3

8032 MCU + Soft Modules

Objectives

- Learn how to add soft modules to an MCU-based CSoC design.
- Learn how the MCU communicates with soft modules.
- Learn how to use the DMA controllers of the CSoC.

Miscellaneous Resources in the CSoC

You have created designs using the soft modules in Chapter 1 and the 8032 MCU core in Chapter 2. Now you will do *hardware/software codesign* by combining soft modules with the MCU. The new areas of the CSoC you will use are:

- the *selectors* that activate the interface between soft modules and the 8032 data bus,
- the *direct memory access (DMA) controllers* that can transfer data to and from SRAM independently of the MCU*,*
- the *bus arbiter unit* that controls access to the CSI address and data buses.

These areas are highlighted in Figure 15.

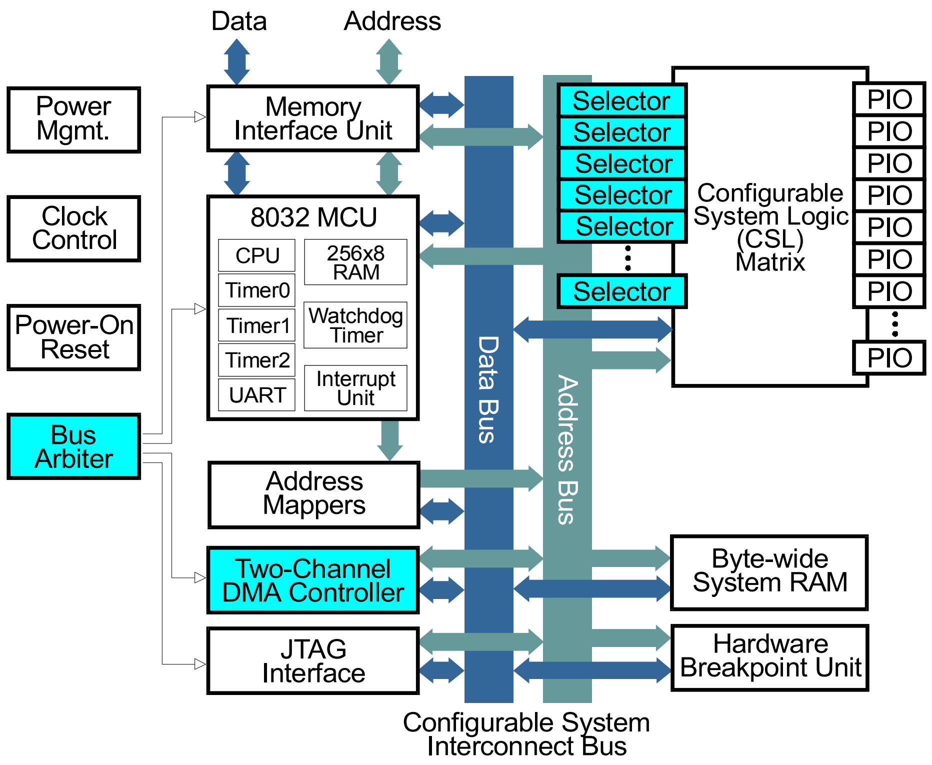


Figure 15: Areas of the Triscend CSoC used in the designs in this chapter.

The Selectors

The TE505 CSoC has 32 address selectors, each of which are programmable to trigger for a range of values on the 32-bit CSI address bus. If a physical address activates a selector, the selector signals to the CSL whether a read or write operation to that address is under way. A soft module in the CSL can use the read/write signals from the selector to gate its output onto the CSI data bus or get data from the CSI data bus, respectively. In this way, the selectors serve to bind the functions implemented in the CSL to locations in the 8032 code, data, and SFR address spaces.

The DMA Controllers

The Triscend CSoC has two independent DMA controllers that can independently

- write data from memory to an I/O device (usually a soft module in the CSL), or
- read data from an I/O device to memory.

The two DMA controllers can also be paired to perform memory-to-memory data transfers.

Each DMA controller has registers to store the number of bytes to be transferred and the 32-bit starting address in memory. A DMA controller can transfer the bytes one at a time under control of a DMA request signal, or a single request can initiate the transfer of an entire block of data. Once completed, the transfer can be automatically repeated or the DMA controller can shut down until it is needed again. The MCU can be interrupted whenever either DMA controller starts or finishes its task.

The Bus Arbiter

The MCU, DMA controllers, and JTAG interface can all attempt to use the CSI Bus at the same time. The bus arbiter grants and denies them access to the CSI Bus so they don't interfere with each other. The MCU has the highest priority for performing CSI Bus transfers while the DMA controllers and JTAG interface are forced to wait. The JTAG interface is usually involved in configuring the CSoC and managing the debugging process so any delays in accessing the CSI Bus will not cause a problem.

Design 3.1 - PS/2 Keyboard Interface to the 8032 MCU

In Chapter 1 you built a circuit from soft modules that received a serial bit stream from a keyboard and displayed the active key on the LED digit. Now you will repeat that design using a combination of soft modules and the 8032 MCU.

You could do this design entirely in software by interrupting the MCU on each falling edge of the keyboard clock and appending the current data bit onto a scan code variable. The keyboard clock period is about 75 μs and each 8032 instruction takes four to eight cycles of the 25 MHz clock, so the MCU can execute approximately 300 instructions between interrupts. This is certainly feasible if the MCU isn't handling any other tasks that have frequent, hard deadlines that can't be missed. But a better design would move the collection of the individual bits into hardware that only interrupts the MCU when a complete keyboard scan code is available. This reduces the load on the 8032 because a keyboard only transmits a maximum of about ten keystrokes per second.

The schematic for a keyboard interface circuit is shown in Figure 16 and some of the waveforms are shown in Figure 17. Falling edges of the **ps2_clock** signal strobe 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_int_comb** signal high which 1) clears the **rcv_active** flip-flop indicating the receiver is no longer active, and 2) sets the **rcv_int** flip-flop that strobes the scan code from the shift register into the

rcvData register and sends an interrupt (**INTR0**) to the 8032 MCU. When the MCU responds to this interrupt, it reads the keyboard data from the logical address of the **rcvData** register. The presence of the **rcvData** register address on the CSI address bus triggers a selector. The **RdSel** signal goes high and this gates the scan code byte onto the CSI data bus while simultaneously clearing the interrupt from the **rcv_int** flip-flop.

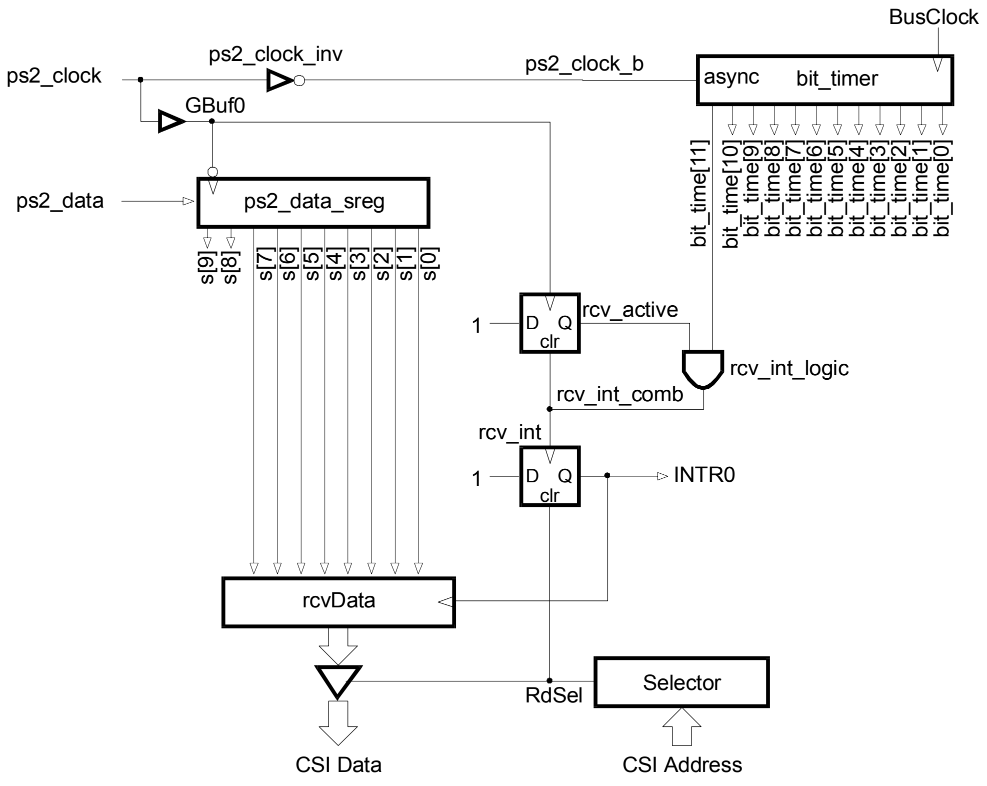


Figure 16: Schematic of a PS/2 keyboard interface circuit that uses interrupts.

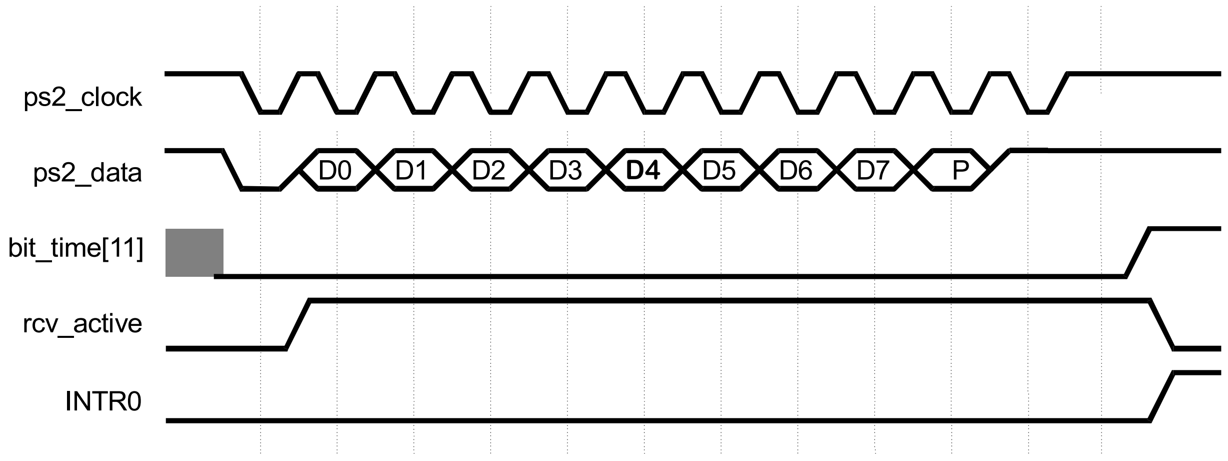
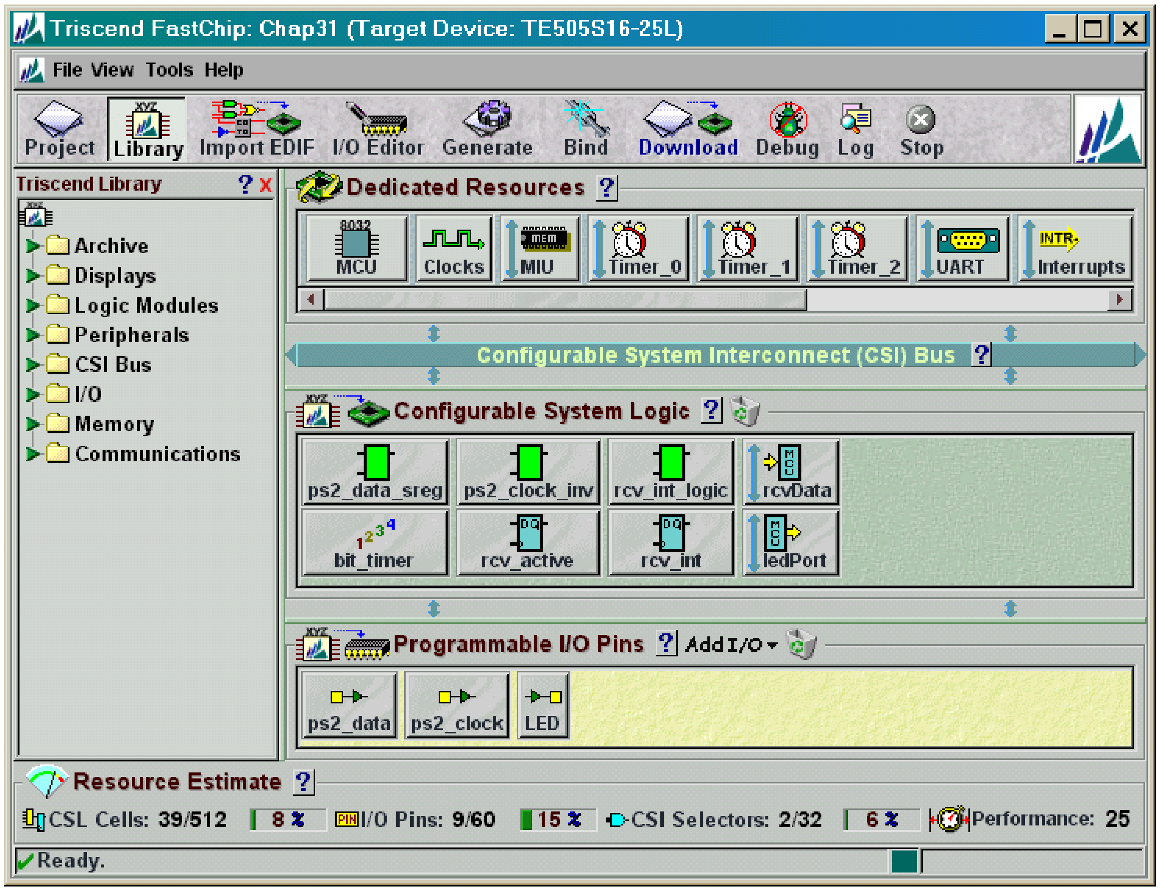


Figure 17: Keyboard interface waveforms.

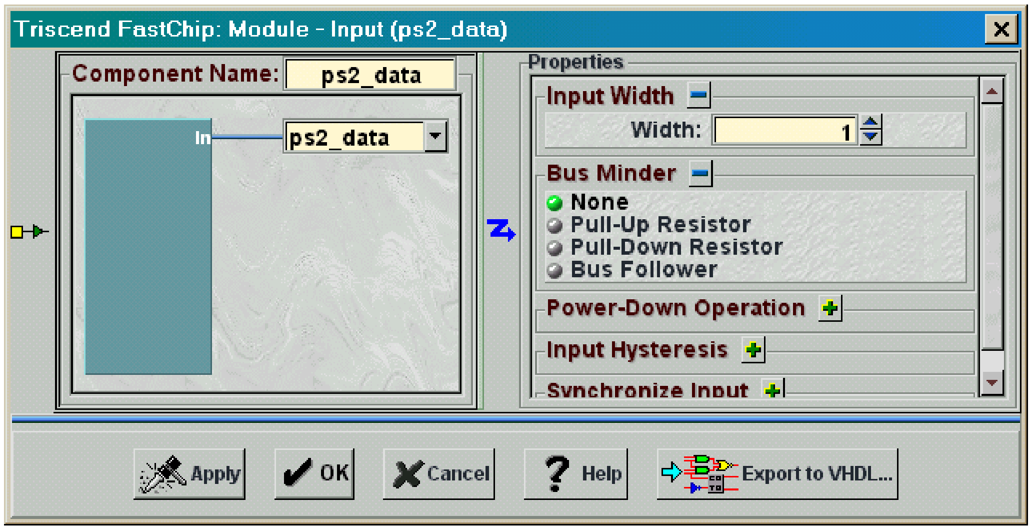
The Chap31 FastChip project for the keyboard interface contains the modules shown below. The module and signal names in the FastChip project follow those used in Figure 16.



CSL Cells: 39/512 | 8 % | I/O Pins: 9/60 | 15 % | CSI Selectors: 2/32 | 6 % | Performance: 25

Ready.

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



Triscend FastChip: Module - Input (ps2_data)

Component Name: ps2_data

Properties

Input Width

Width: 1

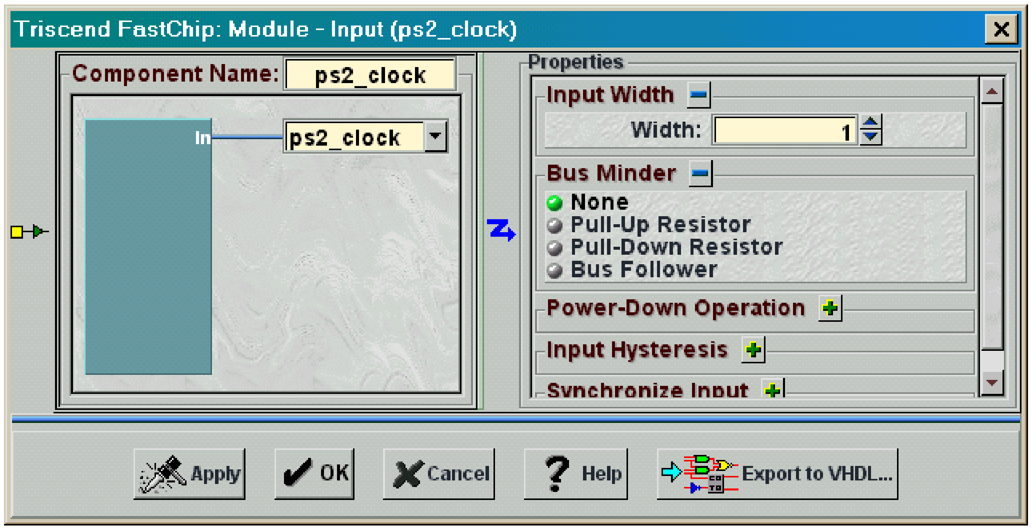
Bus Minder

● None
○ Pull-Up Resistor
○ Pull-Down Resistor
○ Bus Follower

Power-Down Operation

Input Hysteresis

Synchronize Input



Triscend FastChip: Module - Input (ps2_clock)

Component Name: ps2_clock

Properties

Input Width

Width: 1

Bus Minder

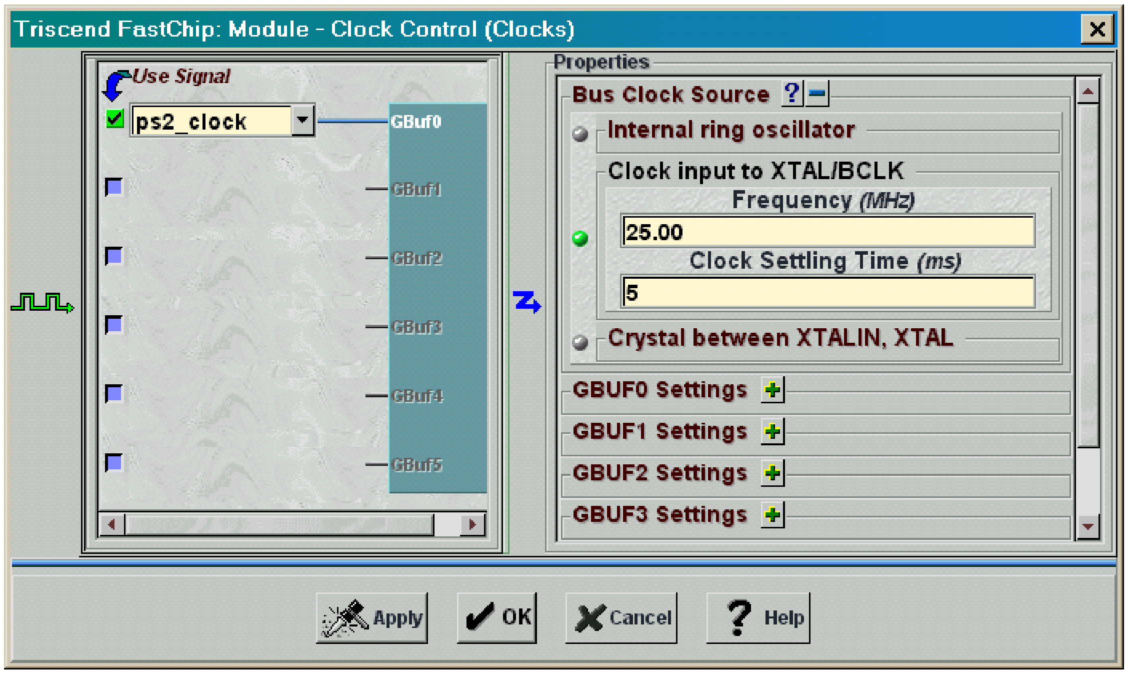
● None
○ Pull-Up Resistor
○ Pull-Down Resistor
○ Bus Follower

Power-Down Operation

Input Hysteresis

Synchronize Input

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**.

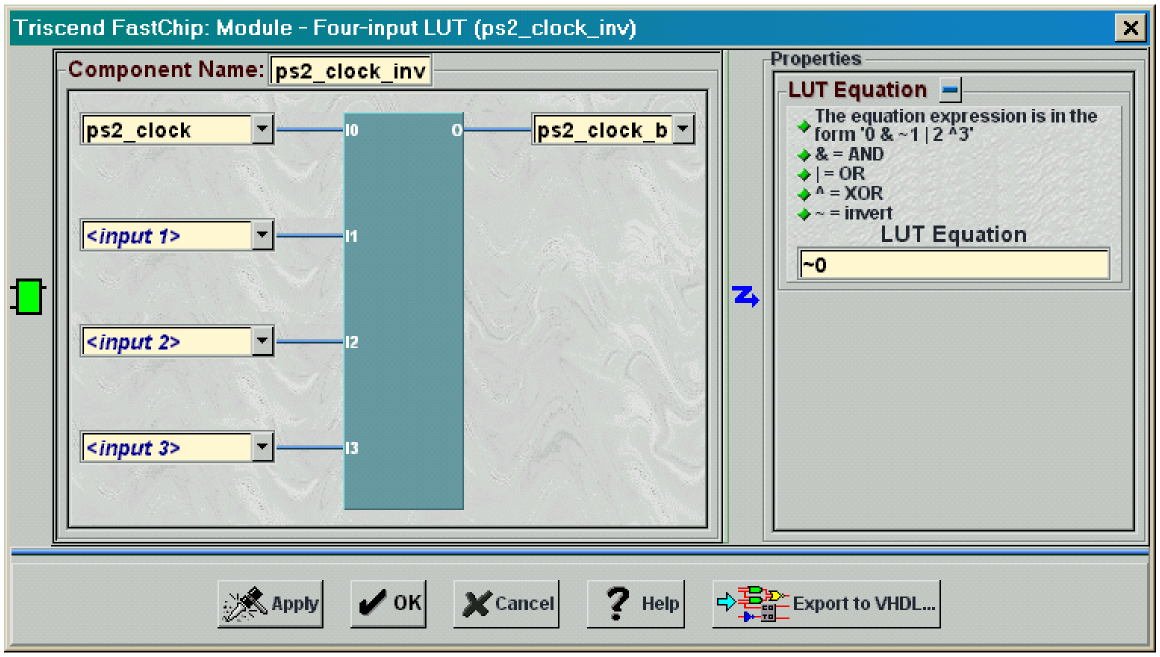

Triscend FastChip: Module - Clock Control (Clocks)

Use Signal: ps2_clock → GBuf0

Properties

Bus Clock Source
- Internal ring oscillator
- Clock input to XTAL/BCLK
  - Frequency *(MHz)*: 25.00
  - Clock Settling Time *(ms)*: 5
- Crystal between XTALIN, XTAL

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


Triscend FastChip: Module - Four-input LUT (ps2_clock_inv)

Component Name: ps2_clock_inv

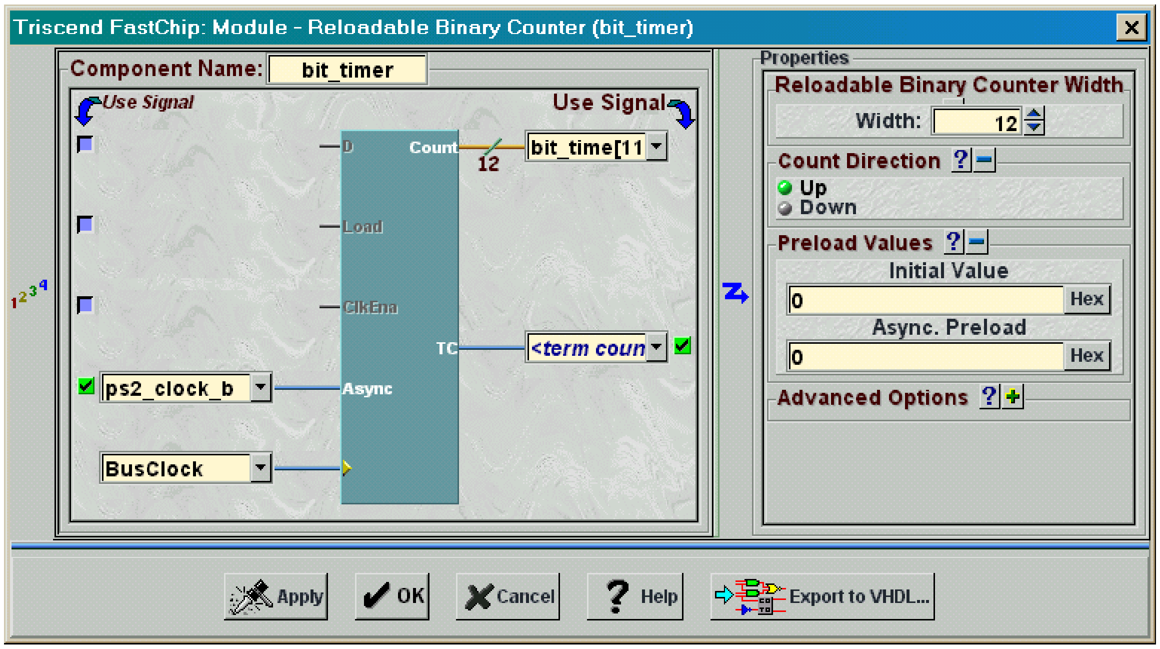
I0: ps2_clock
I1: <input 1>
I2: <input 2>
I3: <input 3>
O: ps2_clock_b

Properties

LUT Equation
- The equation expression is in the form '0 & ~1 | 2 ^3'
- & = AND
- | = OR
- ^ = XOR
- ~ = invert

LUT Equation: ~0

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**.



Triscend FastChip: Module - Reloadable Binary Counter (bit_timer)

Component Name: bit_timer

Use Signal — D, Load, ClkEna; ps2_clock_b → Async; BusClock → clock

Count → bit_time[11 (12); TC → <term coun

Properties

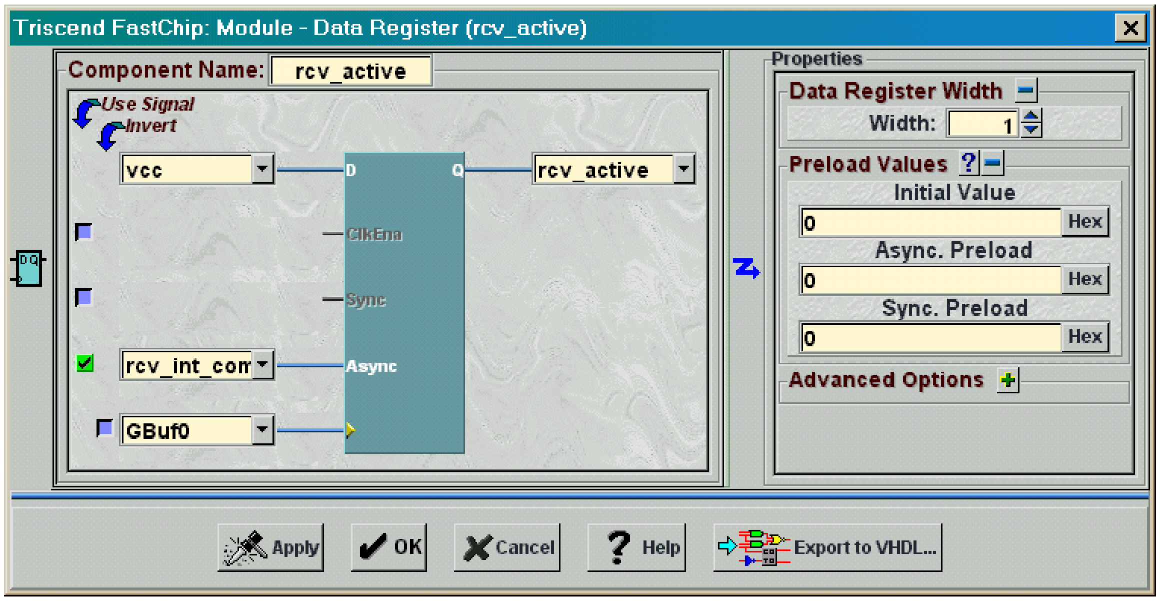
Reloadable Binary Counter Width — Width: 12

Count Direction: [✓ Up] Down

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

Advanced Options

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_int_comb** signal.



Triscend FastChip: Module - Data Register (rcv_active)

Component Name: rcv_active

Use Signal / Invert — vcc → D; ClkEna; Sync; rcv_int_com → Async; GBuf0 → clock

Q → rcv_active

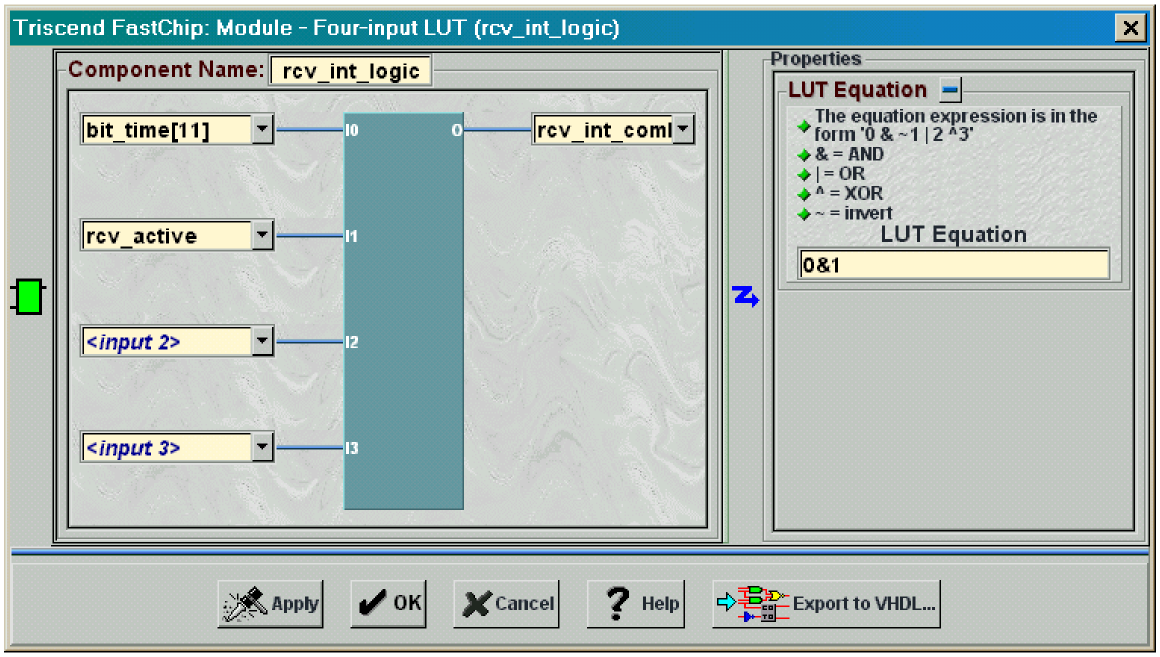
Properties

Data Register Width — Width: 1

Preload Values — Initial Value: 0 Hex; Async. Preload: 0 Hex; Sync. Preload: 0 Hex

Advanced Options

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



Triscend FastChip: Module - Four-input LUT (rcv_int_logic)

Component Name: rcv_int_logic

I0: bit_time[11]
I1: rcv_active
I2: <input 2>
I3: <input 3>
O: rcv_int_coml

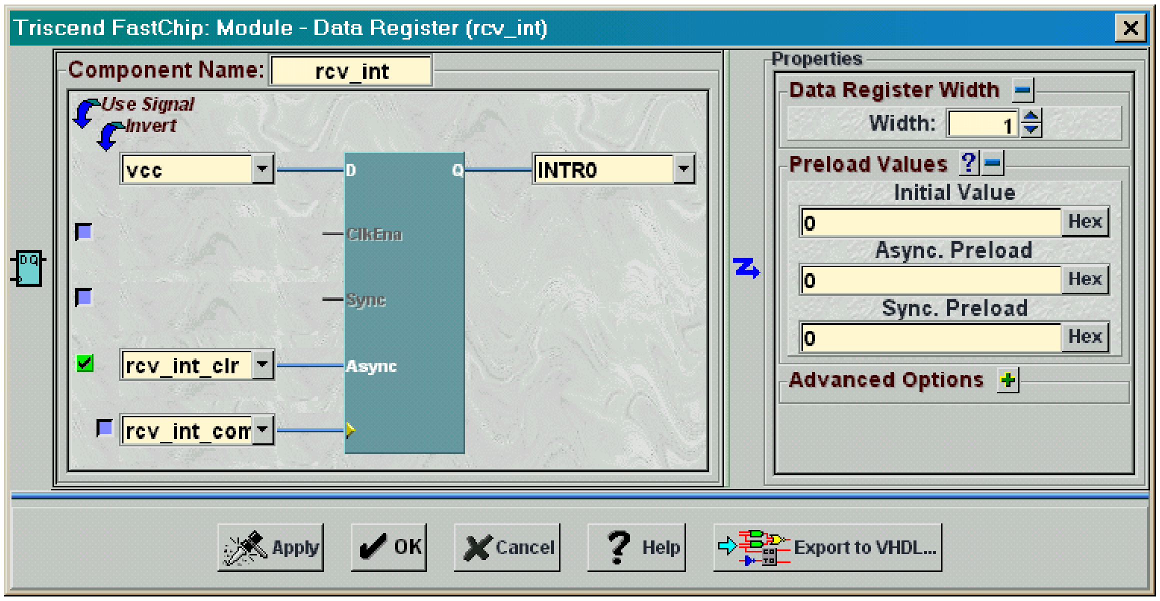
Properties

LUT Equation

- The equation expression is in the form '0 & ~1 | 2 ^3'
- & = AND
- | = OR
- ^ = XOR
- ~ = invert

LUT Equation: 0&1

A rising edge on **rcv_int_comb** clocks a one into the **rcv_int** flip-flop. This sends an interrupt to the MCU on the dedicated **INTR0** interrupt line. The **rcv_int** flip-flop is cleared by loading it with a zero whenever the **rcv_int_clr** signal is high.



Triscend FastChip: Module - Data Register (rcv_int)

Component Name: rcv_int

Use Signal / Invert

D: vcc
Q: INTR0
ClkEna
Sync
Async: rcv_int_clr
Clock: rcv_int_com

Properties

Data Register Width

Width: 1

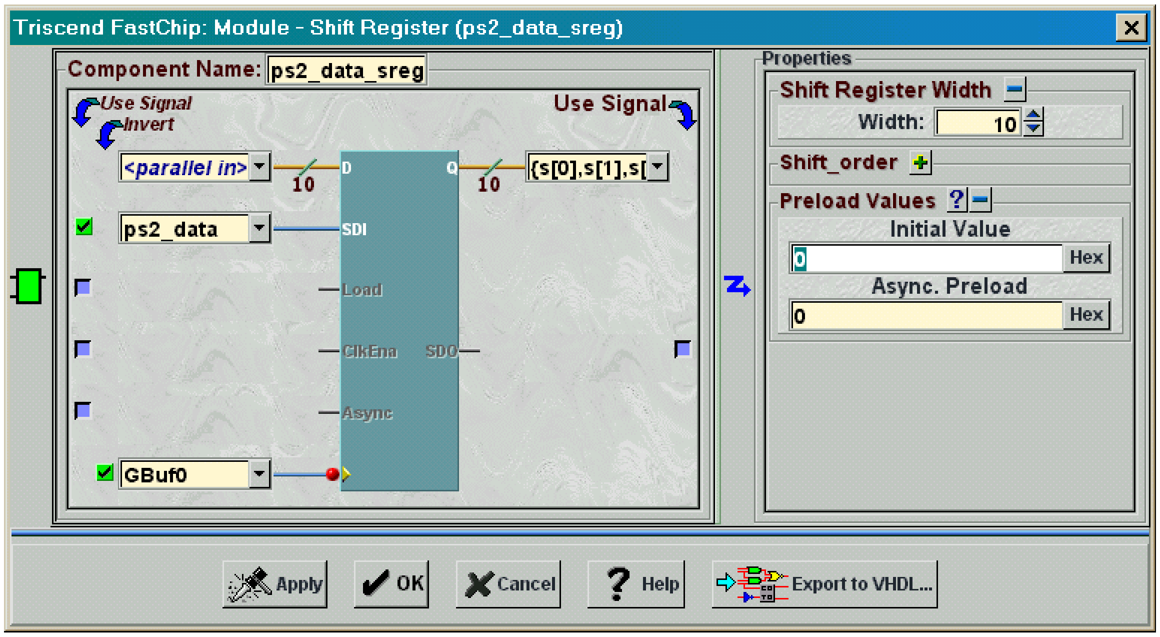
Preload Values

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

Advanced Options

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 scan code (**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]}`.


Triscend FastChip: Module - Shift Register (ps2_data_sreg)

Component Name: ps2_data_sreg

Properties

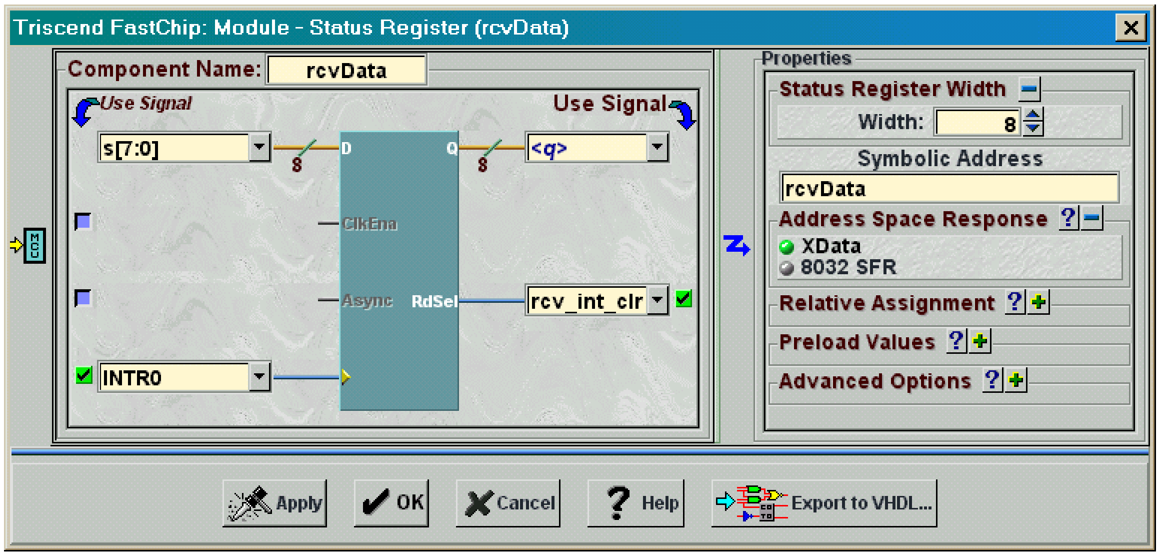
Shift Register Width — Width: 10

Shift_order

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

A rising edge on the **INTR0** interrupt signal clocks bits **s[7:0]** of of the scan code into the **rcvData** status register module. (Bits **s[9:8]** hold the parity and stop bits which aren't needed by the MCU.) The MCU can read the contents of a status register module using the CSI address and data buses. These connections are implied just by instantiating the status register so you don't have to specify them. But if you also wanted to use the register contents in other soft modules you would type the connecting signal names into the <q> field.

The **rcvData** status register is assigned the symbolic name rcvData and placed in the external data address space of the 8032 MCU. FastChip will assign a logical address for this register when you press the Generate icon in the toolbar. When the MCU reads from this address, the value in the register will enter the MCU. The **RdSel** output from the **rcvData** register will also pulse high. This output is attached to the **rcv_int_clr** signal and will clear the **rcv_int** interrupt flip-flop. That prevents the same scan code from interrupting the MCU multiple times.


Triscend FastChip: Module - Status Register (rcvData)

Component Name: rcvData

s[7:0] 8 D Q 8 <q>

RdSel → rcv_int_clr ✓

INTR0 ✓

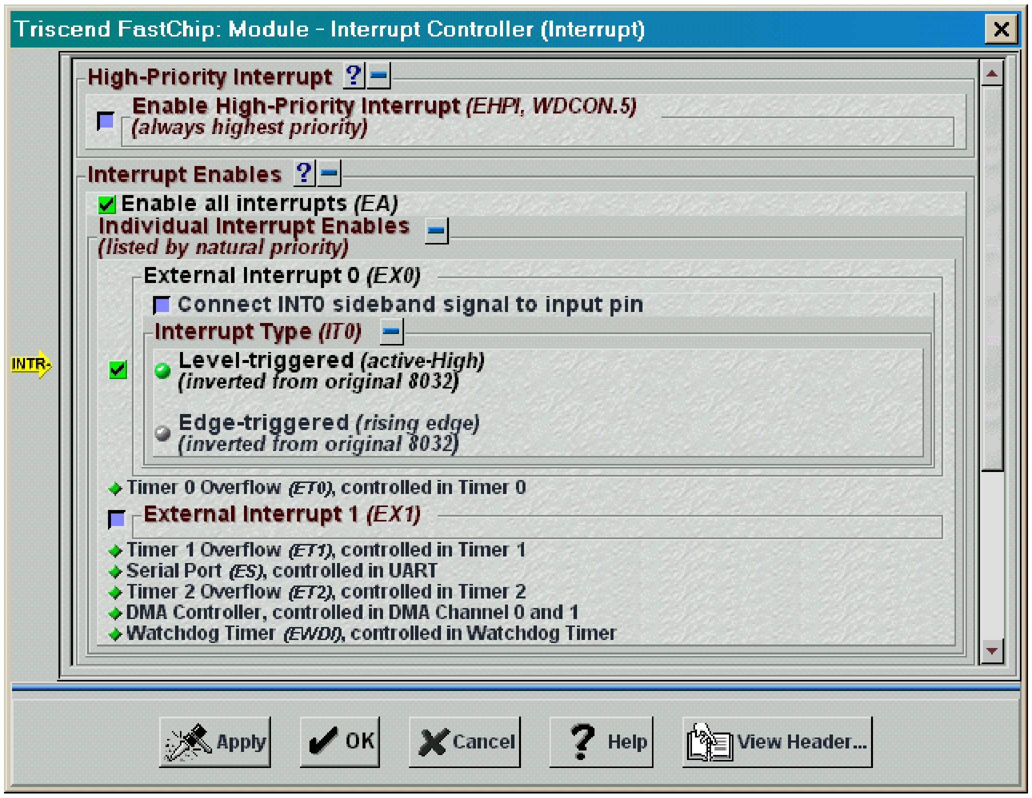
Properties

Status Register Width — Width: 8

Symbolic Address: rcvData

Address Space Response: ● XData, 8032 SFR

The 8032 MCU must be allowed to process interrupts on its **INTR0** input in order for it to respond to keyboard interrupts. Click on the Interrupts icon in the Dedicated Resources area of the FastChip project window. Then check the Enable all interrupts box and the External Interrupt 0 box in the **Interrupt Controller** window. The click on OK.



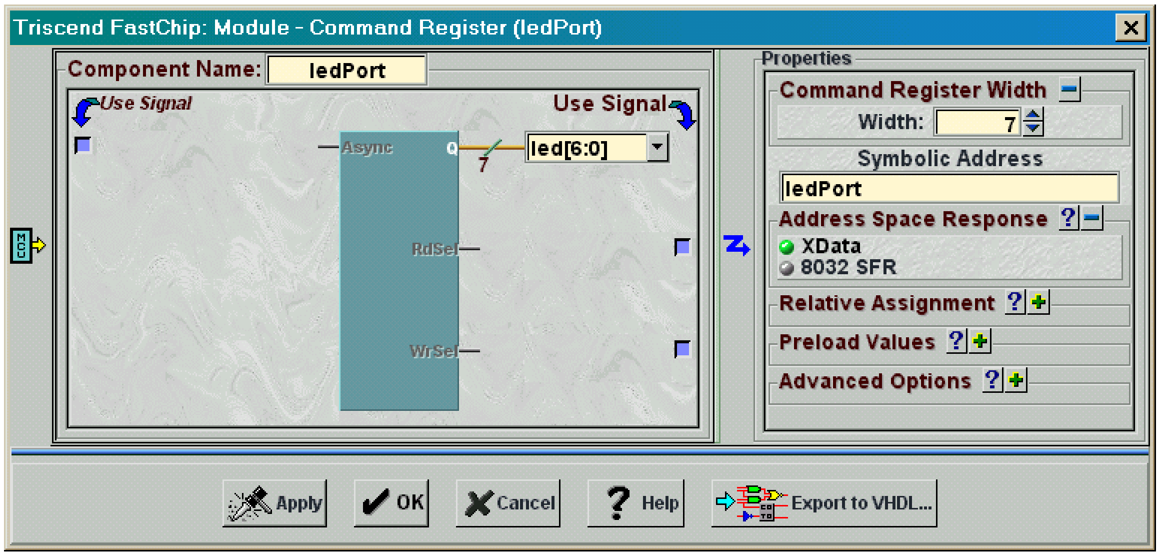
Triscend FastChip: Module - Interrupt Controller (Interrupt)

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

Interrupt Enables
- [✓] Enable all interrupts *(EA)*

Individual Interrupt Enables *(listed by natural priority)*
- [✓] External Interrupt 0 *(EX0)*
  - Connect INT0 sideband signal to input pin
  - Interrupt Type *(IT0)*
    - [✓ Level-triggered *(active-High)* *(inverted from original 8032)*]
    - Edge-triggered *(rising edge)* *(inverted from original 8032)*
- 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

The 8032 MCU needs a means of displaying the scan code it has received on the LED digit. The **ledPort** command register placed in the external data space can be written with a 7-bit pattern that will activate the LED segments. Once again, the FastChip software will assign a logical address for this register.



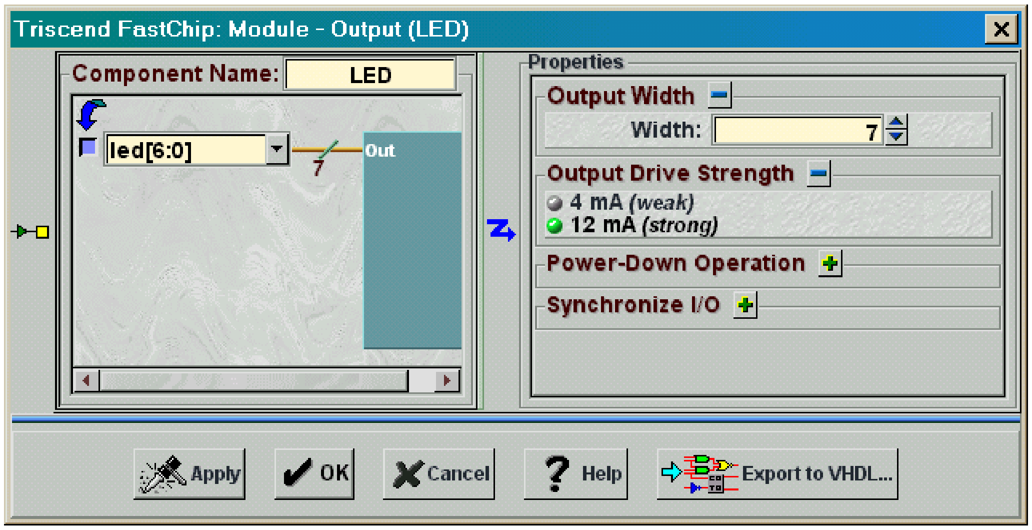
Triscend FastChip: Module - Command Register (ledPort)

Component Name: ledPort

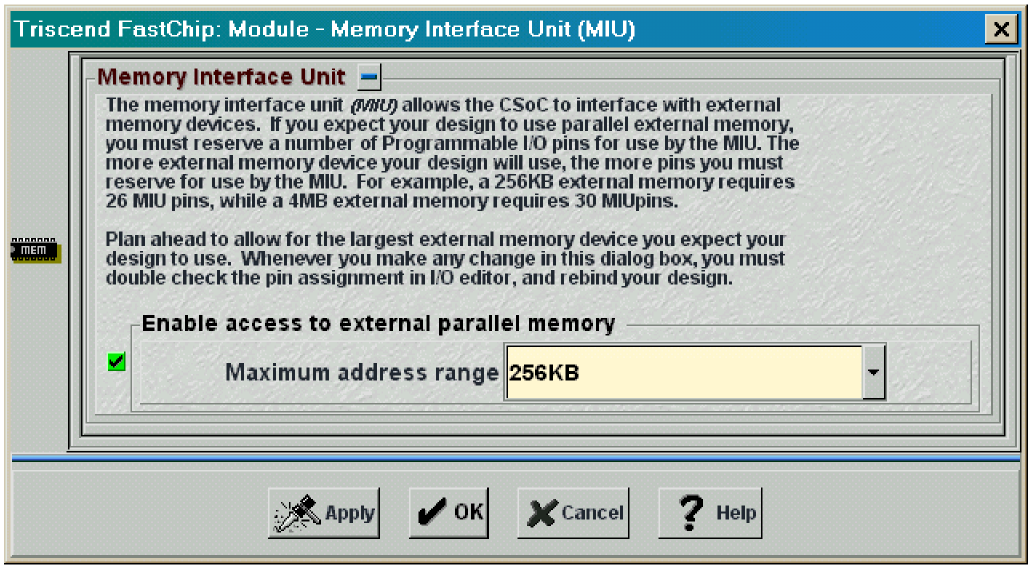
Use Signal: led[6:0]

Properties
- Command Register Width
  - Width: 7
- Symbolic Address: ledPort
- Address Space Response
  - [✓ XData]
  - 8032 SFR
- Relative Assignment
- Preload Values
- Advanced Options

Next, the seven outputs of the **ledPort** register are connected to seven output ports through the seven-bit **led** bus as shown below.



Finally, since the 8032 MCU object 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. Click on OK and move to the next step.


Triscend FastChip: Module - Memory Interface Unit (MIU)
Memory Interface Unit
mem
Enable access to external parallel memory
Maximum address range
256KB
Apply
OK
Cancel
Help

The memory interface unit (MIU) allows the CSoC to interface with external memory devices. If you expect your design to use parallel external memory, you must reserve a number of Programmable I/O pins for use by the MIU. The more external memory device your design will use, the more pins you must reserve for use by the MIU. For example, a 256KB external memory requires 26 MIU pins, while a 4MB external memory requires 30 MIUpins.

Plan ahead to allow for the largest external memory device you expect your design to use. Whenever you make any change in this dialog box, you must double check the pin assignment in I/O editor, and rebind your design.

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

Table 12: Pin assignments and functions for the keyboard interface design.

Signal	Pin	CSoC Board Resource
ps2_clock	47	PS/2 clock input
ps2_data	51	PS/2 data input
LED.0	35	LED digit segment A
LED.1	39	LED digit segment B
LED.2	43	LED digit segment C
LED.3	41	LED digit segment D
LED.4	40	LED digit segment E
LED.5	34	LED digit segment F
LED.6	36	LED digit segment G

Press the Generate icon on the toolbar and FastChip will create the chap31.h header file. Among other things, the header file contains the logical addresses assigned to the **rcvData** status register and the **ledPort** command register (Listing 6). Note that both registers are declared to be of type `CHAR_XDATA` which means they were placed in the external data address space of the 8032 as you specified.

Listing 6: Top of the header file generated by the FastChip software for the keyboard interface.

```
// Generated 4/21/00 10:27 PM By FastChip Version 1999 Build 30

////////////////////////////////////////////////////////////////
//
//  ---------------------------------
//  ------   GENERATED CODE   --------
//  ---------------------------------
//  The code in this header file was generated automatically for your
//  project by Triscend FastChip. Please DO NOT EDIT this header file.
//  It will be overwritten the next time FastChip generates code for
//  your project.
//
////////////////////////////////////////////////////////////////

//======== Required symbol and macro definitions ========

#ifdef PROTOTYPE_ONLY
#  define CHAR_XDATA(name,location) extern volatile unsigned char xdata name;
#  define CHAR_ARRAY_XDATA(name,location,size) extern volatile unsigned char
xdata name[size];
#else
```

```
#  define CHAR_XDATA(name,location) volatile unsigned char xdata name _at_
location;
#  define CHAR_ARRAY_XDATA(name,location,size) volatile unsigned char xdata
name[size] _at_ location;
#endif

//========= BEGIN SOFT MODULE REGISTER DECLARATIONS ======

//-------------------------------- Module rcvData
    CHAR_XDATA (rcvData,0xefff)

//-------------------------------- Module ledPort
    CHAR_XDATA (ledPort,0xeffe)

//========== END SOFT MODULE REGISTER DECLARATIONS =======
```

The next step is to write your application code. Create a keil folder within your Chap31 FastChip project folder. Then start the Keil IDE and add the C code in Listing 7 to the ***chap31*** Keil project. The `main` routine (lines 3–7) just initializes the 8032 MCU and enters an infinite-loop. All the work is actually done in the `displayPs2Data` interrupt subroutine (lines 28–45). This subroutine is called when the keyboard interface generates an INT0 interrupt (interrupt identifier 0). The interrupt subroutine reads the keyboard scan code from the **rcvData** register on line 33 and this also clears the interrupt flag in the keyboard interface. Then the scan codes in the table defined on lines 13–26 are searched. If a matching scan code is found in the table, the subroutine writes the associated LED segment activation pattern to the **ledPort** register. This displays the digit for the key that was pressed. (The table only has the scan codes for the digits 0–9.) If the received scan code can't be found in the table, the subroutine displays an E on the LED digit.

Listing 7: Keyboard interface interrupt-handling code.

```
 1  #include "..\Chap31.h"
 2
 3  main()
 4  {
 5       Chap31_INIT();
 6       while(1);
 7  }
 8
 9
10  #define ERROR 0x79;
11
12  // translate keyboard scan codes to LED segment activations
13  typedef struct{ unsigned char ps2Data, led; } ps2XlateEntry;
14  ps2XlateEntry ps2XlateTbl[] =
15  {
16       { 0x16, 0x06 }, // "1"
17       { 0x1E, 0x5B }, // "2"
18       { 0x26, 0x4F }, // "3"
```

```
     { 0x25, 0x66 }, // "4"
     { 0x2E, 0x6D }, // "5"
     { 0x36, 0x7D }, // "6"
     { 0x3D, 0x07 }, // "7"
     { 0x3E, 0x7F }, // "8"
     { 0x46, 0x6F }, // "9"
     { 0x45, 0x3F }  // "0"
};

static void displayPs2Data() interrupt 0 using 0
{
     unsigned int i;
     unsigned char c;

     c = rcvData;   // get the keyboard scan code

     // search the translation table for the scan code
     for(i=0; i<sizeof(ps2XlateTbl)/sizeof(ps2XlateEntry); i++)
          if(ps2XlateTbl[i].ps2Data == c)
          { // found a matching scan code in the table
               ledPort = ps2XlateTbl[i].led; // display digit
               return;
          }

     // no matching scan code was found, so display "E"
     ledPort = ERROR;
}
```

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.