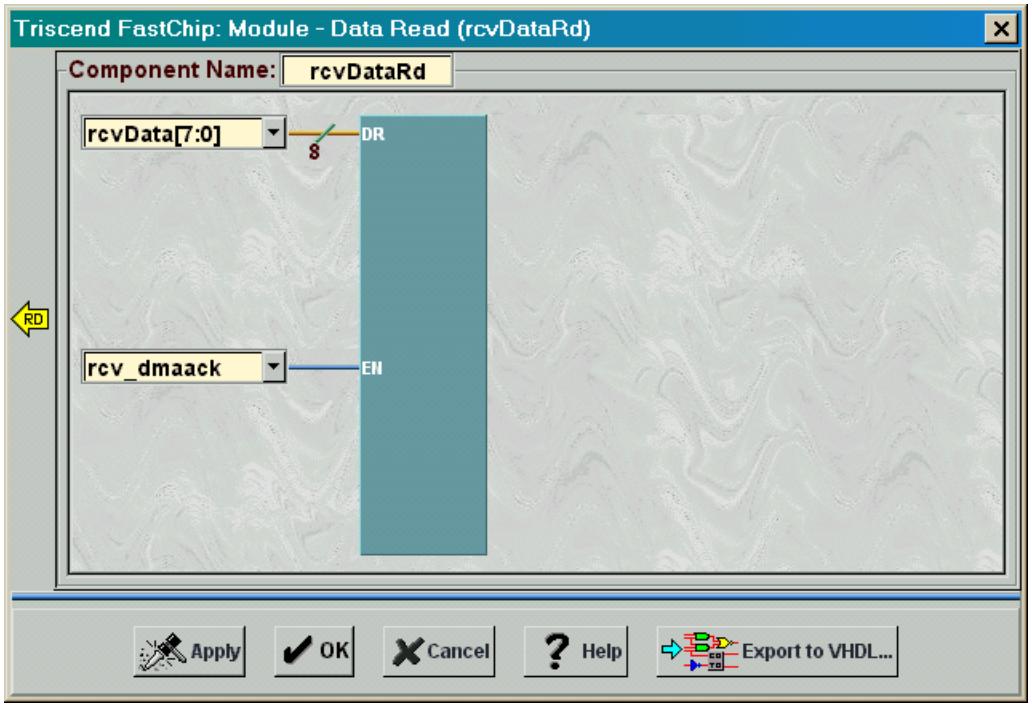


The **rcv_dmareq** signal is also attached to the Req input of the **DMA Control** module **DMA_Ctrl_A**. (The **DMA Control** module is found under the CSI Bus entry in the Library area of the FastChip project window.) There are two DMA controllers in the CSoC, so DMA controller 0 is chosen to handle the keyboard interface by clicking the associated radio button in the DMA Channel Select area of the **DMA Control** window. You also need

to check the box to steer the DMA requests and acknowledgements between the keyboard interface and the DMA controller circuitry.



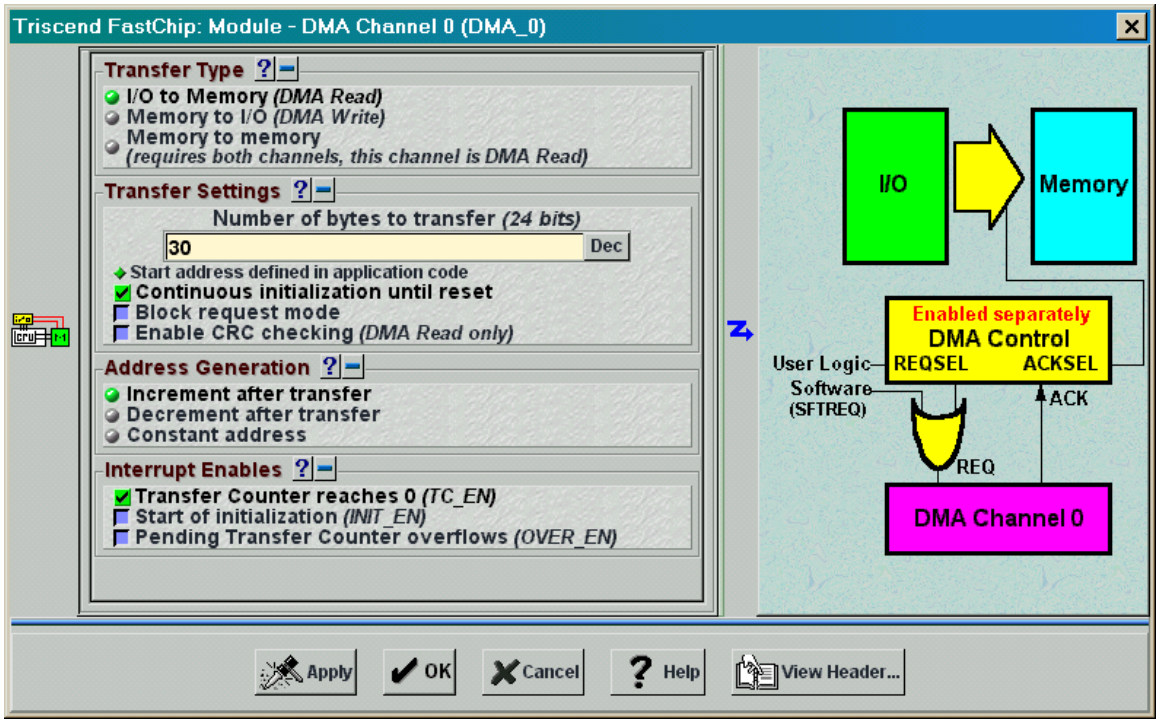
The **rcv_dmaack** acknowledge signal from the **DMA_Ctrl_A** module enables the **rcvDataRd** buffer. (The **Data Read** module is also found under the CSI Bus entry in the Library area.) The **rcvDataRd** buffer gets its input from the outputs of the **rcvData** register that hold the scan code. When **rcv_dmaack** goes high, the **rcvData[7:0]** signals are gated through to the CSI data bus.



Now you can set-up the DMA controller by clicking on the DMA 0 icon in the Dedicated Resources area of the FastChip project window. In the **DMA Channel 0** window that appears, click on the I/O to Memory radio button in the Transfer Type area since this DMA controller is transferring bytes from the keyboard interface to SRAM. Then check the Increment after transfer box in the Address Generation area so that the keyboard scan code buffer is filled starting from lower addresses and proceeding to higher addresses. Set the buffer size to thirty bytes by typing `30` into the Number of bytes to transfer box. Once the DMA controller buffers thirty bytes, it should re-initialize itself and begin storing scan codes at the start of the buffer again. Select this mode of operation by checking the Continuous initialization until reset box. Finally, the 8032 MCU should be interrupted when the buffer is full, so check the Transfer Counter reaches 0 in the Interrupt Enables area of the window.

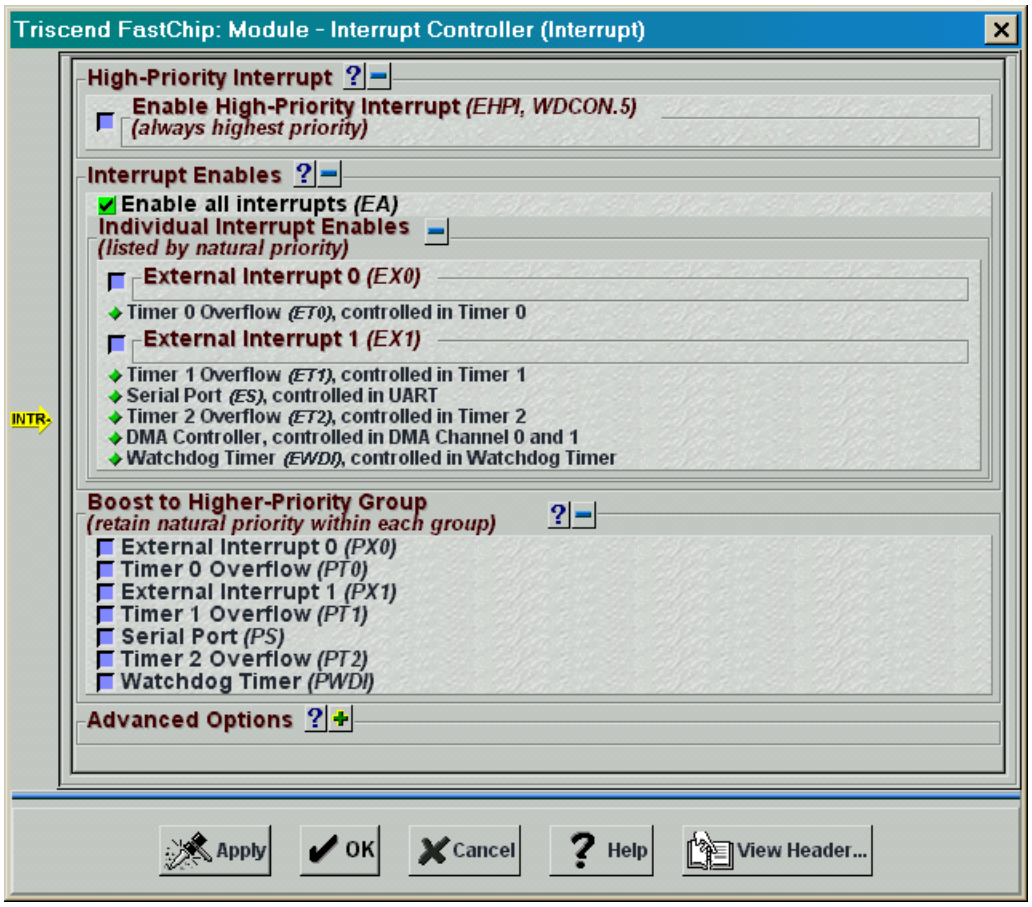
Why is the buffer size set to thirty bytes if you only want to buffer ten keystrokes? The answer is that every keystroke actually transmits three bytes: the scan code when the key is first pressed, and then a *break code* followed by a repeat of the scan code when the key is released. You haven't noticed the break code in any of your previous designs because it goes by so fast and the scan code that follows replaces it on the display. But the break code is there and you have to take it into account in this design.

Finally, what is the starting address of the scan code buffer in SRAM? The starting address is specified in the 8032 source code rather than in the **DMA Channel** window. You will see how to set it up later.

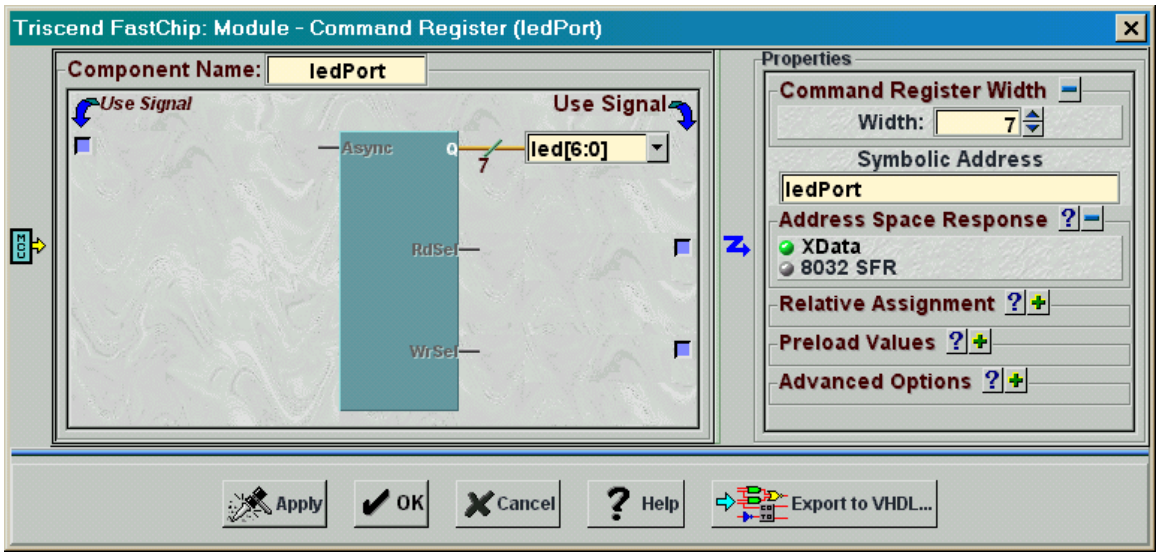


The 8032 MCU must be allowed to process interrupts in order for it to respond to the interrupt from the DMA controller. Click on the Interrupts icon in the Dedicated Resources

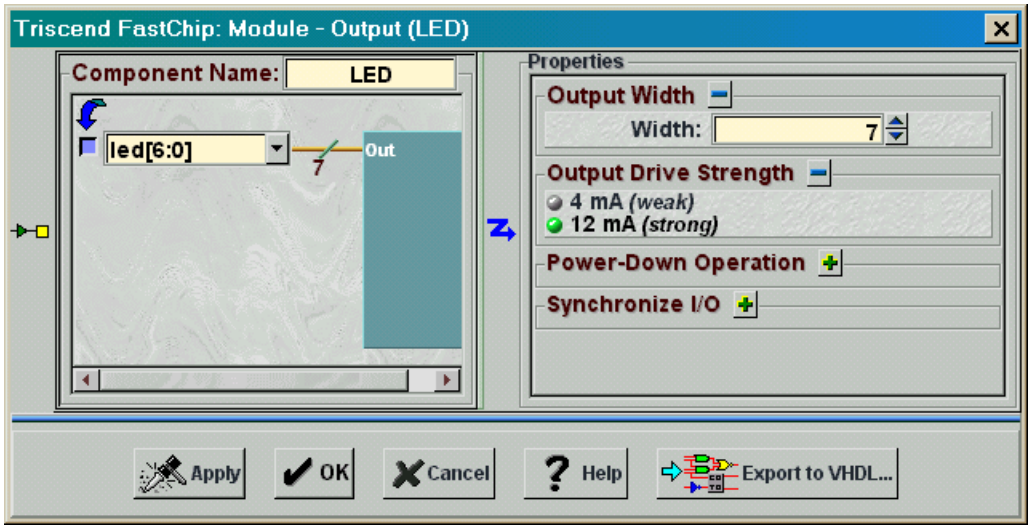
area of the FastChip project window and then check the Enable all interrupts box. Don't enable either of the external interrupts since you don't need these in this design. The 8032 application code will enable the individual interrupt from the DMA controller.



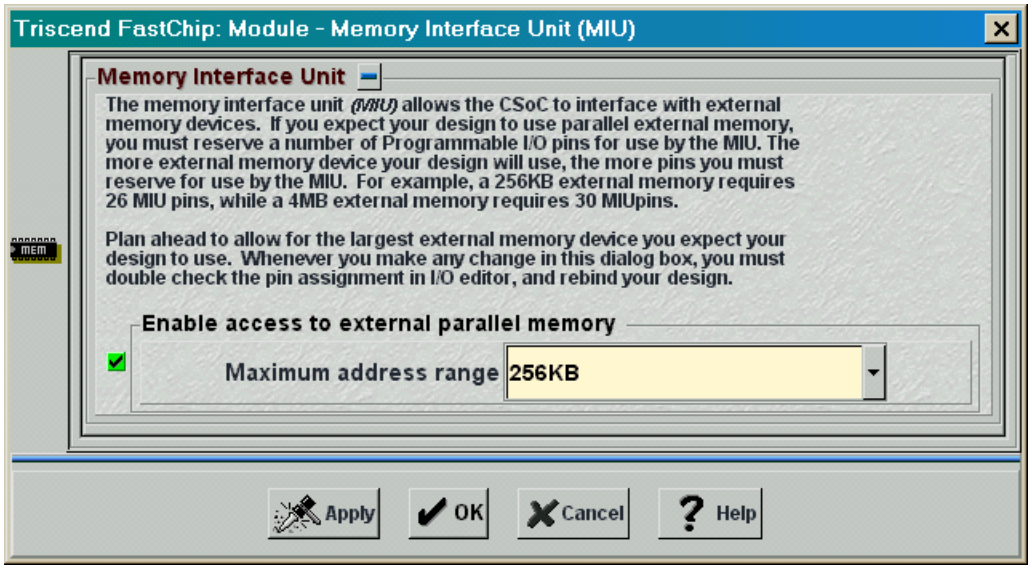
The 8032 MCU needs a means of displaying the buffered scan codes on the LED digit. The **ledPort** command register placed in the external data space can be written with a seven-bit pattern that will activate the LED segments.



Next, the seven outputs of the **ledPort** register are connected to seven output ports as shown below.



Finally, since the 8032 MCU code will be stored in the external SRAM, set-up the MIU as shown below. Click on the MIU icon in the Dedicated Resources area of the project window. This window lets you select how many address bits will be used by the MIU to access external memory. There are only 128 KBytes of external SRAM on your CSoC Board, so select the smallest address range in the drop-down list. This wastes one address bit and the SRAM contents will be replicated twice within the 256 KByte address range, but this won't cause any problems.



The modules and their interconnections have been instantiated. Now use the **I/O Editor** window to assign the pins as shown in Table 12 in the previous design.

Press the Generate icon on the toolbar and FastChip will create the chap32.h header file. The next step is to write your application code. Create a keil folder within your Chap32 FastChip project folder. Then start the Keil IDE and add the C code in Listing 8 to the ***chap32*** Keil project. The `main` routine initializes the 8032 MCU on line 5. Then the beginning address of the scan code buffer is loaded into the source address registers of DMA controller 0 (lines 7–9). The upper byte of the 32-bit physical address is not set because only the lower 24 bits are decoded. The next byte is set to 0x01 which selects the internal 16 KByte SRAM of the CSoC (as per our discussion of address mappers in Chapter 2). The lower two bytes are set to 0x3F00 so the buffer is placed at the start of the last page of the internal SRAM. Then the control bits that enable and initialize DMA controller 0 are set on line 11. At this point the DMA controller is able to respond to DMA requests from the keyboard interface. The main routine enters an infinite loop on line 13.

All the work is actually done in the `displayDMABuffer` interrupt subroutine (lines 70–88). This subroutine is called when DMA controller 0 issues an interrupt (interrupt identifier 7). The interrupt subroutine checks the cause of the interrupt by reading the DMA interrupt flags on line 77. The 8032 exits the subroutine if the DMA interrupt was not a result of the transfer counter reaching zero as indicated by bit 0 in the `DMAINT0` register being set. If the transfer counter has reached zero, then the buffer is filled with

keyboard scan codes. The loop on lines 80–83 reads the scan codes from the buffer and displays them on the LED digit using the `displayPs2Data` subroutine. The pointer to the buffer is incremented by three on each loop iteration so as to skip the break code and the repeated scan code that are sent out when the key is released. After all the scan codes in the buffer are displayed, the `displayDMABuffer` subroutine writes a logic 1 to the transfer counter interrupt bit which clears this interrupt. Then control passes from the interrupt subroutine and returns to the infinite loop in the main routine.

The `displayPs2Data` subroutine is passed a scan code and searches for it in the table defined on lines 29–42. If a matching scan code is found in the table, the subroutine writes the associated LED segment activation pattern to the **ledPort** register (line 54). After the digit is displayed, the `wait` subroutine on lines 18–24 is called. `wait` uses the watchdog timer to insert a delay of 1/3 of a second. Upon returning to `displayPs2Data`, the LED segments are all turned off (line 56) and another 1/3 second interval is inserted. The blanked LED makes it easier to separate the buffer entries as they are displayed so you can count the number of scan codes in the buffer.

Listing 8: Keyboard interface interrupt-handling code.

```
1   #include "..\Chap32.h"
2
3   main()
4   {
5        Chap32_INIT();
6
7        DMASADR0_0 = 0x00;  // DMA buffer starts
8        DMASADR0_1 = 0x3F;  // at address 0x3F00
9        DMASADR0_2 = 0x01;  // in 16 KByte internal SRAM
10
11       DMACTRL0_0 |= 0x06; // enable and initialize DMA channel 0
12
13       while(1);           // wait for DMA interrupts
14  }
15
16
17  // use the watchdog timer to wait for about 1/3 second
18  static void wait()
19  {
20       CKCON = 0x80;  // set timeout period to 2^23 clock cycles
21       TA = 0xAA; TA = 0x55;    // enable timed access to WDCON
22       WDCON = 0x01;  // clear WDIF and reset watchdog timer
23       while(WDIF==0);     // wait for watchdog timer to expire
24  }
25
26  #define ERROR 0x79;
27
28  // translate keyboard scan codes to LED segment activations
```

```
29  typedef struct{ unsigned char ps2Data, led; } ps2XlateEntry;
30  ps2XlateEntry ps2XlateTbl[] =
31  {
32       { 0x16, 0x06 },     // "1"
33       { 0x1E, 0x5B },     // "2"
34       { 0x26, 0x4F },     // "3"
35       { 0x25, 0x66 },     // "4"
36       { 0x2E, 0x6D },     // "5"
37       { 0x36, 0x7D },     // "6"
38       { 0x3D, 0x07 },     // "7"
39       { 0x3E, 0x7F },     // "8"
40       { 0x46, 0x6F },     // "9"
41       { 0x45, 0x3F } // "0"
42  };
43
44
45  // flash scan code key on LED digit
46  static void displayPs2Data(unsigned char c)
47  {
48       unsigned int i;
49
50       // search the translation table for the scan code
51       for(i=0; i<sizeof(ps2XlateTbl)/sizeof(ps2XlateEntry); i++)
52            if(ps2XlateTbl[i].ps2Data == c)
53            { // found a matching scan code in the table
54                 ledPort = ps2XlateTbl[i].led; // display digit
55                 wait();                       // for 1/3 second
56                 ledPort = 0x00;          // then clear display
57                 wait();                  // and wait 1/3 second
58                 return;
59            }
60
61       // no matching scan code was found, so display "E"
62       ledPort = ERROR;    // display "E"
63       wait();        // for 1/3 second
64       ledPort = 0x00;     // then clear display
65       wait();        // and wait 1/3 second
66  }
67
68
69  // display all the entries in the DMA buffer
70  static void displayDMABuffer() interrupt 7 using 0
71  {
72       unsigned char xdata* buffer;
73       int i;
74
75       // exit subroutine if interrupt is not caused by the
```

```
76        // transfer counter of DMA channel 0 reaching 0
77        if(!(DMAINT0 & 1)) return;
78
79        // display the scan code keys stored in the DMA buffer
80        for(i=0, buffer=0x3F00; i<30; i+=3,buffer+=3)
81        {
82             displayPs2Data(*buffer);
83        }
84
85        // clear the interrupt caused when the transfer count
86        // of DMA channel 0 reaches 0
87        DMAINT0 &= 1;  // setting the bit clears it
88   }
```

Once you set the compiler and linker options as you did in the previous design, you can compile and link the ***Chap32*** Keil project. Then re-enter the FastChip project window and bind your design. Download the keyboard interface circuitry and the 8032 program in the Chap32.HEX file to your CSoC Board. Finally, use dScope to establish a debugging link to the CSoC Board and then reset and execute the application program. At this point, you should be able to type a sequence of ten numeric keys on a keyboard attached to the PS/2 port of your CSoC Board and then see the same sequence of numbers appear on the LED digit over an interval of about six seconds.