AN1154
APPLICATION NOTE

8031 / M88 FLASH+PSD Design Tutorial

This tutorial takes you step-by-step through the development cycle of a M88x3Fxx based design, from design entry, to programming the device. The first part of this tutorial shows how a M8813F1x can be used in conjunction with a handful of other ICs to implement an automatic gain control (AGC) design. The tutorial also shows how this design would be implemented using a discrete part solution, and Appendix E summarizes the various benefits of using a M88x3Fxx device over the discrete solution.

The members of the M88 FLASH+PSD family of programmable system devices are Flash-based peripherals for use with embedded microcontrollers (MCUs), and are In-System-Programmable (ISP). These PSDs are designed to interface easily with a variety of 8-bit MCUs, and to provide them with memory, logic, and I/O.

Embedded designs are typically bound by cost, size, and power consumption. The market for products using embedded MCUs is extremely competitive. Time-to-market and quality features-per-dollar define success. In using a M88 FLASH+PSD device, you will reduce your cost, time-to-market, power consumption, board space, design complexity, and chip count.

As you read this document, you will learn how the M88 FLASH+PSD can enhance your MCU, and meet its needs for Flash memory, EEPROM, SRAM, configurable I/O pins, programmable logic (both sequential and combinatorial), decoded address space, address expansion, backup power, code integrity, code security, and ISP. All of these features are to be found in one cost-effective M8813F1x device, and allow the use of a low cost, minimal feature, ROM-less MCU device.

In addition to giving a step-by-step design entry tutorial, this document usefully highlights three aspects of the M88 FLASH+PSD solution:

– ISP using concurrent memory or JTAG
– Micro⇔Cell technology
– The logic simulation capabilities of PSDsilosIII

A typical MCU design with Flash memory consists of:

- an MCU
- the main Flash memory
- and
  - a boot PROM or SRAM to implement an ISP download to the main Flash memory
  - over an UART channel, or some other communication link.

For systems that use SRAM for ISP, the Flash-programming algorithm must first be downloaded to SRAM and then the MCU executes from SRAM during ISP. Any power interruption or system glitches that occur will corrupt the system. Therefore, a boot PROM is a necessity for applications that demand high system reliability. However, a boot PROM adds cost to the system, and is difficult to update once in service. Flash-based PSDs address these concerns and combine all of the elements necessary to enable the MCU to download easily to main Flash memory, and boot memory, while in-system.

The ISP method just described requires MCU participation. The M88 FLASH+PSD also offers another ISP method, which uses a JTAG interface, and requires no MCU participation. This means that a completely blank PSD can be soldered into place, and the entire chip can be programmed, in-system, using ST's JTAG FlashLink cable and PSDsoft development software. This is a powerful new feature of the M88 FLASH+PSD that allows for easy updates in the field.

Typically, adding a peripheral to the MCU memory space involves adding a lot of circuitry to decode the address lines, to latch the data lines, and to handle the bus timing. If an M88 FLASH+PSD device is used, the MCU address, data, and control signals are already routed and processed inside the PSD, and so this hardware overhead is not required.

Micro⇔Cells take advantage of this, and allow the designer to build logic peripherals inside the PSD in an efficient and flexible manner. This tutorial compares a PSD Micro⇔Cell design with an equivalent functional design using an Altera EPM7064S CPLD device, thereby emphasizing the efficiency of the PSD approach.

The M88x3Fxx has 16 output Micro⇔Cells (OMCs) and 24 input Micro⇔Cells (IMCs). Each Micro⇔Cell occupies a memory location in the MCU address space, and is connected to the data bus. The ability to load the flip-flops in the OMCs, and to read them back, is useful in such applications as loadable counters, shift registers, and other system logic. The IMCs can latch external inputs, and be read by the microcontroller. IMCs are also useful when implementing handshake communication logic with an outside source.

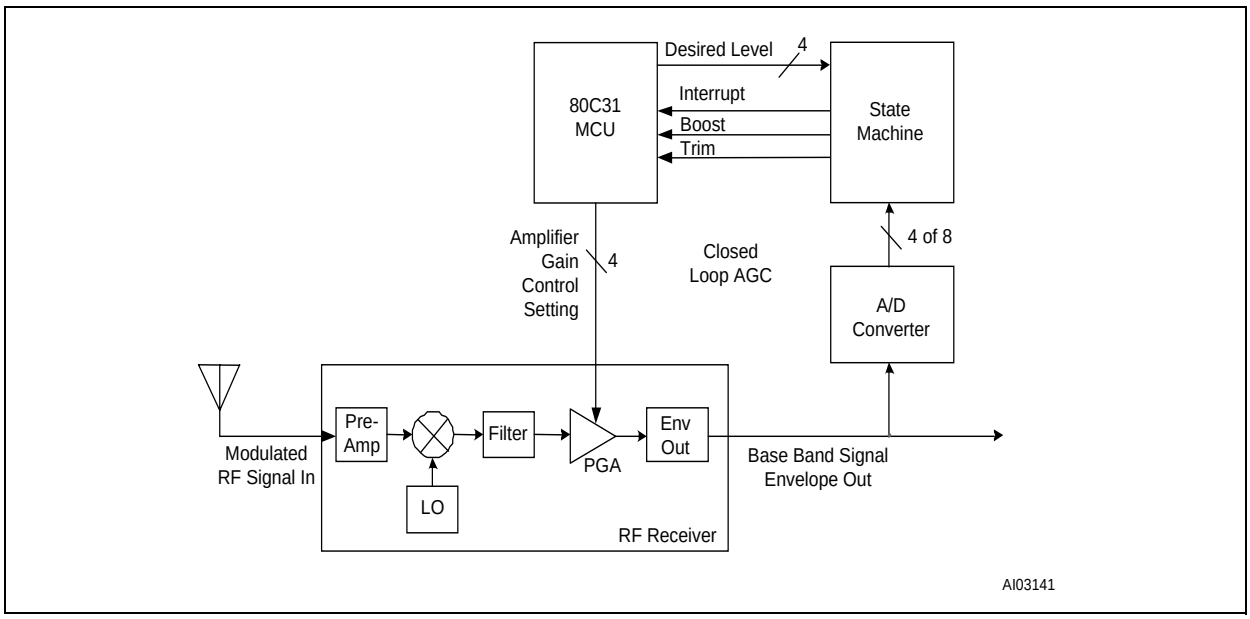
ST provides complete chip-level Verilog-HDL models of all PSD devices for use with the PSDsilosIII simulator. These models can be used in conjunction with a user-defined stimulus file to simulate the functionality of the PSD. PSDsilosIII also comes with a Waveform Editor/Viewer and Watch window (for stepping through the simulation) that are used in conjunction with the stimulus file. Most of the PSD's status and control signals, as well as all the user-defined logic in the CPLD, are available for use with the Waveform Editor/Viewer and the Watch window. Thus, the user can define MCU-level tasks, such as read and write, that can be used as external chip-level stimuli to the PSD, and the results of the stimuli can be viewed using the Waveform Editor/Viewer and Watch window of PSDsilosIII.

A new utility is featured in PSDsoft version 5.X. This utility automatically generates ANSI-C code for the PSD functions, and can be used with the user's choice of MCU cross-compilers.

Design Example

The design that has been chosen, by way of an example, in Figure 1, is a piece of hardware with closed-loop Automatic Gain Control (AGC). This has an analog RF receiver section, which has a Programmable Gain Amplifier (PGA) to control the signal level that is output though an envelope detection circuit. The PGA gain must be adjusted in real-time to keep a constant signal level at the envelope detection output. An Analog-to-Digital Converter (ADC) monitors this output. When the AGC function works properly, a constant signal level is output from the receiver, which can be used by other analog and digital circuitry for signal processing.

Figure 1. Block Diagram of Automatic Gain Control Circuit



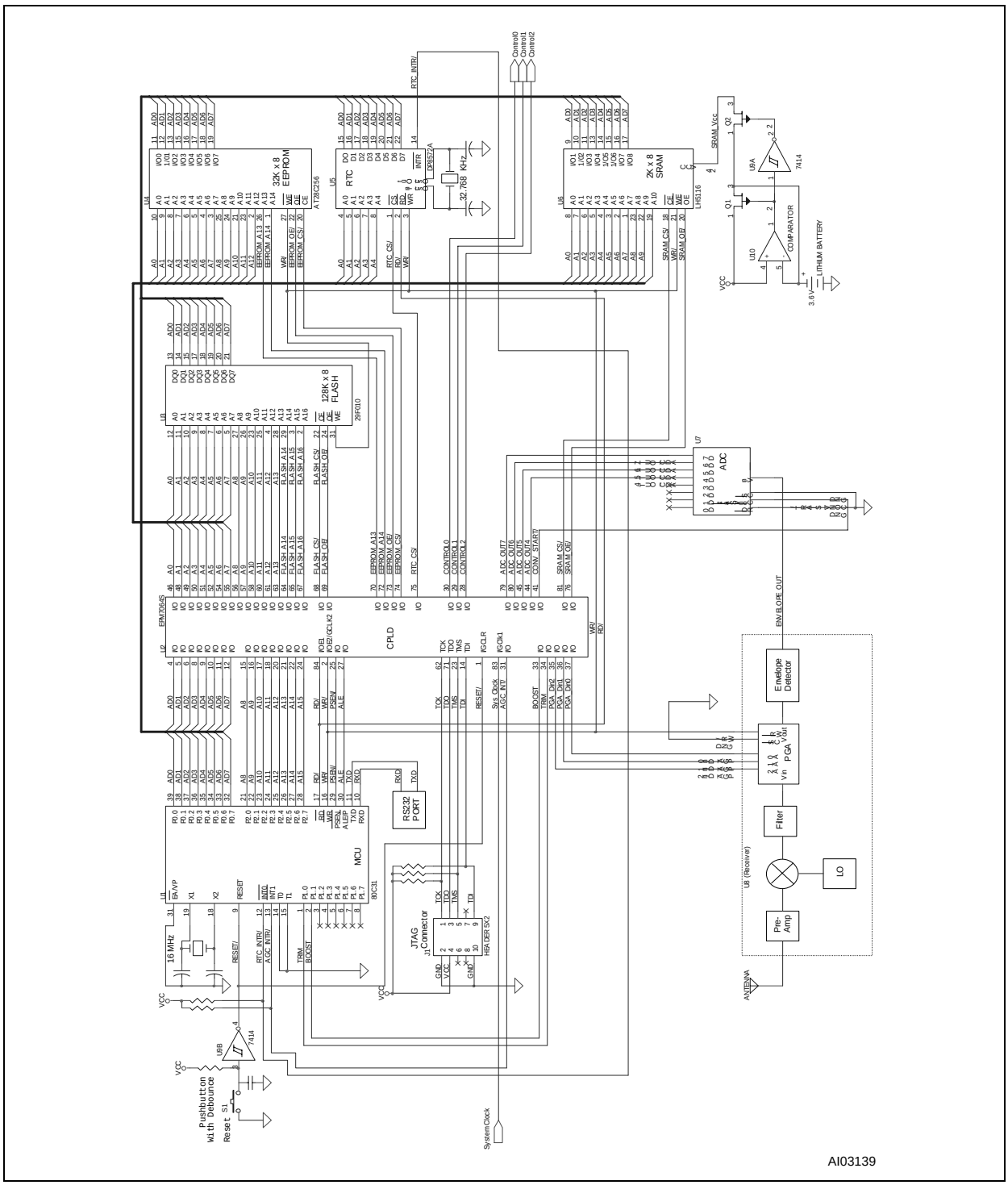
The MCU could be used to perform this real-time gain adjustment, but this would leave it with little execution time for other tasks. It is highly desirable to free up the MCU by off-loading these repetitive tasks to dedicated hardware. The AGC function can be moved into the state machine, implemented using programmable logic, as shown in Figure 1.

In this configuration, the MCU first loads the state machine with a desired signal level, and starts it running, and then gets on with other tasks. Most of the time, the state machine works autonomously, reading the outputs of the ADC and comparing the measured value with the desired value. The state machine only needs to interrupt the MCU when the signal has drifted from the desired level. Along with the Interrupt line, it provides two signals: 'Trim' and 'Boost'. If the signal level from the receiver is too high, the interrupt is accompanied by 'Trim', and the MCU writes the appropriate value to the PGA to decrement the gain. Likewise, if the signal level is too low, the interrupt from the state machine is accompanied by 'Boost'.

This tutorial shows how to implement this AGC function two different ways: a discrete IC solution (using individual IC devices for programmable logic, memory, etc.), as shown in Figure 2, and an integrated PSD solution, as shown in Figure 3. In addition to the AGC function, other features that have been implemented include: the Real-Time-Clock (RTC), In-System Programmability (ISP), and miscellaneous I/O signals.

Please refer to Appendix F for information related to system memory mapping, ISP issues using a UART, and memory paging considerations.

Figure 2. Block Diagram for the Discrete Solution



This is an 80C31 MCU application that has a 128K x 8 Flash memory, a 2K x 8 battery-backed SRAM, a 32K x 8 EEPROM, a real-time clock (RTC), an 8-bit analog-to-digital converter (ADC), a JTAG interface, an EPM7064S ISP CPLD, and an analog receiver circuit (including PGA). In the discrete solution, in Figure 2, four extra IC devices are required. In the M8813F1x solution, in Figure 3, the Flash memory, EEPROM, SRAM, CPLD, and battery backup circuitry are all combined in the M8813F1x device.

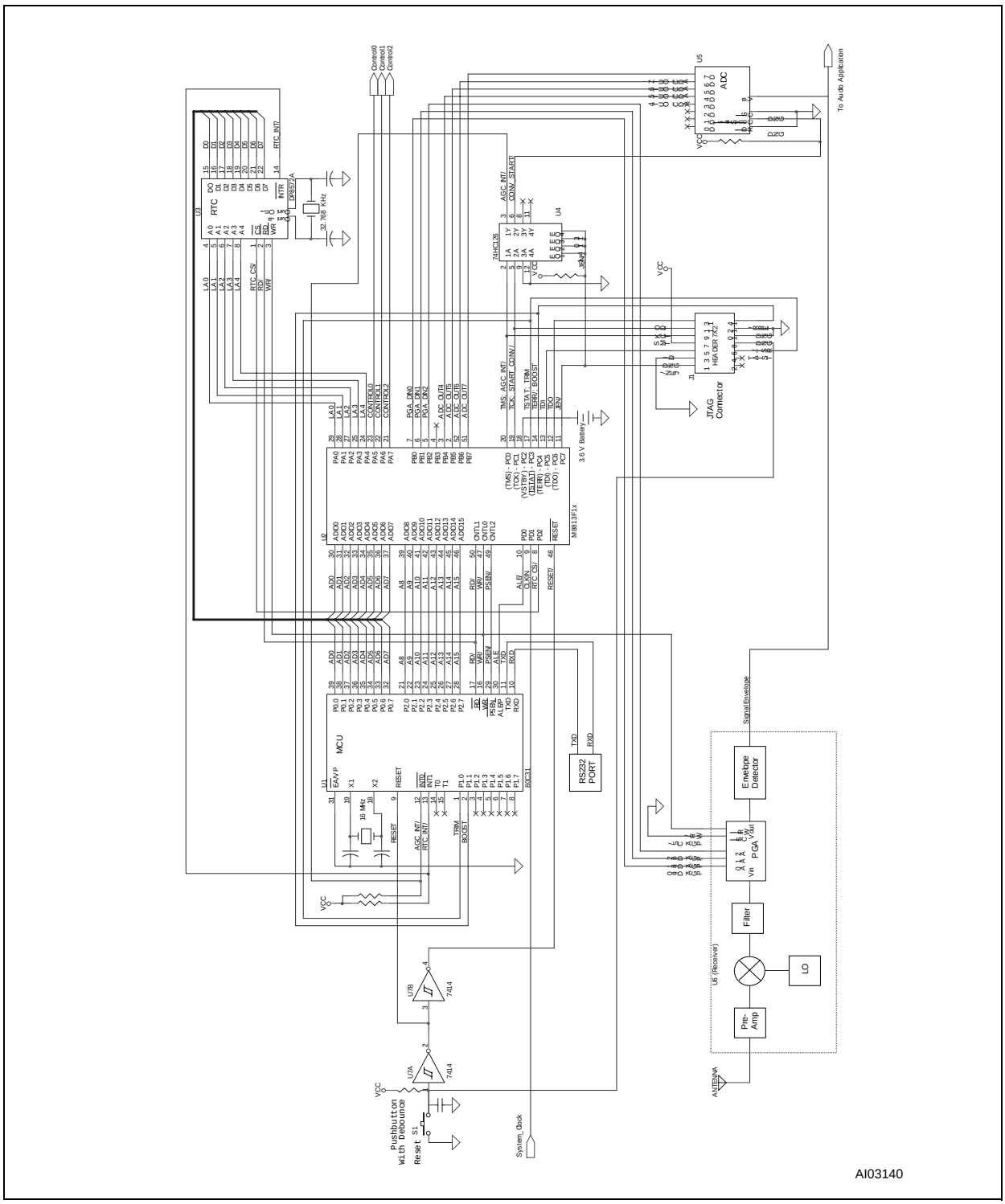
The following notes can be made regarding the discrete solution (Figure 2):

- The 80C31 MCU is using external memory since internal program and data storage is not sufficient. As a result, Port 0 and Port 1 are sacrificed for address and data.
- The EPM7064SLC84-5 CPLD is needed for address decoding, control logic, implementation of a paging/segmentation scheme for the Flash memory and EEPROM, and interfacing to the PGA and ADC. Please refer to Appendix D for the complete design listing for U6.
- The 29F010 Flash memory contains 128K x 8 bits of program memory. Notice that address lines A14-A16 are driven by the CPLD to support the additional address space.
- The A128C256 EEPROM contains 32K x 8 bits of boot memory. This allows concurrent programming of the Flash memory. Address lines A13-A14 are driven by the CPLD to support the additional address space.
- The DP8572A RTC, programmable Real-time Clock, is used to time-stamp various data received by the MCU.
- The LH5116–2K x 8 bit SRAM is configured with battery backup protection.
- The generic 8-bit ADC converts the target signal envelope into a digital value. This IC is controlled by the CPLD.
- The receiver circuit consists of a collection of components, including: a pre-amplifier, a mixer, a local oscillator (LO), a PGA, and an envelope detector circuit. The circuit takes an RF signal from the antenna, as input, and outputs the signal envelop.
- The 7414 inverter with hysteresis (U7B) is used to provide a stable reset signal to the MCU (U1). U7A is part of the battery backup circuit for the SRAM.
- The generic OPAMP comparator is part of the battery backup circuit for the SRAM. When V_{CC} falls below the battery voltage, the circuit switches over to powering the SRAM from the battery.

The integrated PSD design, in Figure 3, can be compared to the discrete design, in Figure 2. The memory (U3, U4, and U6), and the battery backup circuit (U9A and U10) of Figure 2 are all incorporated into the M8813F1x (U2) of Figure 3. Also, all of the functions handled by the CPLD (U2 of Figure 2), are implemented in the PSD's CPLD. The I/O pins are individually configured to match the functions implemented in the original design.

Using JTAG, the entire M8813F1x device can be programmed. Also, the PSD JTAG pins can be multiplexed with other I/O. These JTAG features are beyond the capabilities of the EPM7064S.

Figure 3. Block Diagram for the Integrated PSD Design



Matching the Functions to a M8813F1x

The mapping of the functional areas, of the original design, into the M8813F1x are shown in Table 1.

The 80C31, running at 16 MHz, has a t_{AVIV} (time between address valid and instruction valid) of 207 ns. An M8813F1x-15 (the 150 ns part) was selected to meet the 80C31 access time requirement.

Table 1. Discrete Solution Compared to the M8813F1x Solution

Functional Area	Design Example with Discrete Components	The Matching M8813F1x Function
Memory	128 KByte Flash Memory 32 KByte EEPROM 2 KByte SRAM	Same Same Same
Memory Paging/Segmentation, and Control	Extra logic to drive the address lines, output enables, and chip selects to the Flash and EEPROM	Automatically taken care of internally by the DPLD, PSD page register, PSD VM register, and prioritized memory access.
PLD/Control/Demux	Decoder (EPM7064S) Address latch logic in CPLD Various registers used to hold data or control information to be used by external devices	Use DPLD (Decoding PLD) Port A in latched address mode (A7-A0) Use one Output Micro⇔Cell per bit for each register
I/O	Latched data inputs and outputs on CPLD Combinatorial outputs on CPLD	MCU I/O mode feature same
Supervisory/JTAG	Automatic switch to battery backup Limited JTAG interface with no multiplexing of the JTAG port available, and no JTAG ISP of memory available	Built-in comparator automatically switches to battery power when the system voltage drops below the battery voltage on pin PC2 (V_{STBY}) Utilizes standard JTAG and non-standard extensions (to speed programming); the JTAG port can be multiplexed with other I/O, and the memory and logic within the PSD is ISP via the JTAG port.

THE M88 FLASH+PSD FUNCTIONAL BLOCKS

The M88x3Fxx provides five system-level functional blocks, and allows the user to define and configure these blocks to meet the design specification.

MCU Bus Interface

The MCU Bus Interface adapts the address, data, and control lines of a particular MCU to the PSD. Choices include multiplexed or non-multiplexed address/data bus, and the associated control/handshake signals.

PLDs (Decode for memory and registers, General logic)

The DPLD generates internal chip selects for the M88 FLASH+PSD Flash memory, EEPROM, SRAM, Control registers and I/O Ports, Peripheral I/O mode, and Micro⇔Cells.

The CPLD implements general logical functions, such as state machines, shift registers, counters, and combinatorial logic.

Both PLDs are based on Flash memory technology.

I/O Ports

The M88x3Fxx has four I/O ports: Ports A, B, C, and D. These ports have several modes of operation and may be selected within PSDsoft during design entry, or by MCU firmware at run-time. Modes that are defined by PSDsoft are implemented with Non-Volatile Memory (NVM) configuration bits that cannot be altered unless the device is reprogrammed. The remaining available I/O port operational modes are determined by the MCU writing to PSD control registers. Please see *Application Note AN1x55* for more details.

Memory

The M8813F1x has 128 KBytes of Flash memory, 32 KBytes of EEPROM, and 2 KBytes of battery-backed SRAM. All of these memories may operate concurrently. That is to say that, while one (or more) type of memory is being written to, erased or read, the MCU can still be fetching program code from another.

These memory blocks are placed in the system address space using the PSDsoft development software. The M88 FLASH+PSD also offers some run-time features that can be used to alter the system memory map on the fly, which is useful for memory paging, and ISP.

JTAG ISC interface

The M88 FLASH+PSD family includes a JTAG channel for In-System Programming (ISP). This ISP function is an extension of the typical JTAG boundary-scan function. It is an implementation of the JTAG-ISC (In-System Configuration) specification that is becoming an industry standard. The entire PSD device may be configured and programmed while soldered to the end product. The PSD can be completely blank before programming because the JTAG interface needs no assistance from the MCU. ST has enhanced the standard four-wire IEEE 1149.1 JTAG interface by making available two additional handshake lines to speed the programming.

The use of the JTAG interface, and the two additional handshake lines, are defined using PSDsoft. Also, the MCU has some control over the JTAG interface at run-time.

PSDSOFT DEVELOPMENT TOOLS

PSDsoft is ST's integrated system development software tool, which runs on a PC in the Windows 95 and Windows NT environments. PSDsoft supports the configuration of the functional blocks, as described in the previous sections. PSDsoft consists of the following major modules:

- PSDabel
- PSD Configuration
- PSD Fitter
- PSD Simulator
- PSD Programmer
- C Code Generator

The PSDsoft design process for a PSD devices follows the flow shown in Figure 4.

PSDabel

PSDabel has MINC's HDL ABEL engine at its core (formerly DATA I/O ABEL). The PSDabel environment provides an editor to create/edit an .abl file that can be used to define chip select logic, general-purpose logic, and PSD configuration parameters. Template files are provided for many MCU and PSD combinations. When the ABL file is compiled, logic is synthesized, and files are created and passed on to the PSDsoft fitting utility.

PSD Configuration

This utility is used to specify the PSD MCU bus interface type, special I/O pin assignments, and particular internal PSD functions. The output of this module is a .glc configuration file, which is also used by the PSDsoft Fitter.

PSD Fitter

PSD Fitter has two main functions: the Fitter and the Address Translator. The Fitter accepts input from PSDabel and PSD Configuration, synthesizes this user logic and configuration, and fits the design to the PSD silicon. The Address Translator process allows the user to map the MCU firmware from a cross-compiler (in Intel HEX or S-Record format) into the NVM blocks within the PSD. As a result, the MCU firmware is merged with the logic and configuration definition of the PSD. The output of the Address Translator is an .obj file that can be used by a programmer to program the PSD device. This .obj file can also be used to program an M88 FLASH+PSD using the JTAG FlashLink cable. The .obj file includes chip configuration information, the PLD fuse-map, and MCU firmware.

PSD Simulator

ST's version of SIMUCAD's SILOSIII simulation software provides functional chip-level simulation for PSD devices. PSDsoft automatically creates files for input to the simulator. These files convey relevant design information to the simulator. As a result, the user only has to create a stimulus file since all of the signals and node names are taken from the .abl file.

PSD Programmer

PSD Programmer is the interface to the ST MagicProIII®, PSDpro, PEP300, and FlashLink programming devices. It accepts the .obj file as input, allows viewing and editing of the .obj file, and programs the PSD device.

C Code Generation

This is a new feature of PSDsoft that automatically generates C code functions and headers for controlling Flash PSD devices. These functions and headers are ANSI-C compatible. The generated files (.c and .h)

may be edited to suit the particular application, then compiled and linked with the rest of the code. Afterwards, the linker output of the cross-compiler (usually in Intel HEX or Motorola S-record format) is merged with the configuration file of the PSD device in the Address Translate utility of PSDsoft.

The functions and headers provided by PSDsoft, cover PSD operations such as:

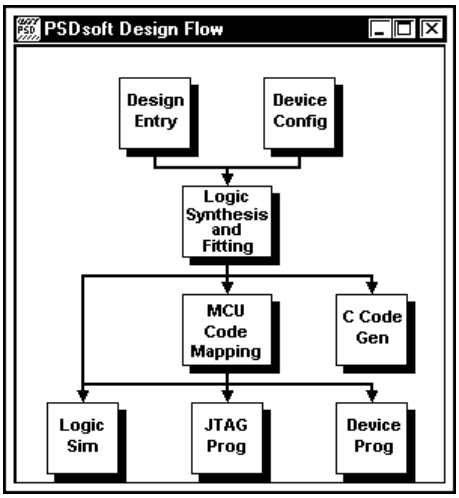
- Flash memory program and erase algorithms
- EEPROM program algorithms
- I/O control and definition
- memory management
- power management.

DESIGN FLOW

This section describes the design flow of a project, from the initial entering of the design, in PSDabel, to the programming of the device, and its simulation.

Figure 4 shows the PSDsoft Design Flow utility. This is the first window to appear after you invoke PSDsoft. By double clicking on each box, the associated process is initiated. While this is a convenient method to navigate through the steps, this tutorial shows how to step through the process using menus and tool-bars since this approach is less obvious. The section, starting on the page after next, takes you step-by-step through a tutorial design.

Figure 4. PSDsoft Design Flow



PSDsoft Program Flow

The high level steps for a PSD design are as follows:

1. Create or open a project, after entry into PSDsoft. If you are creating a new project, specify the project name, the directory path, device family, part number, and provide a small description of the design if desired.
2. Select a design template (*project*.abl file), and modify this template to fit your design.
3. Use PSDabel to edit, compile, and optimize the *project*.abl file. Perform ABEL simulation if desired. To do so, you will need to create the necessary test vectors, and to place them at the end of the PSDabel file. A successful PSDabel compile operation generates an optimized PLA file (*project*.tt2) for the Fitter.
4. Configure the PSD device using PSD Configuration. This generates the *project*.glc file for the Fitter.
5. Fit the design using PSD Fitter. The Fitter's input files are obtained from PSDabel and PSD Configuration. The Fitter generates the *project*.fob file that is passed on to the Address Translator. The Fitter also generates two fuse-map files, *project*.afu and *project*.pfu for the Simulator. After a successful fit, it is possible to skip to step 8 (simulation), if desired, since PSDsilosIII can be used before or after MCU firmware is merged with the PSD configuration.
6. Generate the C code, if desired. Edit this C code to suit your particular application. Then, compile and link it with your other application C code. Your cross-compiler will output an Intel HEX or Motorola S-record file containing the firmware.
7. Perform the address translation. The Address Translator combines the MCU firmware file and the *project*.fob file into a *project*.obj file. This *project*.obj file includes the MCU firmware, the fuse-map, and the configuration bits.
8. Verify the design using PSD Simulator. Chip level simulation is based on the user's Verilog stimulus file (*project*.stl) and fuse-map files from the Fitter. You must create the *project*.stl file. However, PSDsoft creates files to be used with the simulator that allow you to use the same names that appear in your *project*.abl file, and various reserved names.
9. Use PSDsoft to download the *project*.obj file to the MagicProIII®, PSDpro, or FlashLink JTAG programmer to program the device. A compatible third party programmer can also be used. Contact ST or a representative near you for a list of compatible programmers.

M88 FLASH+PSD TUTORIAL EXAMPLE

This section uses the tutorial design example to illustrate the steps involved in implementing the functionality discussed earlier. The files required, which were generated for the tutorial design, can be found in the \PSDSOFT\TUTORIAL\TUTOR8XX\TUTOR directory.

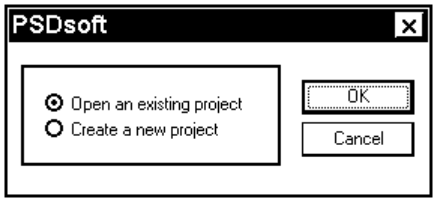
At this point, you may wish to start the PSDsoft program, so that you can follow along with the tutorial example.

Managing the Project

Each new project may have its own working directory, in which all the files generated by PSDsoft can reside. Once you specify the new project name, PSDsoft passes the working directory and pertinent information to other functional modules. In the following sections, you will be guided through a full sample design process, and key windows are displayed to help you follow the example.

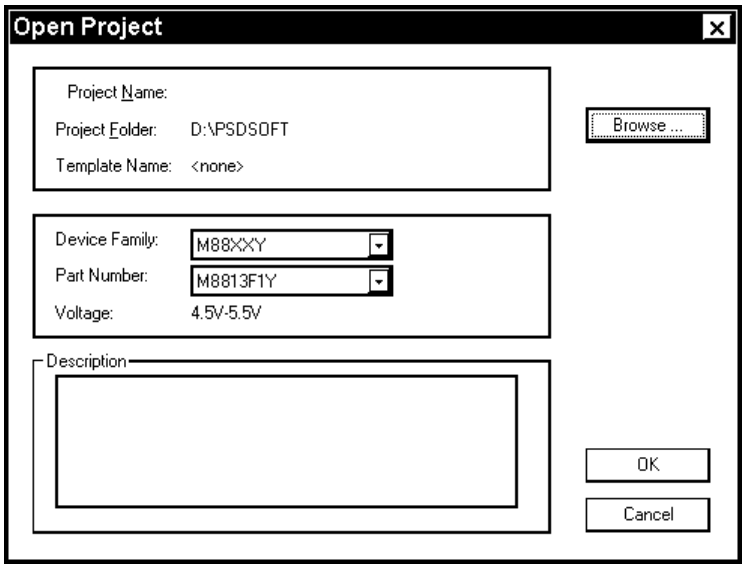
1. Start PSDsoft. The PSDsoft dialog box pops up (Figure 5) enquiring whether you want to open an existing project, or to create a new one. Select “Open an existing project”, and click **OK**.

Figure 5. PSDsoft Dialog Box



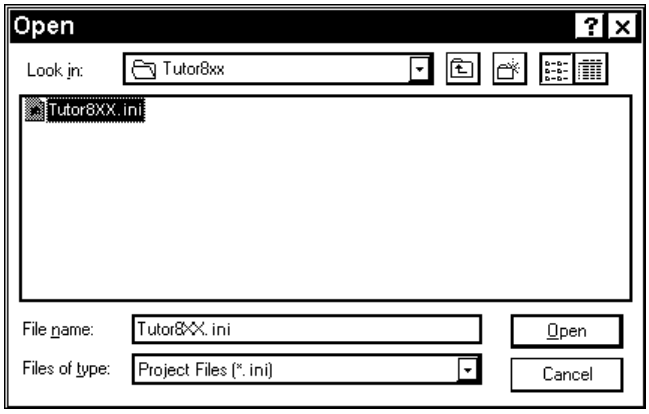
(If you leave PSDsoft without closing the project you were working on, it will automatically reopen when you next run PSDsoft. If the PSDsoft dialog box does not appear, pull down the **Project** menu and select **Open Project**). Either way, the “Open Project” dialog box appears, as shown in Figure 6.

Figure 6. Open Project Dialog Box



2. Click on the **Browse** button, which brings up the “Open” dialog box, as shown in Figure 7. Go to the \PSDSOFT\TUTORIAL\TUTOR8XX\TUTOR directory, select the *tutor8XX.ini* file, and click on the **Open** button (this closes the “Open” dialog box).

Figure 7. Open Project – Open Dialog Box

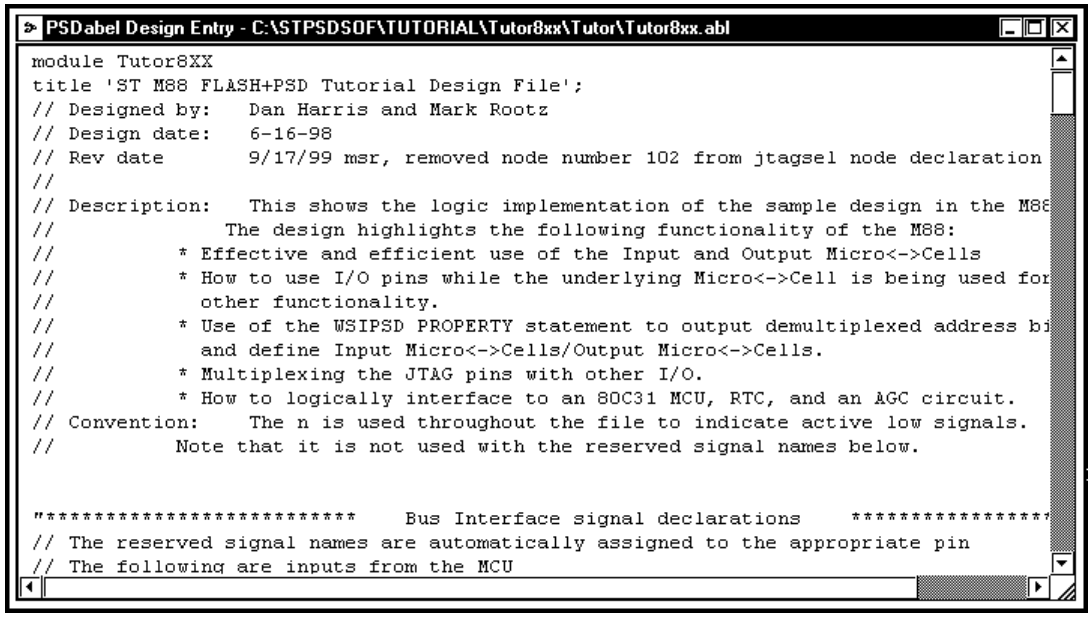


3. Click on the **OK** button (this closes the “Open Project” dialog box).

The PSDabel File

For detailed information on PSDabel, and how it relates to the M88 FLASH+PSD, please read the comments in the file *tutor8XX.abl* in Appendix A. Also, refer to ST's *Application Note AN1171* and the *PSDsoft PSDabel-HDL Reference Manual*. For more information on the system memory map for this tutorial design, see Appendix F. To open the *tutor8XX.abl* design file, as shown in Figure 8, click on **View->Design File**, and click the "Design Entry" button on the tool bar, or click on "Design Entry" in the design flow window.

Figure 8. Design File



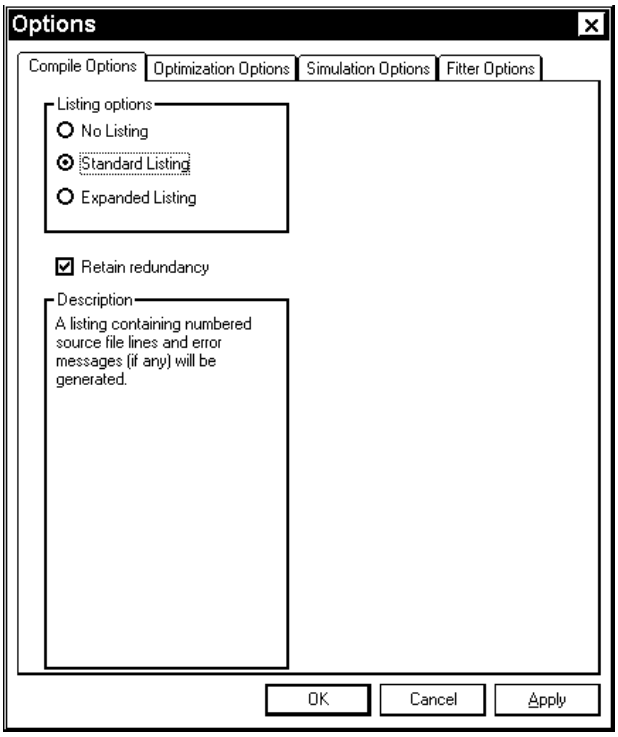
AI03146

Compiling the Tutor Design

To compile the *tutor8XX.abl* file, take the following steps:

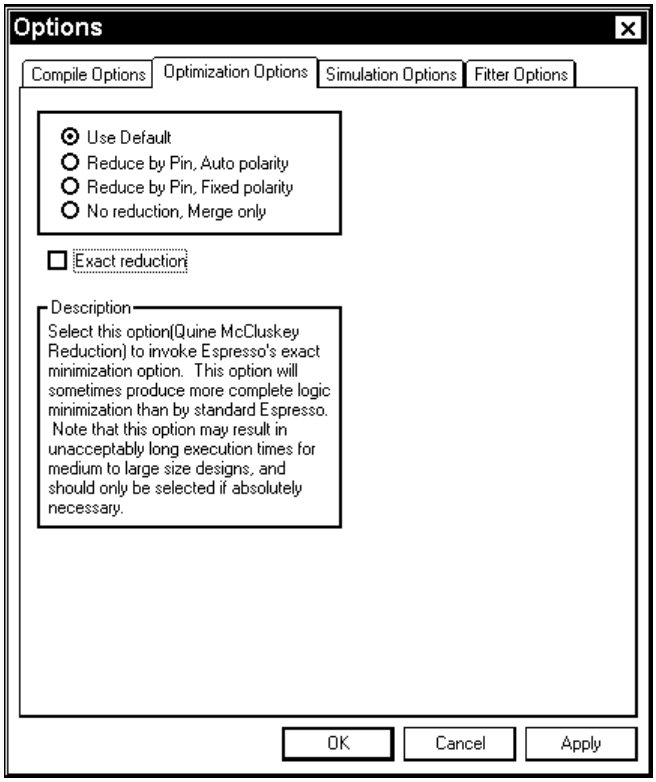
1. Click on the **Options** menu. This brings up the “Options” dialog box, with the “Compile Options” tab selected.
2. Click on each of the options, and read the description in the “Description” box to get a feel for what each option will do. Then set up the options, as shown in Figure 9, with the **Standard Listing** selected under “Listing options” and the **Retain redundancy** box checked. For a better description of the various options available, please refer to the *PSDabel–HDL Reference Manual*.

Figure 9. ABEL Compiler Options



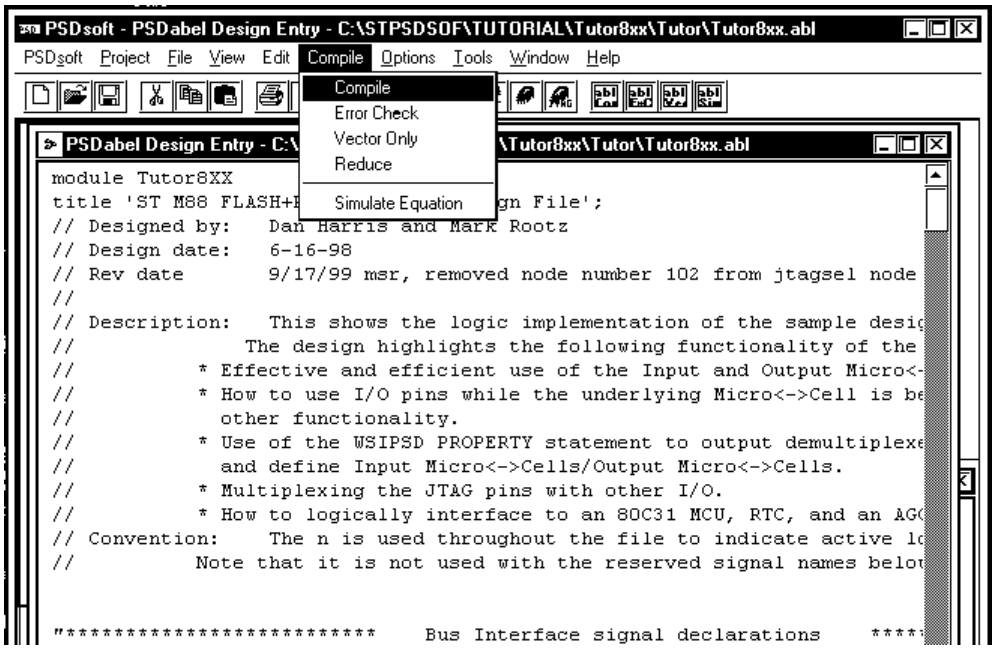
3. Select the “Optimization Options” tab, and set up the options, as shown in Figure 10. **Use Default** should be the only item selected.

Figure 10. ABEL Compiler Options – Optimization Options



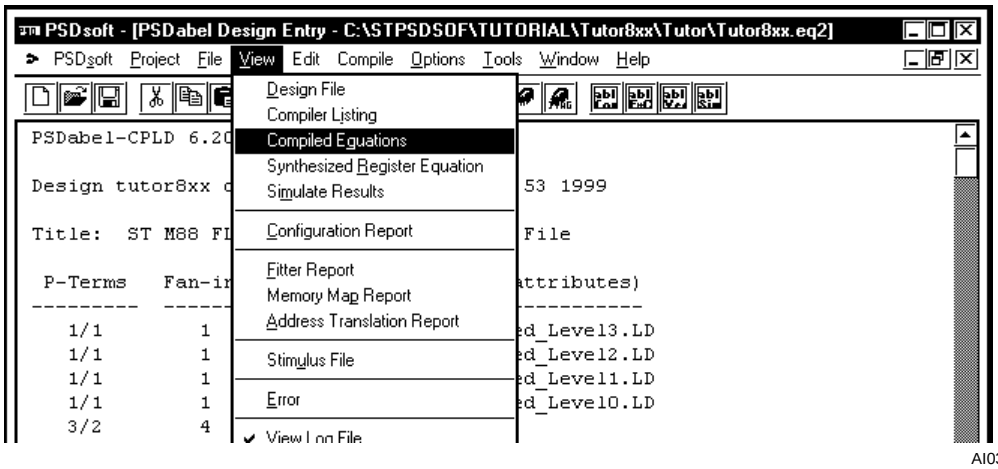
4. Click on the **OK** button when you have finished setting up the options.
5. Click on **Compile->Compile**, as shown in Figure 11. Or, click on the “Compile” button on the tool bar.

Figure 11. Compile->Compile



6. The PSDabel compiler generates an error file—*tutor8XX.ERR* (even if no errors are present), and writes to the log file. The compiler also generates a PLA output file, *tutor8XX.tt2*, which is used by PSDsoft for fitting, and is optimized, based on the reduction algorithm specified in the “Optimization Options” under the **Options** menu.
7. After compilation, you can display the optimized PLD logic equations that will be used by the Fitter by pulling down the **VIEW** menu and selecting the **Compiled Equations**, as shown in Figure 12. This opens the *tutor8XX.eq2* file.

Figure 12. View->Compiled Equations

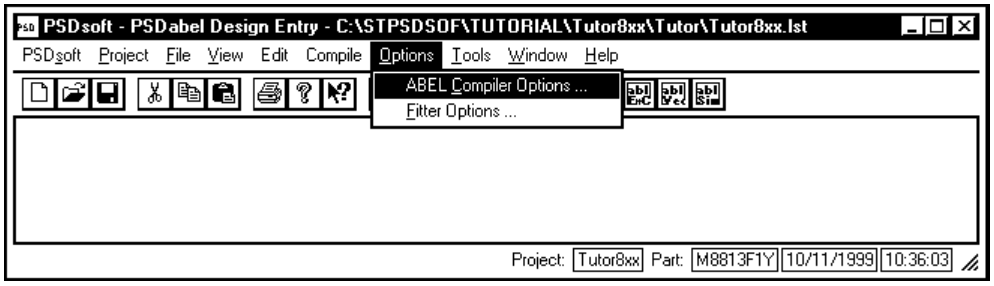


Simulating Your Design Using ABEL Simulation

You can do a very simplistic functional simulation of the blocks that make up the PLD using the simulator that is included with PSDabel. It is important to note that only the functions that are generated within the .abl file can be simulated using test vectors at the end of the file. For chip-level functional simulation, you must have the version of PSDsoft that includes the PSDsilosIII simulation software. To use the simulator that comes with PSDabel, take the following steps:

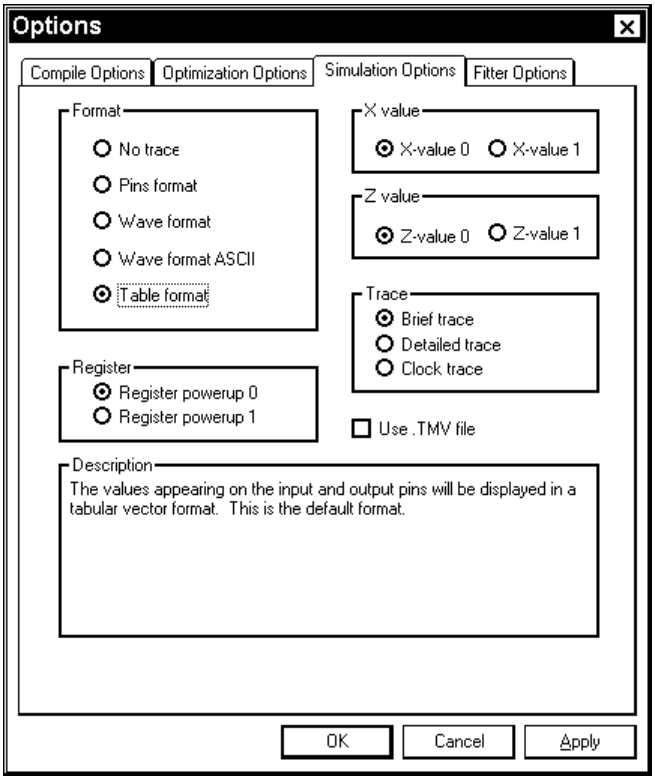
1. Click on the **Options** menu, as shown in Figure 13, which brings up the “Options” dialog box.

Figure 13. Options Dialog Box



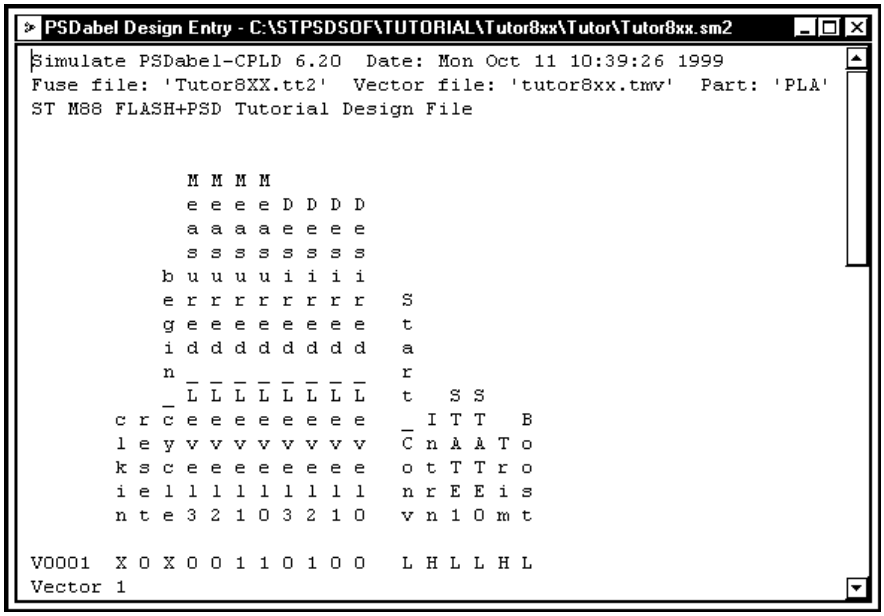
2. Click on the “Simulator Options” tab, and set up the window, as shown in Figure 14: under “Format”, choose **Table format**. Ensure that the **X-value 0** and **Z-value 0** are selected in their respective boxes, and that **Brief trace** is selected in the “Trace” box. For the “Register” box, select the **Register power-up 0**, and make sure that the “Use .TMV file” box is not checked.
3. Click on the **OK** button to save your changes.

Figure 14. ABEL Compiler – Simulation Options



4. If you select **Simulation Results** in the **View** menu, PSDabel will automatically start the simulation process, and display the simulation results based on the logic equations and test vectors in the .abl file, as shown in Figure 15.

Figure 15. Simulation Results



AI03153

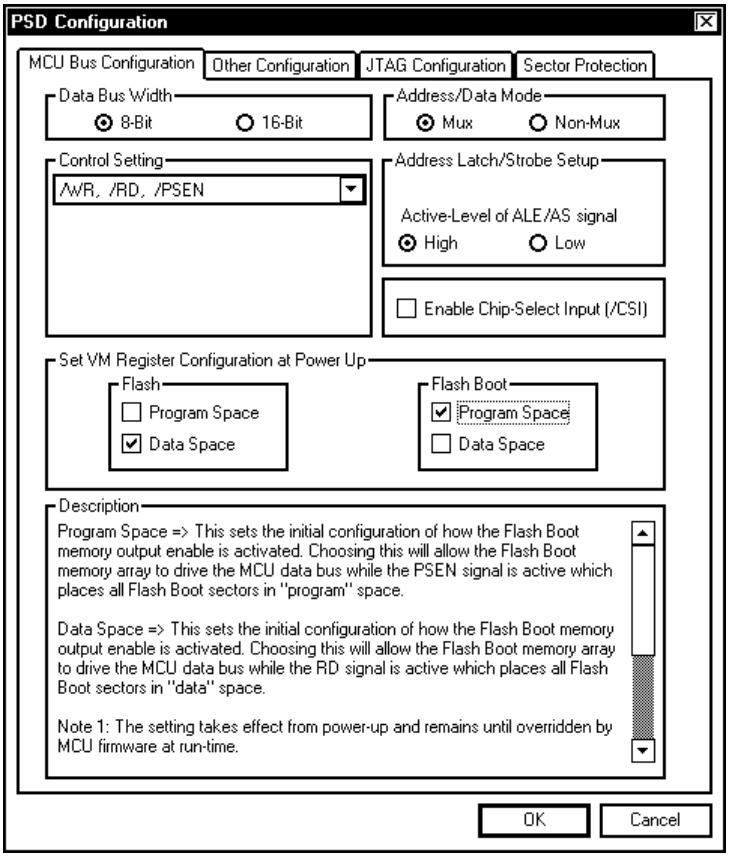
PSDsoft Configuration

The M88 FLASH+PSD has a programmable MCU bus interface, and is able to interface directly to many microcontrollers. Using PSD Configuration, you can specify how to interface to the MCU you have chosen for your design. You can also configure functions specific to the PSD device you are using. This tutorial design is based on the Intel 80C31 microcontroller, which has an 8-bit multiplexed bus with $\overline{RD}$, $\overline{WR}$, and $\overline{PSEN}$ as the control signals, and an active-high level address latch enable (ALE).

To perform the configuration, take the following steps:

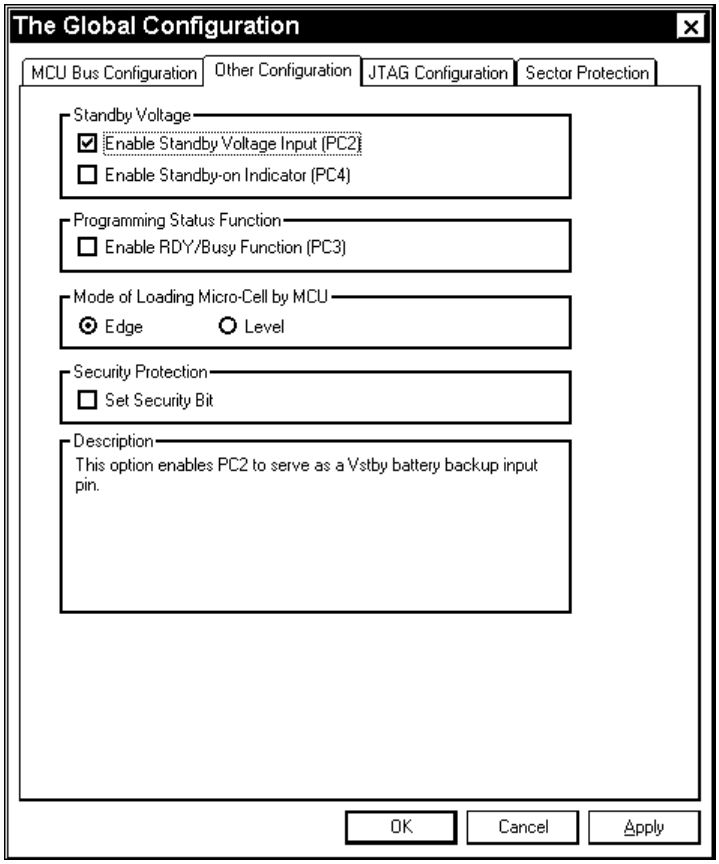
1. Pull down the **PSDsoft** menu in the main PSDsoft window and choose **PSD Configuration**, click the “Configuration” button or click on “Device Config” in the “PSDsoft Design Flow” window. A dialog box opens entitled “The Global Configuration”, as shown in Figure 16. Make sure the “MCU Bus Configuration” tab is selected.
2. Set up the global configuration as shown in Figure 16. Ensure **8-Bit** is selected under “Data Bus Width”, **Mux** is selected under “Address/Data Mode”, **High** is selected under “Address Latch/Strobe Setup”, the “Enable Chip-Select Input ($\overline{CSI}$)” box is not checked, the **$\overline{WR}$, $\overline{RD}$, $\overline{PSEN}$** is selected under “Control Setting”, **Data Space** is selected under “Flash”, and **Program Space** is checked under “EEPROM”. This arrangement for program and data space allows the MCU to boot from EEPROM in the program space, and to download to Flash memory in the data space, if needed. Afterwards, the MCU can override this arrangement if, for example, the Flash memory needs to become part of the program space. This can be done by the MCU writing the VM register.

Figure 16. MCU Bus Configuration



3. Click on the “Other Configuration” tab, as shown in Figure 17, and ensure that the **Enable Standby Voltage Input (PC2)** box is checked under “Standby Voltage”, **Edge** is selected under “Mode of Loading Micro⇔Cell by MCU”, and all other boxes are unchecked.

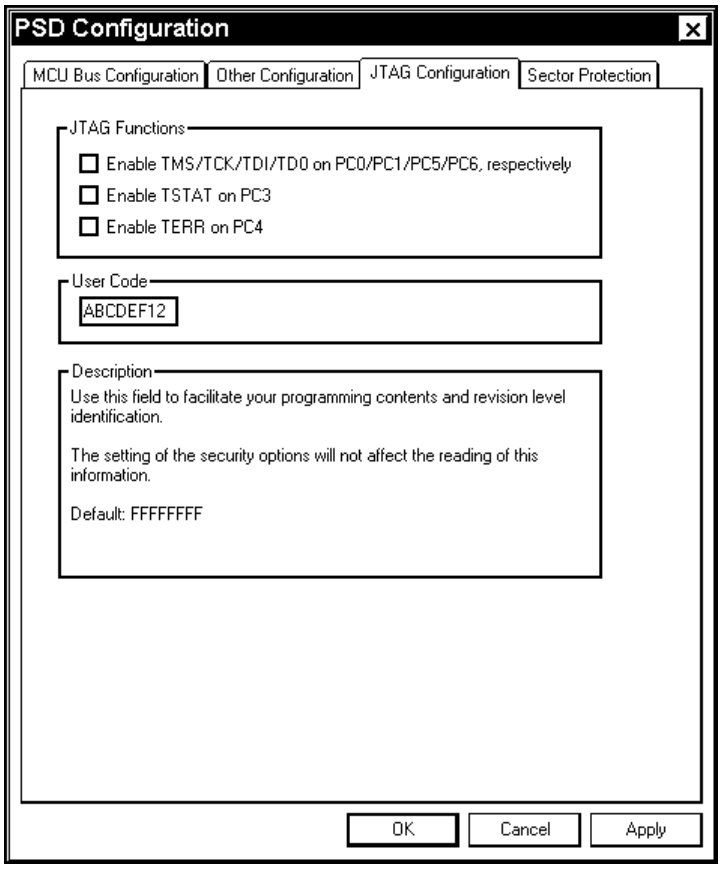
Figure 17. Other Configuration



4. Click on the “JTAG Configuration” tab, as shown in Figure 18, and ensure that none of the boxes are checked (because checking the boxes would enable the JTAG port to be operational 100% of the time). Since, in this tutorial, we are multiplexing the JTAG pins with other signal functions, it is desired that JTAG functions only be operational when the $\overline{\text{JEN}}$ signal is active (see the Figure 3 schematic).

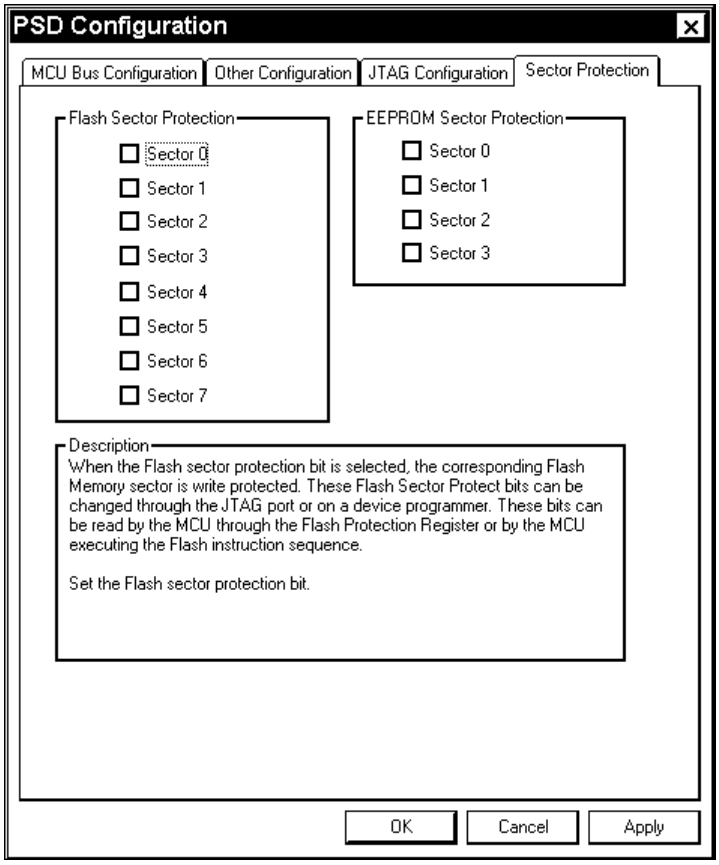
   Enter the value ‘ABCDEF12’ in the “User Code” box below. This value will be programmed into your PSD device. The User Code can be any value you wish (e.g. to identify end product software revisions, serial numbers, etc.). Up to eight hexadecimal characters may be entered.

Figure 18. JTAG Configuration



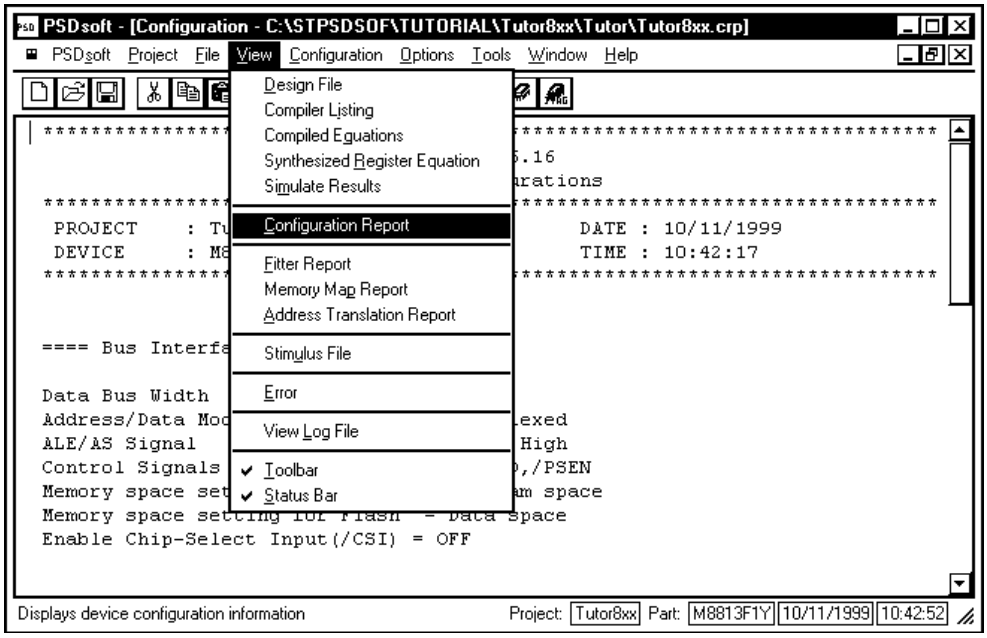
5. Click on the “Sector Protection” tab, as shown in Figure 19, and ensure that none of the boxes are checked. The appropriate sector box should only be checked if it is desired that the selected sector be write protected. These bits can be changed, later, through the JTAG port or the device programmer.

Figure 19. Sector Protection



6. When you are finished with the global configuration settings, click on the **OK** button. This saves the configuration. The M88 FLASH+PSD configuration is now completed. If you ever wish to view the configuration file, first ensure you are in Configuration Mode (see step 1 of this section). Next, pull down the **View** menu and select **Configuration Report**, as shown in Figure 20, and then select **File->Print**.

Figure 20. Configuration Report



PSD Fitter: Fitting and Address Translation

PSD Fitter consists of the Fitter and the Address Translator. The Fitter accepts input from PSDabel and PSD Configuration, synthesizes the user logic and configuration, and fits the design to the M88 FLASH+PSD silicon. The Address Translator process allows the user to map the MCU firmware from a cross-compiler (in Intel HEX or S-Record format) into the NVM blocks within the PSD. As a result, the MCU firmware is merged with the logic and configuration definition of the PSD. The output of the Address Translator is the *tutor8XX.obj* file.

Fitting the Design

The input files to the Fitter are:

- *tutor8XX.tt2*–PLA file generated by PSDabel.
- *tutor8XX.glc*–M88 FLASH+PSD configuration file generated by PSD Configuration.

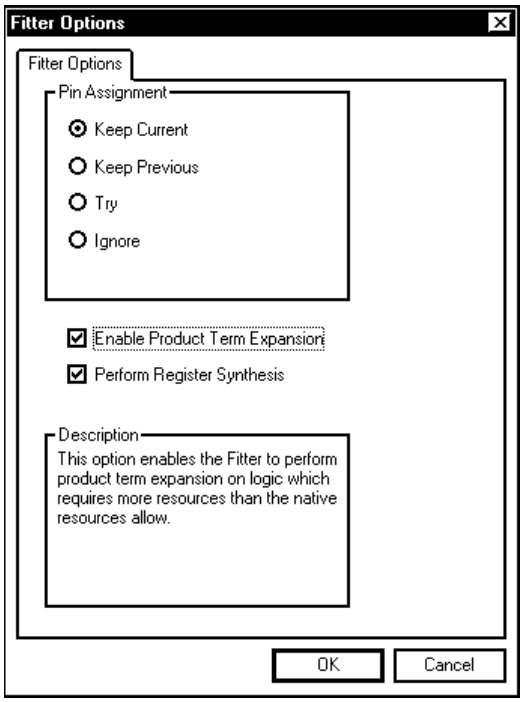
The output files generated by the Fitter are:

- *tutor8XX.fob*–PLD fuse-map and M88 FLASH+PSD configuration file.
- *tutor8XX.afu*–Generated for use by the Simulator.
- *tutor8XX.pfu*–Generated for use by the Simulator.
- *tutor8XX.obj*–Object file (PLD and Configuration portion only).
- *tutor8XX.frp*–Fitter report file.

To Fit a Design:

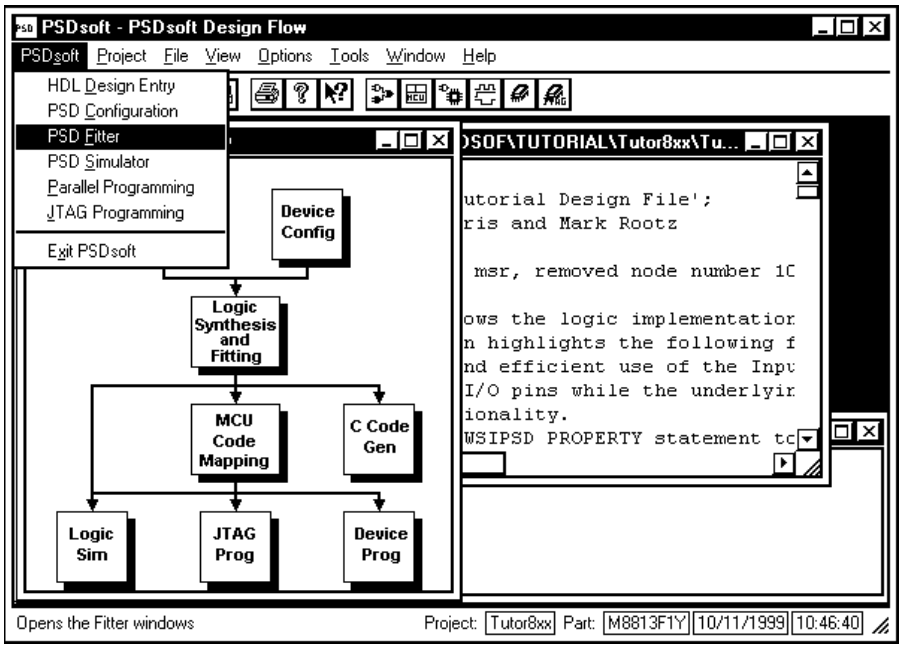
1. Click on the **Options** Menu, and select the “Fitter Options” tab to specify one of the four fitting options, as shown in Figure 21. For the tutorial, choose **Keep Current** under “Pin Assignment”, and ensure that the “Enable Product Term Expansion” and “Perform Register Synthesis” boxes are checked.

Figure 21. Fitter Options



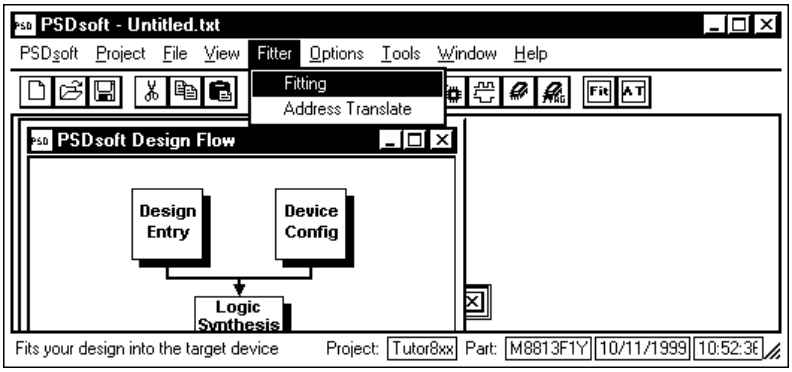
2. Click **OK** to save the Fitter options. Then, pull down the **PSDsoft** menu in the PSDsoft window and choose **PSD Fitter**, as shown in Figure 22, or click on the “Logic Synthesis and Fitting” box in the “PSDsoft Design Flow” window.

Figure 22. PSDsoft->PSD Fitter



3. Pull down the **Fitter** menu and choose **Fitting**, as shown in Figure 23, or click the **Fit** button on the tool bar. If you had clicked on the “Logic Synthesis and Fitting” box in the design flow, the Fitter would run automatically, and this step would not be necessary.

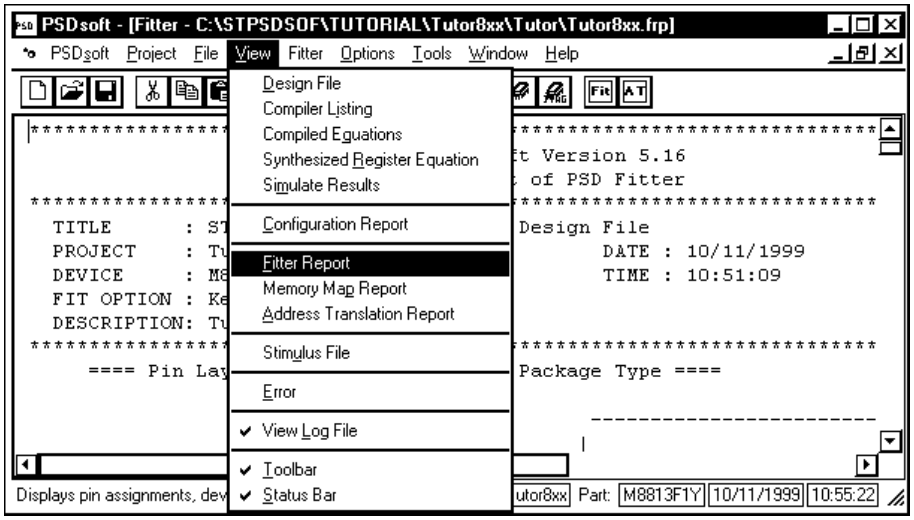
Figure 23. Fitter->Fitting



4. The Fitter appends to two files: the log file (*PSDsoft.log*) and the error file (*tutor8XX.ERR*). Check the log file for any possible errors. If there are no errors present (there should not be if you did not modify the *tutor8XX.abl* file), skip to Step 7.
5. If the fitting is not successful, you may have to view the *tutor8XX.eq2* file in PSDabel to see which logic function caused the fitting problem, and to modify the *tutor8XX.abl* file accordingly. To view the optimized equation file (*tutor8XX.eq2*), see step 7 in the section entitled “Compiling the Tutor Design” on page 15.

6. Re-compile the modified *tutor8XX.abl* file. Repeat Steps 3 to 6 until a successful fit has been found. Re-enter the Fitter program, and proceed to Step 7.
7. Examine the Fitter Report File by pulling down the **VIEW** menu, as shown in Figure 24. The report file shows the results of the fitting process, and the pin assignment for the M88x3Fxx. If you want a fitting other than the one generated, return to the *tutor8XX.abl* file to change the signal and pin assignments as appropriate.

Figure 24. View->Fitter Report



AI03162

Generating C code

PSDsoft can generate ANSI C code functions and headers for controlling the M88 FLASH+PSD. This is an optional step. However, it will save you time by implementing low-level PSD driver function and header files.

The functions and headers are ANSI-C compatible. The .c and .h files that are generated should be edited to suit your application, then compiled and linked with the rest of your application code, using an MCU cross-compiler and linker.

The functions and headers that can be generated by PSDsoft include the following operations:

- Flash memory program and erase algorithms
- EEPROM program algorithms
- I/O control and definition
- memory and power management.

Although C code generation can be performed anytime after a project is opened, we recommend it be done after you have successfully performed the fit of your design. Once a successful fit is achieved, all pin functions and PSD configurations are defined, and the C code may be tailored accordingly.

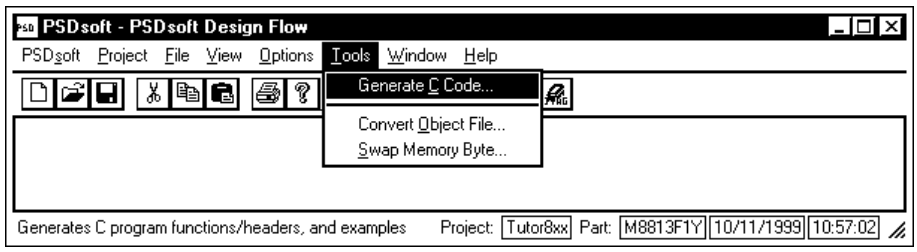
The source C programming files to implement the AGC function for this tutorial have not been provided. Since this tutorial is meant to cover all aspects of a M88 FLASH+PSD design, though, we cover a description of how you would use the C code generation utility for your own project.

Take the following steps to generate C code:

1. Pull down the **Tools** menu in the PSDsoft window and choose **Generate C Code**, as shown in Figure

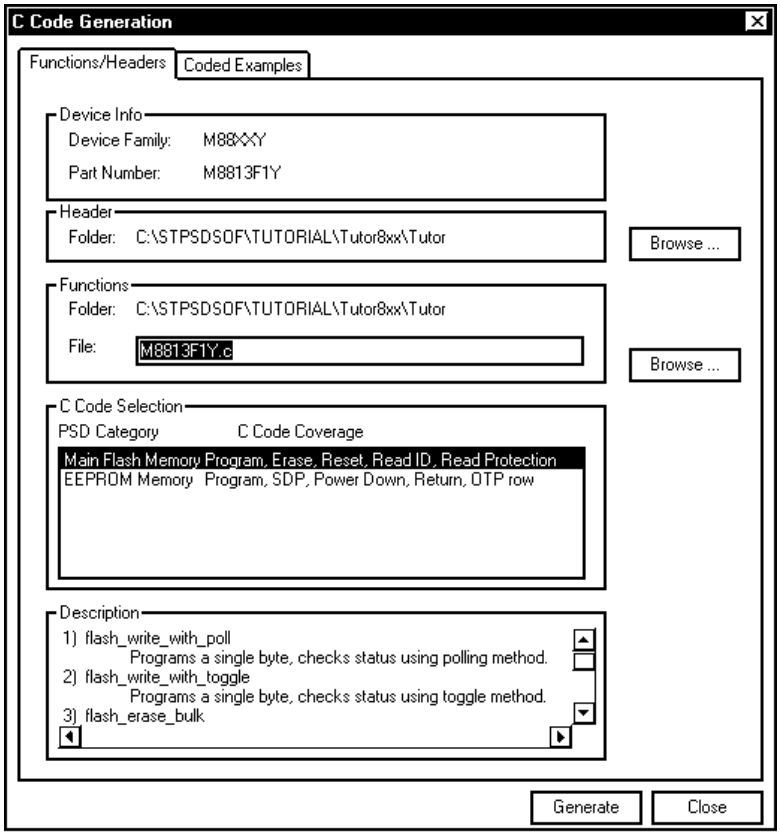
25. Alternately, click the “C Code Gen” button in the Design Flow window.

Figure 25. Tools->Generate C Code



The dialog box should appear, as shown in Figure 26.

Figure 26. C Code Generation – Functions/Headers



The “Functions/Headers” dialog box has the following sections:

– **Device Info**: This contains the M88 FLASH+PSD family and part number of the current project. These values cannot be changed unless this project is closed and a different one is opened.

– **Header**: This is for specifying the folder in which you would like to place the C header files (.h) generated by PSDsoft. (Click the **Browse** button to help in filling in this section). Typically, a folder in your MCU cross-compiler environment is chosen. You cannot change the name of the headers

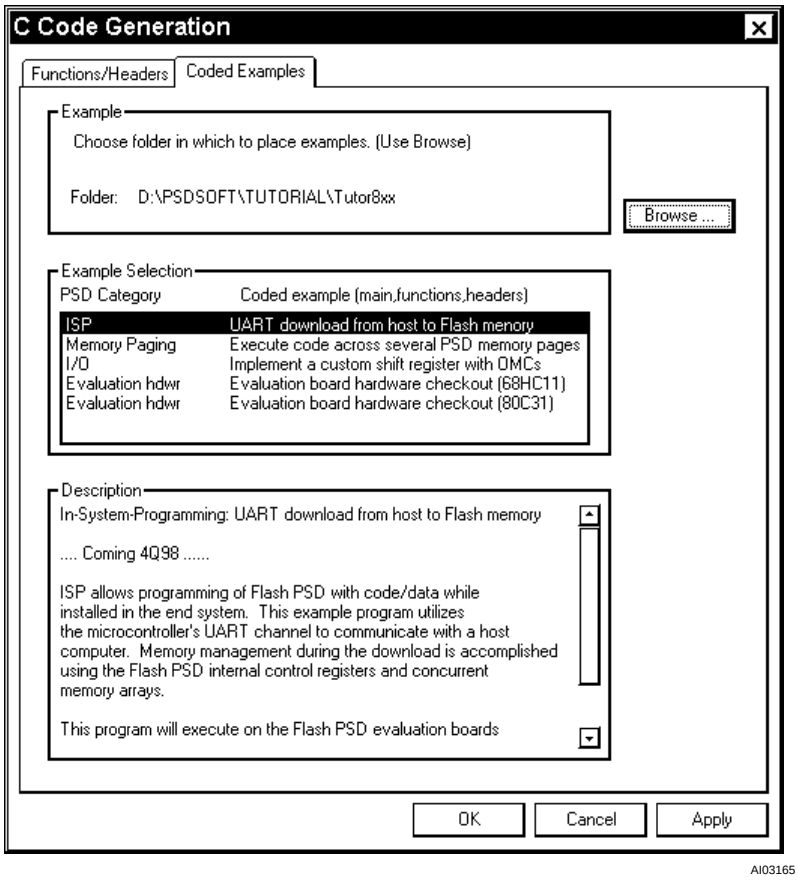
file(s) at this point since these header files may referenced by name within other header files or the C functions that are also generated by PSDsoft. Once all of the headers and functions are copied to their designated folders, you may edit the header file names any way you wish, as long as you change their names in the respective “#include” statements.

- **Functions**: This is for specifying the folder in which you would like to place the C function file (.c). (Click the **Browse** button to help in filling in this section). Typically, a folder in your MCU cross-compiler environment is chosen. This file will contain the functions you specify in the next section.
- **C Code Selection**: Select the categories of C code functions that you would like to integrate into your C application program. Under “PSD Category” are the major PSD functional groups that are supported with C code for the PSD device that is used for this project. Under “C Code Coverage” is a brief list of the individual functions that are available within each category. To select more than one category, hold the “Ctrl” key while making selections with the left mouse button. Even if more than one category is selected, though, only one .c file is generated because functions are appended within the same file.
- **Description**: This offers a description of the functions that are generated if selected in the “C Code Selection” box. If you double-click on a function within the Description box, the C code that will be generated is shown, so you can get an idea of what will appear in the sample C file.

2. After you have made your selection, first click **Apply**, then click **OK**. In this example, three files will be written to your folder(s), which are:
   - m8813F1.c: the ANSI-C source for all of the selected functions
   - m8813F1.h: the ANSI-C header file to define particular PSD registers
   - map813F1.h: the ANSI-C header file to define locations of system memory elements (Flash, EEPROM, PSD registers, etc.).

   The m8813F1.h file contains define statements for each individual C function within the m8813F1.c file. Later, edit m8813F1.h, and simply remove the comment delimiters (//) from the define statement for any C function that you would like to be compiled with the rest of your C source code.
3. Click on the “Coded Examples” tab at the top of the dialog box, as shown in Figure 27.

Figure 27. C Code Generation – Coded Examples



This sheet contains several examples that you may use as a basis for building your own C code application. These are complete projects (main, functions, and headers) targeted at a particular MCU. You may copy these files to another folder, to browse them for ideas, or cut and paste sections from the examples into your own cross-compiler environment. There are three sections:

– Example: To specify the folder in which you would like to place the example project files generated by PSDsoft. (Click the **Browse** button, and select a folder, when filling in this section).

– Example Selection: There are several areas for which you may generate C code. Each category implements a high-level system function, such as memory paging, UART downloads to Flash memory, etc.

– Description: This describes each of the coded examples in the “Example Section”

Once the C code generated by PSDsoft is integrated into your own C application, and is successfully compiled and linked by your MCU cross-compiler, you are ready for address translation.

Performing the Address Translation

The Address Translator combines the *tutor8XX.fob* file with the MCU firmware file(s) generated by your chosen MCU cross-compiler. Address Translator generates the *tutor8XX.obj* file that is to be downloaded to a programmer that is compatible with the M88 FLASH+PSD.

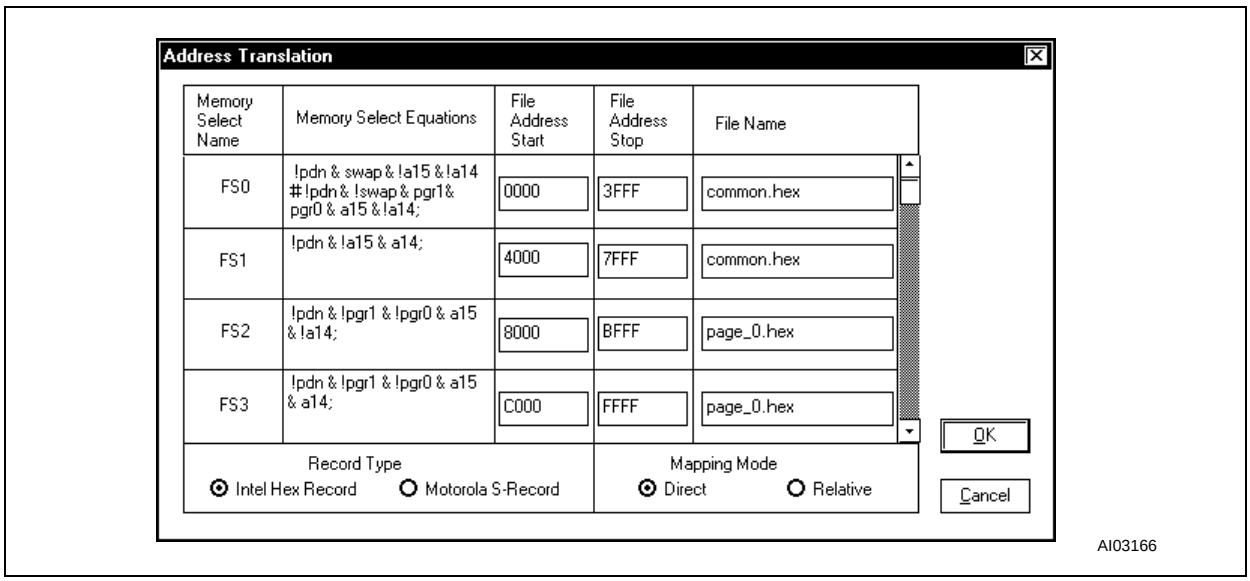
The addresses within in the generated *tutor8XX.obj* file are special “direct” addresses – meaningful to a programming device. They are not “system” addresses, that an MCU would use, or that the DPLD decodes. That is what is meant by “Address Translate”. It is a translation of “system” addresses that the

MCU and its compiler/linker knows about, to a set of “direct” addresses that a device programmer knows about.

To perform the address translation, take the following steps:

1. Pull down the **PSDsoft** menu, and choose **PSD Fitter**. Then, pull down the **Fitter** menu and choose **Address Translate**, or select “MCU Code Mapping” in the design flow. The “Address Translation” dialog box appears, as shown in Figure 28.

Figure 28. Address Translation



You will notice a warning message from PSDsoft upon entering the Address Translate window. This warning is a reminder to ensure that you take paging into account when entering the start/stop addresses and file names.

The Address Translation dialog box has the following sections:

– Memory Select Name: This is the name of the PSD memory segment that will be selected when the associated equation is true.

– Memory Select Equations: Each cell shows the equation for the appropriate PSD memory segment. These are the optimized equations from the PSDabel file. They are displayed for convenience, and cannot be modified in this window.

– File Address Start: This is the first MCU system address, from MCU compiler/linker, that will be mapped to a PSD memory segment.

– File Address Stop: This is the last MCU system address, from MCU compiler/linker, that will be mapped to a PSD memory segment.

– File Name: This is the MCU firmware file that is generated by your MCU Compiler/Linker.

– Record Type: The supported formats are Intel HEX or Motorola S-Record.

– Mapping Mode: Two modes of mapping are supported, direct and relative. For more information, please consult the *PSDsoft User Manual*.

Notice that PSDsoft attempts to fill in the File Start and File Stop Addresses based on your PSDabel equations. However, if paging is used, as in this tutorial, these file addresses must be handled carefully since PSDsoft does not know how your MCU cross-compiler and linker handles paging. As we progress, this process should become clear.

2. Type in the file names of your MCU linker output in the appropriate places. In this example, five files are used. (See Appendix F for information on the system memory map and how these files relate.) Four of the five files are to be programmed into Flash memory on different pages. The remaining file is to be programmed into the boot area of the EEPROM. The four Flash files are *page_0.hex, page_1.hex, page_2.hex*, and *common.hex*. The file for the EEPROM is *boot.hex*. Each of these files can contain up to 32 KBytes of code.
3. Enter the File Start Addresses, File Stop Addresses, and File Names according to Table 2.

Table 2. Mapping the Memory Sectors to Files

Memory Select	File Start Address	File Stop Address	File Name
FS0	0000	3FFF	Common.hex
FS1	4000	7FFF	Common.hex
FS2	8000	BFFF	Page_0.hex
FS3	C000	FFFF	Page_0.hex
FS4	8000	BFFF	Page_1.hex
FS5	C000	FFFF	Page_1.hex
FS6	8000	BFFF	Page_2.hex
FS7	C000	FFFF	Page_2.hex
EES0	0000	1FFF	boot.hex
EES1	2000	3FFF	boot.hex
EES2	-	-	-
EES3	-	-	-

In this design, a different file name has been used for each of the sections of code in the Flash memory. This is because of the address space overlap of the segments. This file scheme is used because, even though these sections of code physically reside on different memory pages, some linkers will place them in overlapping absolute address space. The method you use depends on your linker. Alternatively, you can use a single file name across many memory chip selects if your linker automatically appends extra address bits that represent your paging scheme. You would then, for example, enter 18-bit addresses to accompany the single file name, which is passed to the Address Translate utility, instead of 16-bit addresses to accompany several file names.

Optionally, you can specify only the EEPROM contents to be programmed by the device programmer. It may be desired to load system code into Flash memory while it is in-system, not on a device programmer. In this case, only information for EES0 and EES1 should be entered in the Address Translate utility.

1. Ensure that **Direct Mapping** is selected in the “Mapping Mode” box.
2. Select **Intel Hex Record** in the “Record Type” box.
3. Click on **OK** to perform the address translation. If no errors are indicated, then *tutor8XX.obj* will be placed in your project directory.

If your copy of PSDsoft includes the PSDsilosIII simulator, you should simulate and verify your design before programming the M88 FLASH+PSD. Please see the next section on how to simulate the tutorial design.

M88 FLASH+PSD Chip Simulation

PSDsilosIII is ST's version of SIMUCAD's SILOSIII simulator software. It provides chip-level simulation and design verification using the Verilog Hardware Description Language (Verilog-HDL). Appendix B lists the stimulus file (*tutor8XX.stl*) for this tutorial.

Many of the internal nodes on the M88x3Fxx are available for tracing. Descriptions of the signals that can be traced by the simulator are listed in Appendix C.

PSDsoft generates all but one of the input files required by the simulator. The file that must be created is the stimulus file (.stl). In the stimulus file, you can use the same names you used in your PSDabel file, and the predefined ones in Appendix C.

PSDsoft.run File

One of the files generated by PSDsoft for the simulation process is *PSDsoft.run,* as listed in Figure 29. It is a command batch file used by PSDsilosIII. For additional information on the PSDsilosIII commands (those commands that start with !), please refer to PSDsilosIII's on-line help.

Figure 29. *PSDsoft.run* File

```
!Reset all
!file .sav = Tutor8XX
!control .ext = all
‘timescale 1ns/0.1ns
!lib d:\psdsoft\psd8.v
‘include "tutor8XX.top"
‘include "tutor8XX.stl"
endmodule
```

In the *PSDsoft.run* file:

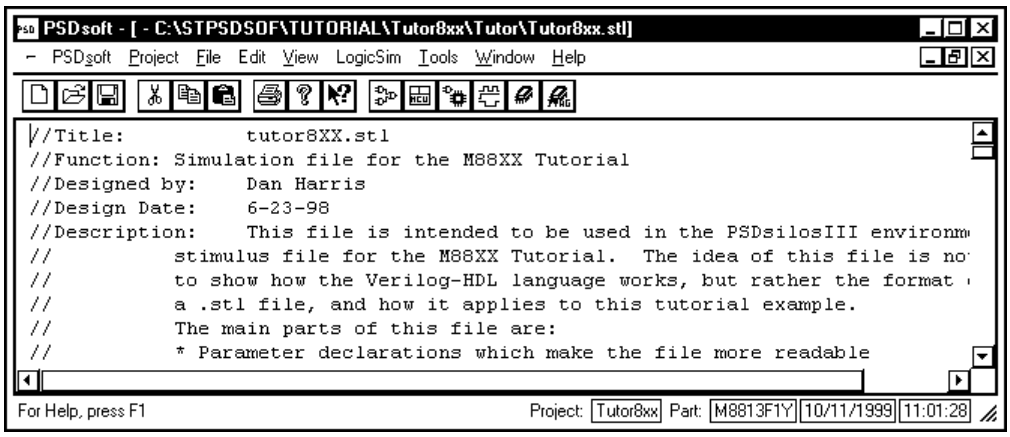
- “`time-scale 1ns/0.1ns” is a compiler directive for defining the delay values for a module
- “1 ns” is the unit of measurement for times and delays
- “0.1 ns” is the precision to which the delays are rounded off
- “`include” is also a compiler directive that allows the entire contents of a Verilog source file to be included in another file (*PSDsoft.run* in this case).

Tutor8XX.top is generated by PSDsoft, based on the PSDabel file, and allows you to use of any of the signal names within the PSDabel file. There are also parameter definitions for high impedance state signals (Z1 through Z32) in the .top file. Notice how the “endmodule” statement is the last statement in the *PSDsoft.run* file. It is there because it complements the “module WSIdesign” statement in the .top file. There is one important thing to note about the included library files: these files look for other files automatically generated by PSDsoft from the fuse-map file, and have a .afu or .pfu extension. They allow simulation of the logic, defined in the .abl file, in the stimulus file.

Running the Logic Simulator

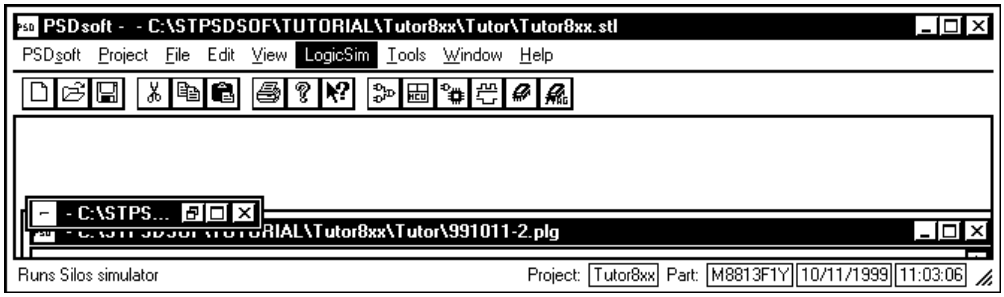
1. Review the stimulus file (*tutor8XX.stl*) listed in Appendix B.
2. Pull down the **PSDsoft** menu in the main PSDsoft window and select **PSD Simulator**, or click the simulator button on the tool bar.
3. The *tutor8XX.stl* file is automatically opened in PSDsoft, as shown in Figure 30.

Figure 30. Running the Logic Simulator



4. Click on **LogicSim**, as shown in Figure 31, to invoke the PSDsilosIII simulator

Figure 31. LogicSim

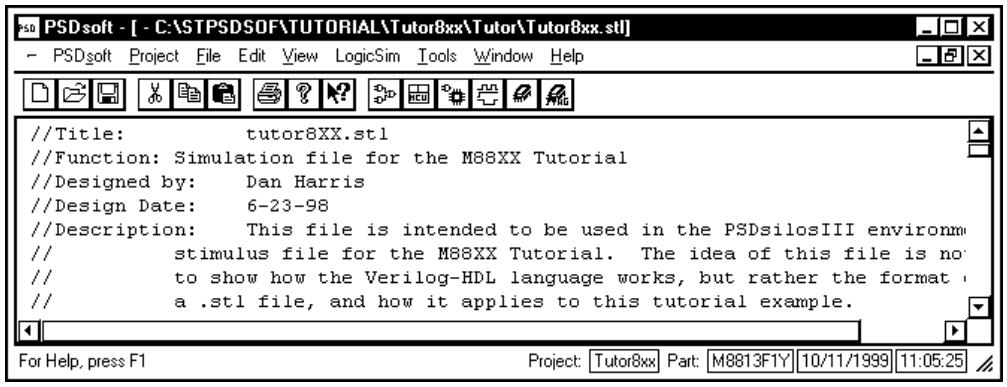


The following events happen automatically, as a result of clicking on the **LogicSim** button:

– The PSDsilosIII simulator starts

– The simulator loads the project *tutor8XX.spj*, *PSDsoft.run*, and a window displaying the *tutor8XX.stl* file, as shown in Figure 32.

Figure 32. Logic Simulator Input



5. Click on the **Go** button. This automatically opens an “Output” window for viewing the results of the simulation, as shown in Figure 33.

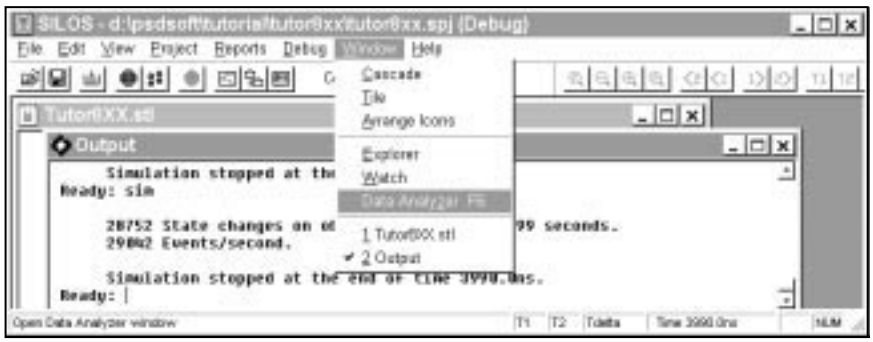
Figure 33. Logic Simulator Output

Running the Analyzer

Now that the logic simulation is complete, the results can be displayed with the PSDsilosIII Data Analyzer by performing the following steps:

1. Pull down the **Window** menu and select **Data Analyzer**, as shown in Figure 34; or press F6, or click on the appropriate button on the tool bar.

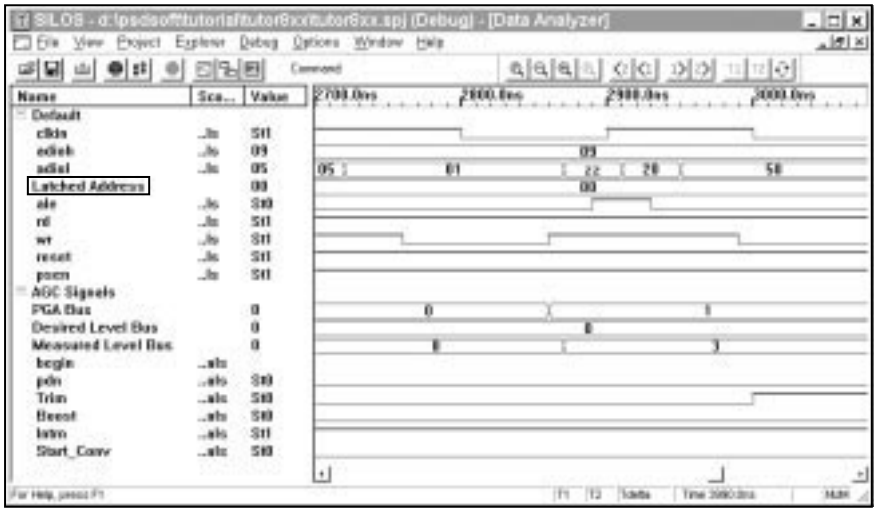
Figure 34. Running the Analyzer



AI03171

2. The PSDSilosIII Data Analyzer window appears with the simulation results displayed on screen, as shown in Figure 35 (but *Your screen will look different*.). Please see the tutorial on the Data Analyzer and the Explorer under **Help->Contents** on how to rearrange and group signals.

Figure 35. Simulation Results

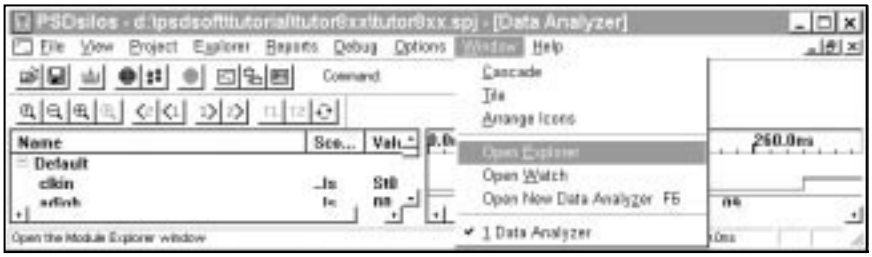


AI03172

Working With the Explorer

The Explorer in PSDSilosIII can be used in conjunction with the Data Analyzer to add and trace signals. To open the explorer, ensure that you have simulated the design by following the steps in the section entitled “Running the Logic Simulator”, on page 34. Next click on the **Window->Open Explorer** menu selection or the Explorer button and the explorer window will appear, as shown in Figure 36. The Explorer shows all viewable signals.

Figure 36. Explorer



AI03173

Signals can be added to the Data Analyzer window using the Explorer by holding the “CTRL” button down, and clicking on all the signals that you want to add to the Data Analyzer window. Once you have chosen all the desired signals, right-click on one of the signals, and select **Add Signals to Analyzer**. Next, click anywhere in the Data Analyzer window, and the signals you added will appear at the bottom of the window, as shown in Figure 37.

Figure 37. Adding Signals to the Analyzer

AI03174

For more information on the Explorer or Data Analyzer, please see the on-line help, and the *PSDsilosIII User Manual*. Also, please refer to this manual for information on how to us the PSDsilosIII Watch Window, which is beyond the scope of this tutorial.

Programming the M88 FLASH+PSD

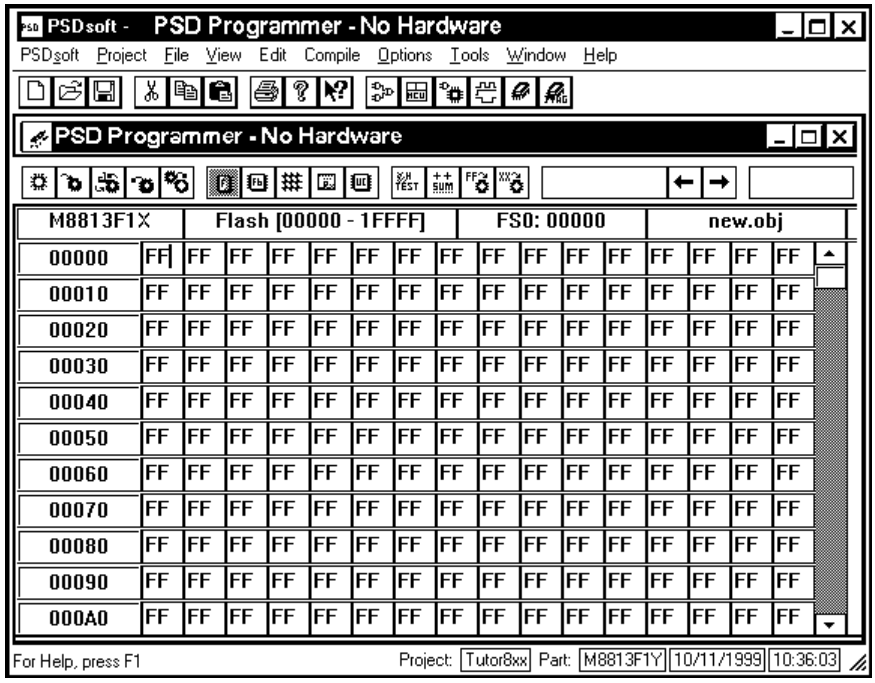
The PSD Programmer is the programming interface to the ST MagicProIII®, PSDpro, and FlashLink programmers. It enables downloading any PSD .obj file; and displays the Flash and EEPROM locations, the PLD fuse-map, and the configuration bits (ACR). You can also perform the following operations from the **Functions** menu:

– **Blank Test:** to check to see if the device is blank.
– **Upload:** to upload the contents of the device that were programmed to the buffer.
– **Program:** to program the device with the .obj file.
– **Verify:** to verify the programmed device against the .obj file in the buffer.
– **Erase:** to erase the device completely.

If you have a MagicProIII, PSDpro, or FlashLink device programmer connected to your PC, take the following steps to program the M88 FLASH+PSD after the design has been compiled and the .obj file has been generated:

1. Pull down the **PSDsoft** menu in the main PSDsoft window and choose **PSD Programmer**, or click the appropriate button on the tool bar.
2. The *tutor813XX.obj* file is downloaded and displayed on the screen automatically, as shown in Figure 38.

Figure 38. PSD Programmer

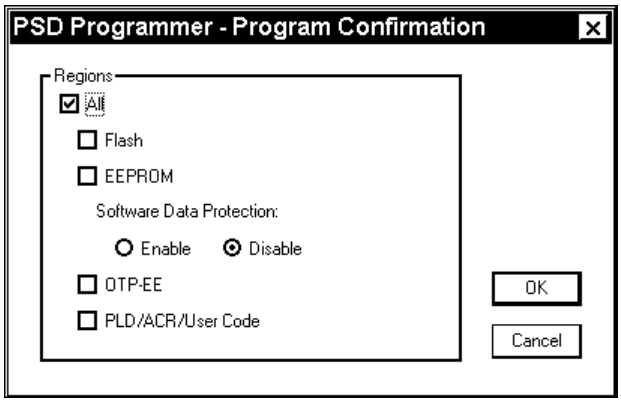


AI03175

Assuming you have a MagicProIII programmer installed, to program a device do the following:

1. Pull down the **Functions** menu and select **Program;** or click the Program button on the tool bar that is available when the PSD Programmer is invoked. The “PSD Programmer – Program Confirmation” dialog box appears, as shown in Figure 39, which enables you to program the Flash, EEPROM or PLD/ ACR (PSD Configuration) regions of the device.

Figure 39. PSD Programmer – Program Confirmation

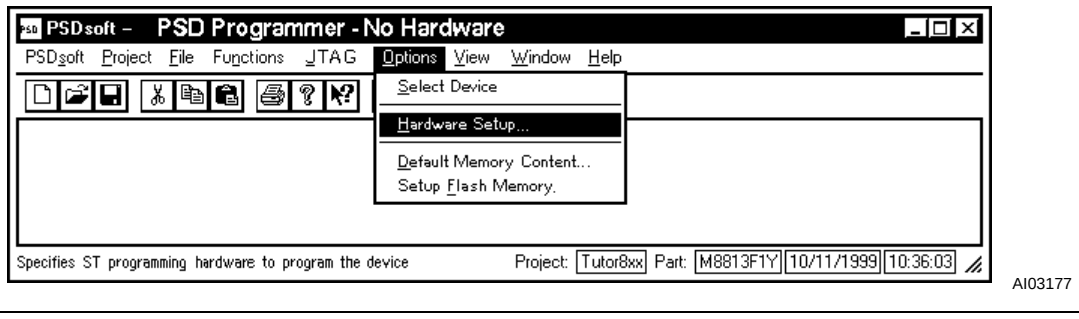


2. Select “All”, as shown in Figure 39.
3. Place the PSD device into the programmer, checking that it is correctly orientated, and snap the lid down on the device carrier. Then, click on the **OK** button. As programming takes place, the MagicProIII programmer checks each location, after it is programmed, to make sure it matches the contents in the .obj file. If a particular location cannot be programmed properly, an error message is shown. If this occurs, you must restart from the beginning, and program a fully erased and functional part.

PSDpro

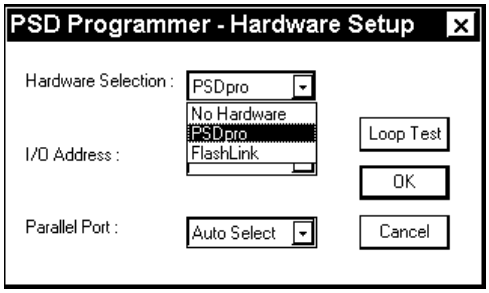
If you have a PSDpro connected to one of your PC’s parallel ports, you can select and configure it by going to the **Options** menu in the PSD Programmer environment and selecting **Hardware Setup**, as shown in Figure 40.

Figure 40. PSD Programmer – Hardware Setup



Once the “PSD Programmer – Hardware Setup” dialog box appears, select PSDpro, in the “Hardware Section”, as shown in Figure 41.

Figure 41. PSD Pro



Next, you will see that the **Auto Select** option becomes active. This means that PSDsoft will automatically detect to which PC parallel port your PSDpro is connected. Just click **OK**, and the PSDpro will be detected and configured if the connections are good.

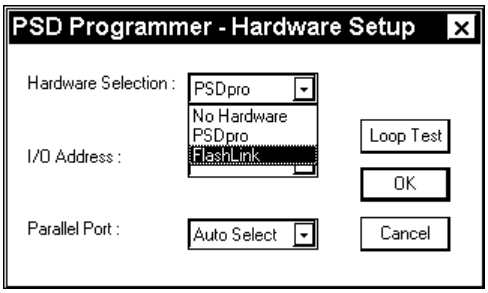
The same menu options and capabilities that apply to the MagicProIII in the section above also apply to the PSDpro.

JTAG: FlashLink

If you have a FlashLink cable installed on one of your PC's parallel ports, you can select and configure it as follows:

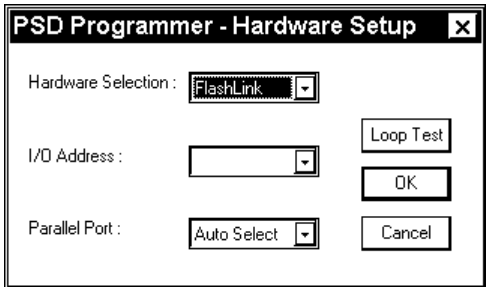
1. Go to the **Options** menu in the PSD Programmer environment, and select **Hardware Setup**. Once the "PSD Programmer – Hardware Setup" dialog box appears, select FlashLink, in the "Hardware Section", as shown in Figure 42.

Figure 42. FlashLink



2. Next, you will see that the **Auto Select** option becomes active, as well as **Loop Test**, as shown in Figure 43.

Figure 43. Auto-select and Loop Test



Auto Select means that PSDsoft will automatically detect to which PC parallel port your FlashLink cable is connected (even if the other end of the FlashLink cable is not connected to the target system). Just click **OK**, and the FlashLink cable will be detected. Optionally, you can return to the Hardware Setup menu to run a Loop Test on the FlashLink cable. This is a hardware integrity test that requires the loop-back cable (that is provided) to be installed on the FlashLink cable. (Please see the FlashLink installation manual).

3. Now connect the FlashLink cable to your target system, and power-on the system. The target system needs to be powered up since the FlashLink circuitry draws its power from the target.
4. Set up your JTAG chain (as described in the next section)
5. Once your JTAG chain has been set up, program your device while it is in-system. (Programming is accomplished in the “JTAG Chain Setup” window. This is described in the next section.)

Setting up a JTAG Chain

This section takes you step-by-step through the creation of a JTAG Chain File. Since this procedure has not been finalized, please check our web site (www.st.com) for updates to this document.

The following rules apply for setting up a JTAG chain:

- A JTAG chain of one to or more devices must be defined.
- All JTAG compatible devices that are connected to the JTAG bus, including the M88 FLASH+PSD and non-PSD devices from other vendors compose a JTAG chain.
- Non-PSD devices that are part of the JTAG chain will be placed, automatically, in bypass mode.
- The length of the instruction register, along with a name and device ID must be entered for each non-PSD device. (In future versions of PSDsoft, you will be able to load this information automatically with a BSDL file.)
- Before programming the PSD device(s), the user must have a valid .obj file for each PSD device in the chain
- Additionally, a Serial Vector Format file, *filename*.svf, can be created for third party JTAG programming support.

Please refer to Application Note *AN1153* for information in these areas:

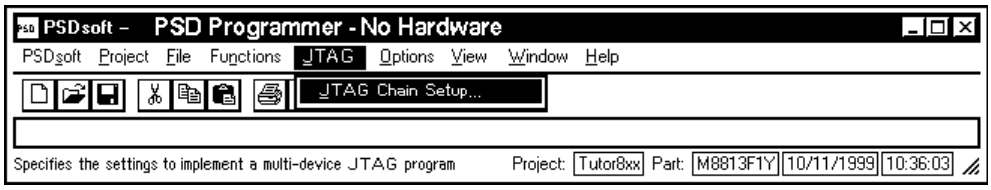
– JTAG Spec Compliance
– Programming Support
– Program/Erase Flow Control
– SVF/BSDL file information
– Enhanced ISP functions
– Multiplexed JTAG pin functions
– Dedicated JTAG pin functions

– ST JTAG ISP connector
– JTAG Chaining

Now, let's step through a sample JTAG chain setup, and create a JTAG chain file (.jcf).

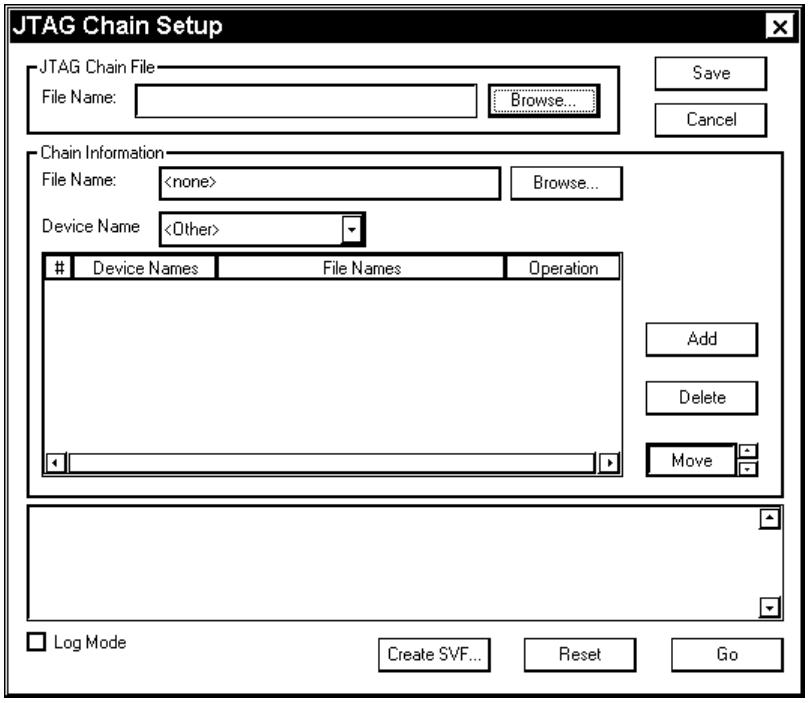
1. Under **JTAG** menu, select **JTAG Chain Setup**, as shown in Figure 44, or click "JTAG Prog" in the design flow.

Figure 44. JTAG Chain Setup



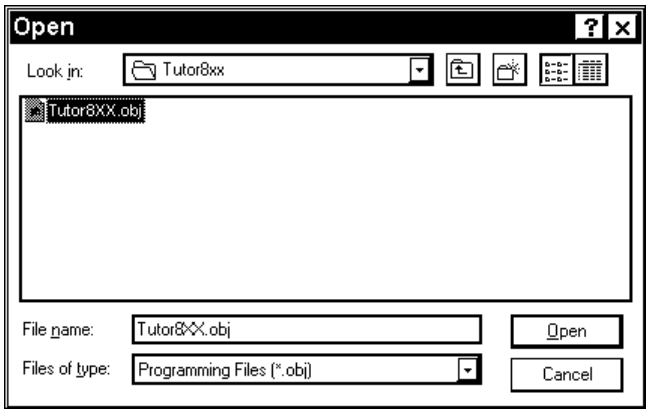
This opens the "JTAG Chain Setup" dialog box, as shown in Figure 45.

Figure 45. JTAG Chain Setup Dialog Box



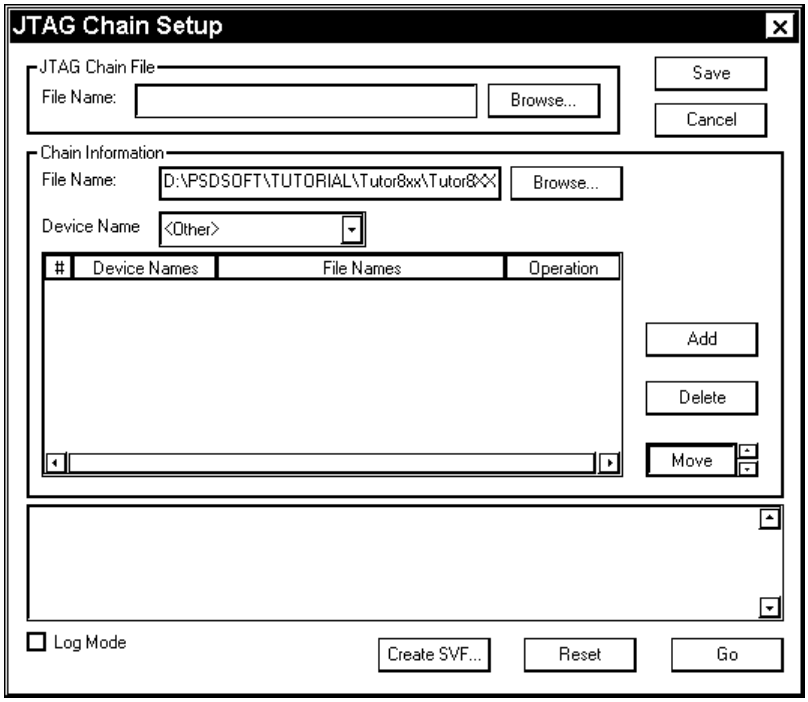
2. In the "Chain Information" box, click **Browse**. This brings up the "Open" window, as shown in Figure 46. Select the *Tutor8XX.obj* file in the \PSDSOFT\TUTORIAL\TUTOR8XX\TUTOR\ directory, and click **Open**.

Figure 46. JTAG Chain Setup – Open Window



Your “JTAG Chain Setup” window should now appear as shown in Figure 47.

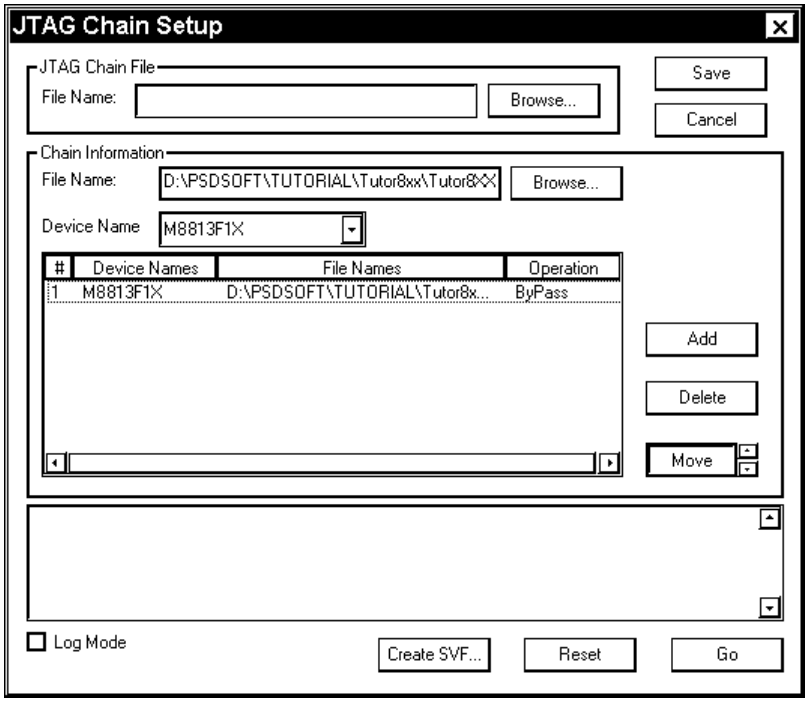
Figure 47. JTAG Chain Setup Window



If the device name (M8813F1x in this case) does not automatically appear in the “Device Name” window, select the appropriate device.

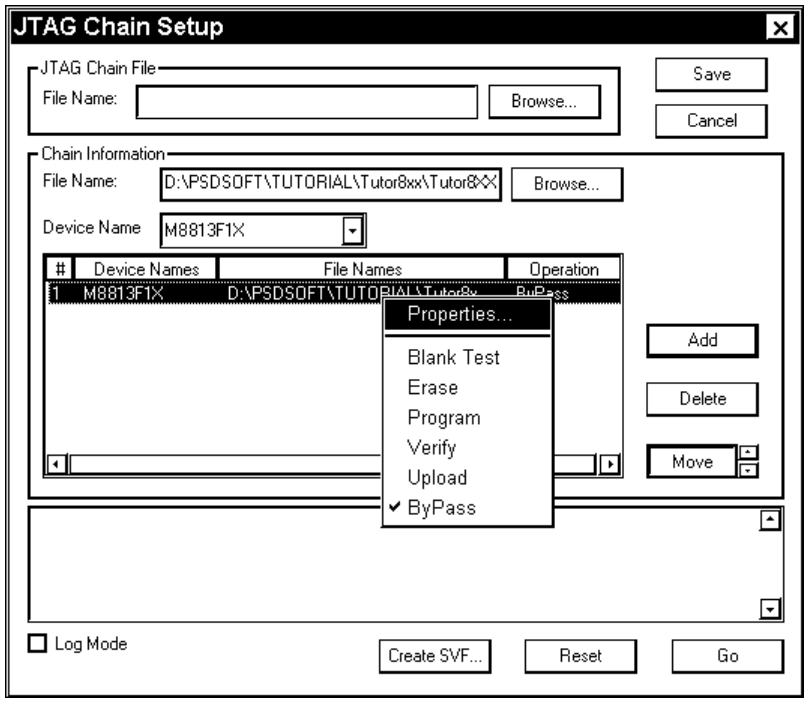
3. Click the **Add** button. Your “JTAG Chain Setup” window should appear as shown in Figure 48.

Figure 48. JTAG Chain Setup Window – After an Add



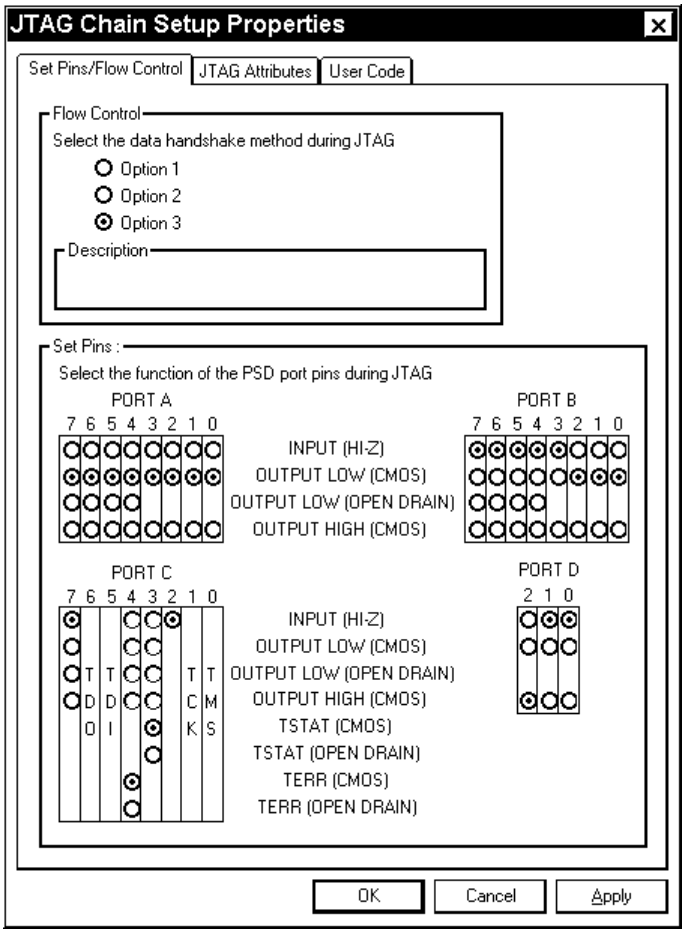
4. **Right**-click anywhere on the line that just appeared (line 1), and select **Properties**, as shown in Figure 49.

Figure 49. JTAG Chain Setup Window – Selecting Properties



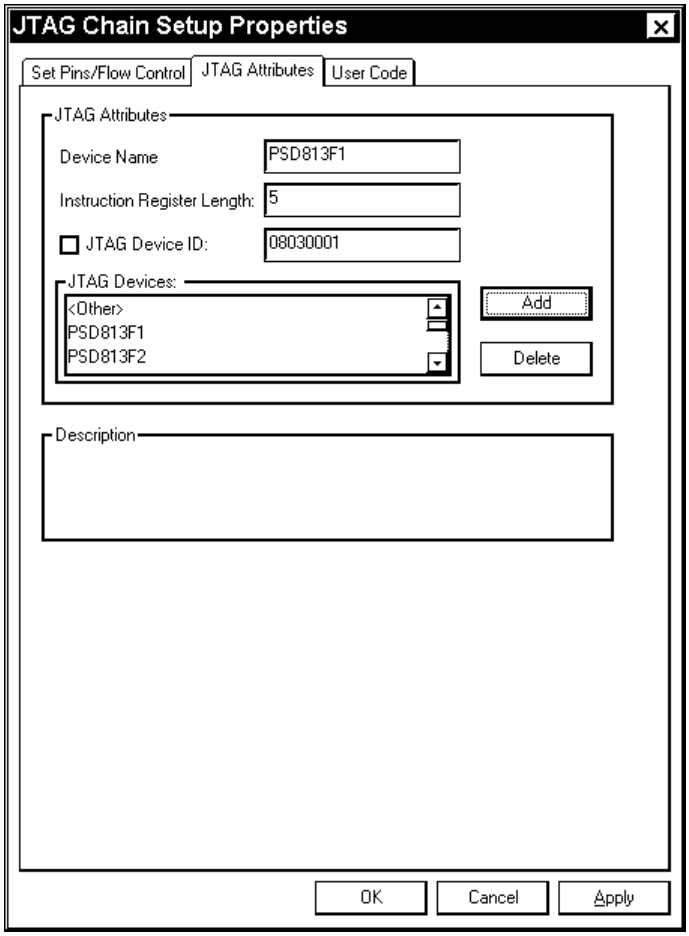
5. This opens the “JTAG Chain Setup Properties” dialog box. Ensure that the “Set Pins/Flow Control” tab is selected, and set up the window with the following selections (based on Figure 3 of this tutorial). The proper selections are shown in Figure 50. Under “Flow Control”, select **Option 3**. In the “Set Pins” box, set up the ports as follows:
   - Port A: set all the pins to “OUTPUT LOW (CMOS)”
   - Port B: set pins pb7 to pb3 to “INPUT (HI-Z)”, and pins pb2 to pb0 to “OUTPUT LOW CMOS)”
   - Port C: change pc3 to “TSTAT (CMOS)”, and pc4 to “TERR (CMOS)”. Leave the rest of the pins as they are.
   - Port D: set pins pd1 and pd0 to “INPUT (HI-Z)”, and pd2 to “OUTPUT HIGH (CMOS)”

Figure 50. JTAG Chain Setup Properties



6. Click on the **Apply** button (which saves the information you have entered, so far, and greys the “Apply” button out). Then, click on the “JTAG Attributes” tab. Your “JTAG Chain Setup Properties” window should now appear as shown in Figure 51.

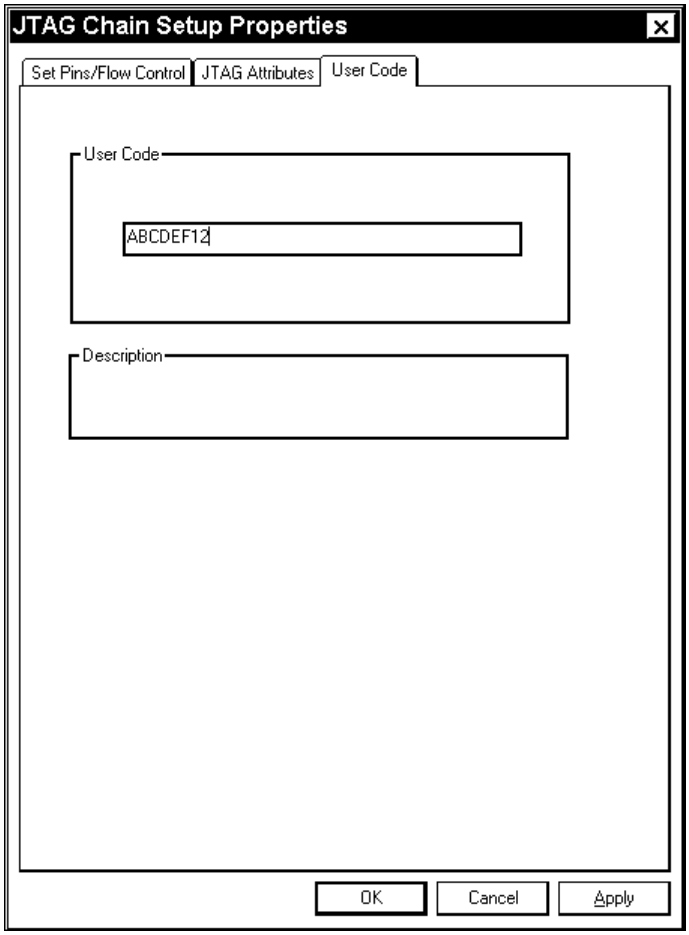
Figure 51. JTAG Chain Setup Properties – JTAG Attributes



The “Device Name”, “Instruction Register Length:”, and “JTAG Device ID:” are all greyed out because this information is automatically entered whenever you select a M88x3Fxx device. If you want to enter information about a non-PSD device, which is to be included in your chain, here is the place to do it. If you add another device, you need to enter valid information in the “JTAG Attributes” section. Also, note that if you select the “JTAG Device ID” box, PSDsoft verifies the JTAG ID before programming or erasing the device.

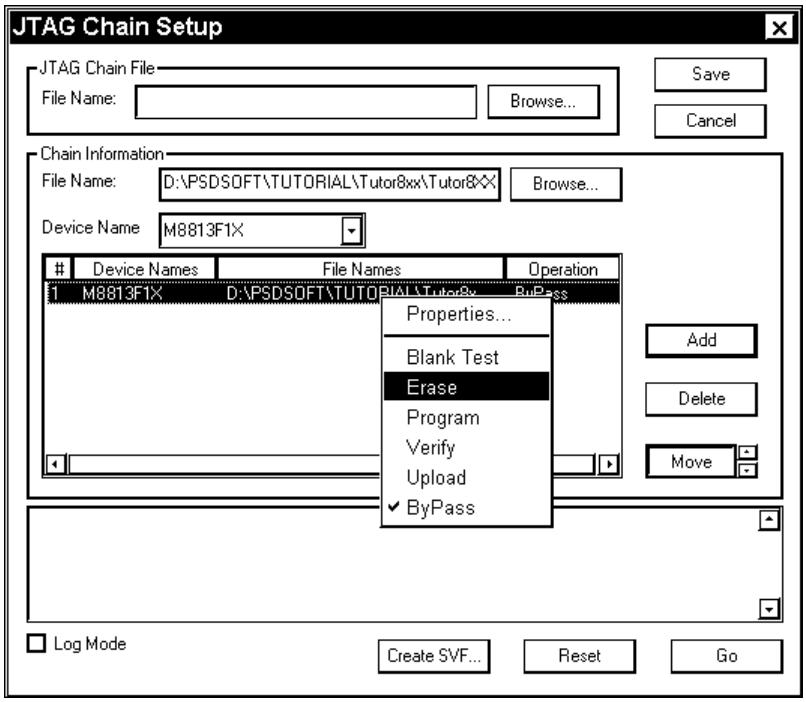
7. Click on the “User Code” tab, to give the display shown in Figure 52. If you enter a value in the “User Code” box, the value is compared with the User Code already programmed into the device before any JTAG operation occurs (e.g. Erase, Program, etc.). If you leave this area blank, no comparison is performed. Enter ABCDEF12 in the “User Code” box. Then press **Apply** (which greys the “Apply” button out), and finally press **OK**.

Figure 52. JTAG Chain Setup Properties – User Code



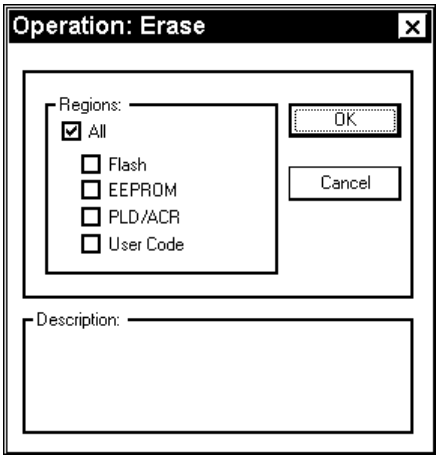
8. Now, you should be back to the “JTAG Chain Setup” window. Right click on the same line as you did in step 4, only this time, choose **Erase**, as shown in Figure 53.

Figure 53. JTAG Chain Setup Erase



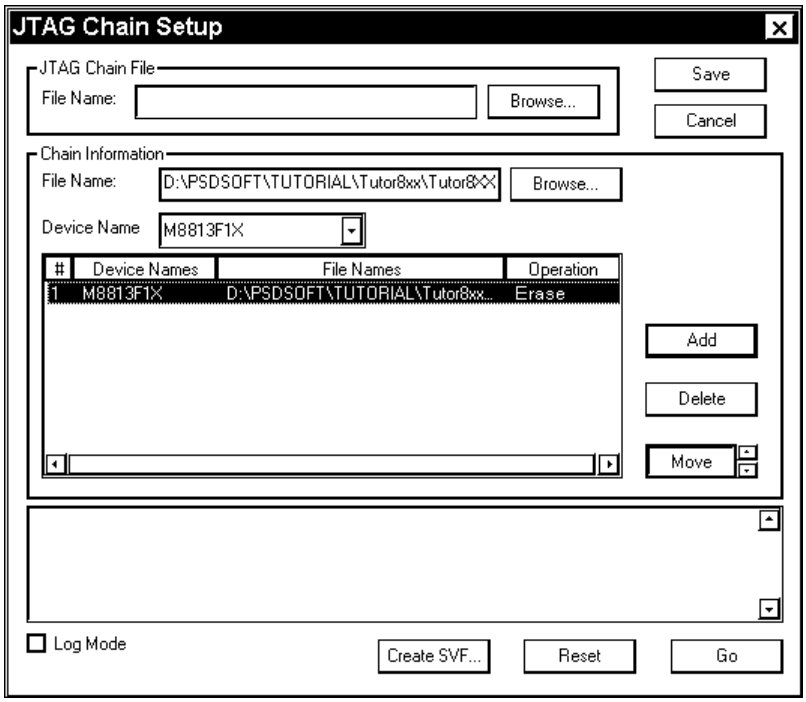
9. You should now see the “Operation: Erase” dialog box, as shown in Figure 54. Ensure that **All** is checked under the “Regions” block, and click **OK**.

Figure 54. Operation Erase



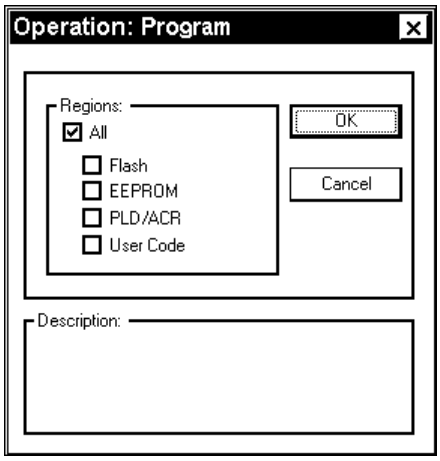
10. Your “JTAG Chain Setup” window should now appear as shown in Figure 55. Click **Go** to start the exchange on the FlashLink cable. If the “Log Mode” box is checked, the JTAG communication status will appear in the PSDsoft log window, and in the file *tutor8XX.log*. Turning on the Log Mode feature will slow down the JTAG communications.

Figure 55. JTAG Chain Setup – After the Erase



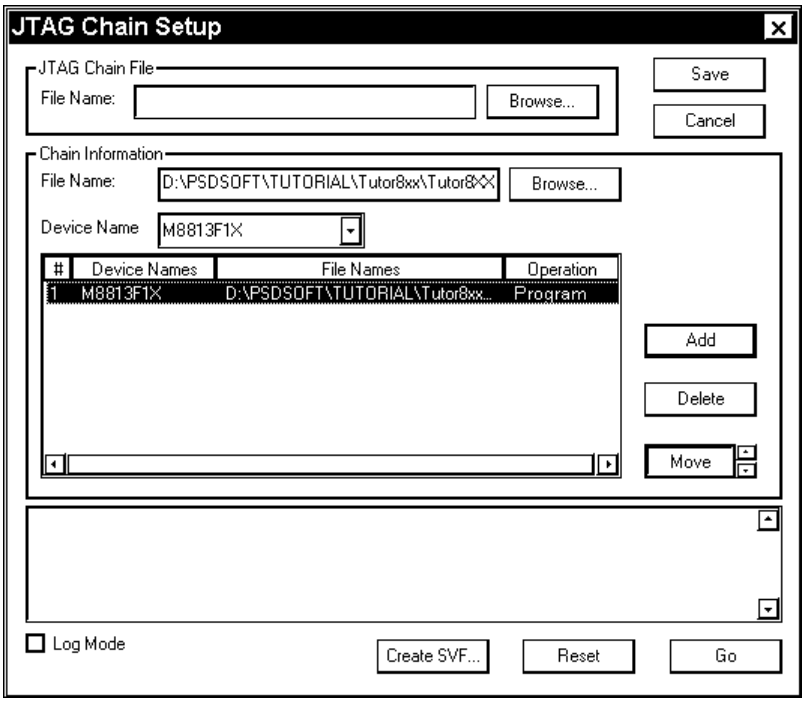
11. After the erase has completed (as indicated in the log window), right click on the same line as in step 4, and choose **Program**. Select **Ok** when the window, as shown in Figure 56, appears.

Figure 56. Operation Program



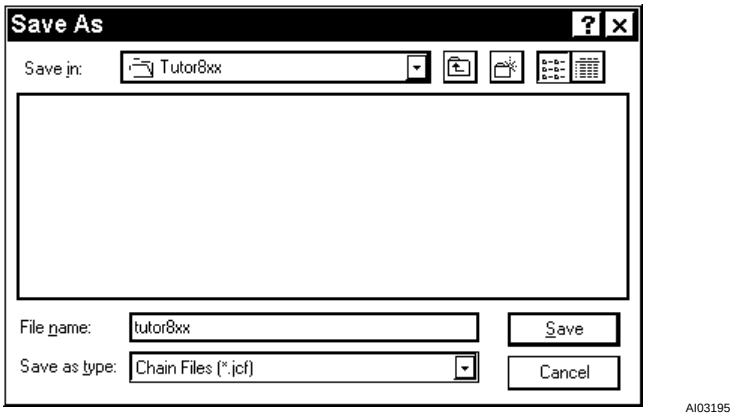
Your “JTAG Chain Setup” should now appear as shown in Figure 57. Select **Go**, and the PSD will be programmed with the information in the *tutor8XX.obj* file.

Figure 57. JTAG Chain Setup – After Program



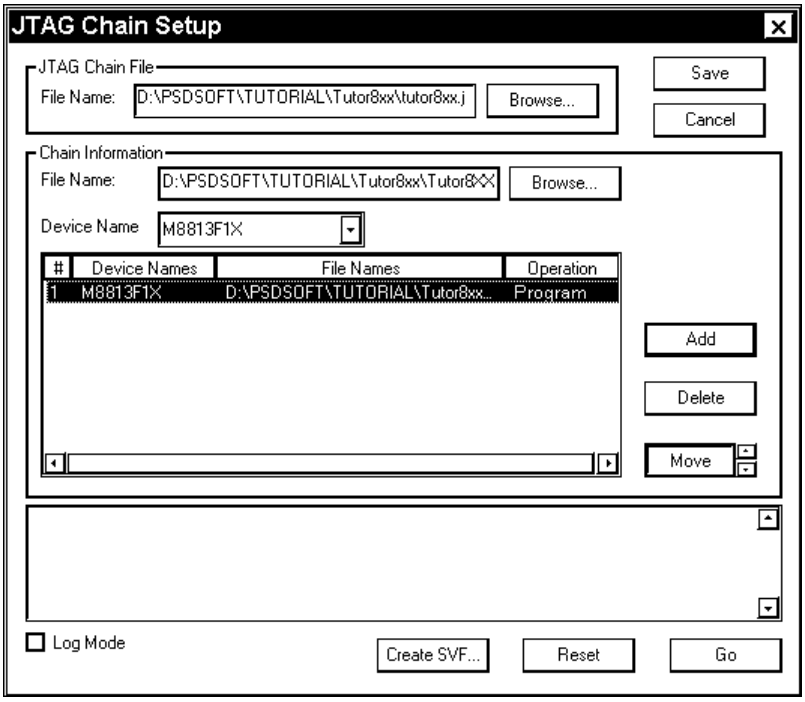
12. After you have programmed the device, click **Reset** at the bottom of the window. This resets the target circuit board that is connected to the FlashLink cable. This is needed after the Flashlink programs the PSD because the MCU will have lost its “mind” at this point.
13. Click on the **Save** button, to save your work in a JTAG Chain File for future use. This action brings up the “Save As” dialog box, as shown in Figure 58. Type “*tutor8xx*” in the “File name” box, and click **Save**. The file *tutor8xx.jcf* will be created.

Figure 58. JTAG Chain Setup – Save



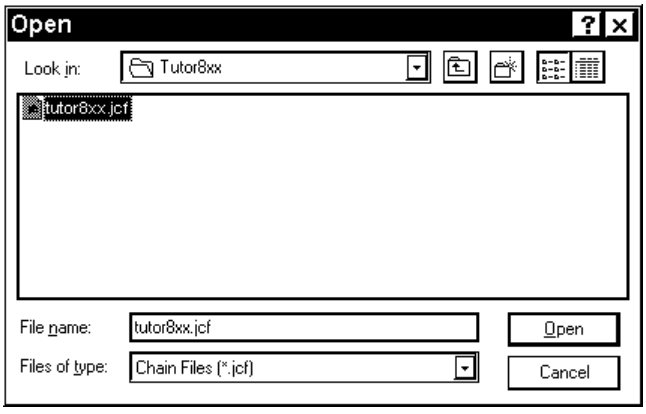
14. Now, the “JTAG Chain Setup” window should appear as shown in Figure 59.

Figure 59. JTAG Chain Setup – After the Save



15. If you need to load this .jcf file in the future, you will have to click on the **Browse** button, which will bring up the “Open” dialog box, as shown in Figure 60. Choose the *tutor8xx.jcf* file, and click **Open**.

Figure 60. JTAG Chain Setup – Open Dialog Box



Lastly, before leaving the “JTAG Chain Setup” window, you may wish to create a Serial Vector Format (.svf) file for use with a third-party programmer. To do so, click the **Create SVF** button, and **Browse** through your directory tree to find a place where you want to place the .svf file.

ISP AND THE M88 FLASH+PSD

The M88 FLASH+PSD may be programmed in-system, with or without participation from the MCU. For ISP with the MCU, please see Appendix F for UART download information and considerations. For ISP without MCU participation, see the Section entitled “JTAG: FlashLink“ on page 41. For FlashLink JTAG programming within the PSDsoft environment please see the application note, *AN1153*.

REFERENCES

- *M88 FLASH+PSD* Family Data Sheet
- Application Note *AN1153* for detailed use of the JTAG channel
- Application Note *AN1171* for details on the CPLD and I/O pins
- Application Note *AN1176* for a design guide for the 68HC11 and M8813F1x.
- Application Note *AN1177* for a design guide for the 80C51XA and M8813F2x.
- Application Note *AN1178* for a design guide for the 80C51 and M8813F2x.

APPENDIX A: ABEL DESIGN FILE "*TUTOR8XX.ABL*"

```
module Tutor8XX
title '8XX Tutorial Design File';
// Designed by:Dan Harris and Mark Rootz
// Design date:6-16-98
// Description:This shows the logic implementation of the sample design in the 8XX
Tutorial.
//    The design highlights the following functionality of the M88x3Fxx:
//    * Effective and efficient use of the Input and Output Micro<->Cells
//    * How to use I/O pins while the underlying Micro<->Cell is being used for
//     other functionality.
//    * Use of the WSIPSD PROPERTY statement to output demultiplexed address bits,
//     and define Input Micro<->Cells/Output Micro<->Cells.
//    * Multiplexing the JTAG pins with other I/O.
//    * How to logically interface to an 80C31 MCU, RTC, and an AGC circuit.
// Revision:1.0
// Rev Date:9-21-98
// Convention:The n is used throughout the file to indicate active low signals.
//    Note that it is not used with the reserved signal names below.


"*************************  Bus Interface signal declarations
*************************
// The reserved signal names are automatically assigned to the appropriate pin
// The following are inputs from the MCU
wr  pin; "CNTL0 Input:(pin 47)- write strobe
rd  pin; "CNTL1 Input:(pin 50)- read strobe
psen pin;"CNTL2 Input:(pin 49)- program store enable
ale pin; "PD0 Input:(pin 10)- address latch enable
reset  pin;"Input:(pin 48)- system reset
a15..a0 pin;"Input:(pins 46..39,37..30)- demuxed address


"*************************   Port A, B, C, D pin declaration
*************************

// Port A I/O
// Control outputs are MCU I/O mode outputs
Control0..Control2pin 23, 22, 21;"Some generic control signals
// Assign the latched/demultiplexed address to Port A, pins pa4 to pa0.
WSIPSD PROPERTY 'Address_Out Aout[4:0]:Addr_Out[4:0]';


// Port B I/O
```

```
PGA_Din2..PGA_Din0pin 5, 6, 7;"Data bits used to program the PGA
               "Implemented with MCU I/O mode
Measured_Level3..Measured_Level0 pin istype 'reg';"Upper 4 bits of the A/D
converter (ADC)
WSIPSD PROPERTY 'DataBus_IMC D[7:4]:Measured_Level[3:0] PortB';

// Port C I/O
// Note that pins pc0, pc1, and pc5-6 are multiplexed output/JTAG signals. pc3,
pc4, and pc6
// are JTAG signals that are not multiplexed.
// Ensure that under "Global Configuration" with the "JTAG Configuration" tab
selected that none
// of the boxes enabling various JTAG signals on certain pins are checked because
the device
// will expect only valid JTAG signals on these pins, and no multiplexing can be
done under these
// circumstances.
// Pin pc2 (pin 18) is used for VSTBY (set in global configuration)
Intrn    pin 20;"Interrupt the MCU when the gain needs to be changed/JTAG TMS
Start_Convpin 19;"Start Conversion signal for the ADC/JTAG TCK
Trim  pin 17;"The gain is too high and needs to be decremented/JTAG TSTAT
Boost    pin 14;"There is not enough gain--increment it/JTAG TERRn
JCEn  pin 11;"JTAG chip enable signal used to demultiplex Port C output and JTAG
I/O


// Port D I/O
// pd0 (pin 10) is assigned above to the ALE signal from the microcontroller.
// Any external chip selects that are generated by decoding an address should be
placed on
// Port D when possible to save as many resources as possible.
RTCcsnpin 8;"Real Time Clock (RTC) chip select/JTAG TDO
clkin pin;"Port D pin pd1 (pin 9) System clock


// Output Micro<->Cell assignments
WSIPSD PROPERTY 'DataBus_OMC D[7:4]:Desired_Level[3:0] MCELLAB';
WSIPSD PROPERTY 'DataBus_OMC D7:begin_cycle MCELLBC';

"*************************    Internal node declarations
*************************

mxord3      node;"This signal is needed to save product terms
meqd     node;"True when the measured signal equals the desired signal level
```

```
begin_cycle  node istype ’reg’;"This signal takes the state machine out of idle
STATE1..STATE0 node istype ’reg’;"State machine bits
Desired_Level3..Desired_Level0 node istype ’reg’;"The desired gain level

fs7..fs0  node;"Main Flash memory segments
ees3..ees0node;  "EEPROM memory segments
// Reserved node names
rs0      node;"Select for the SRAM memory space
csiop       node;  "Control register
jtagsel  node 102;"This is the JTAG enable product term. It is used to enable
            "the JTAG port signals.
pgr1..pgr0   node;"Internal PSD Page Register bits


// The following page register bit definitions are an example of how to manipulate
memory to
// facilitate ISP. This scheme is explained in Appendix F of Application note 57.


swapnode 117;" This page register bit (pgr7) will be used for swapping
         " memory segments after a firmware download from the 8031
         " UART port has completed. When swap = 0, the secondary
         " NVM occupies boot area for ISP, swap = 1, primary NVM
         " occupies boot area.

enable_data_half node 116;
         " This page register bit (pgr6) will be used to manipulate
         " the EEPROM. The use of this bit is one way to divide the
         " EEPROM in into two equal sections, one for boot and one for
         " general data. When this bit=0, the boot section is active.
         " When this bit = 1, the data section is active.


"*************************        DEFINITIONS        *************************

DLEVEL = [Desired_Level3..Desired_Level0];"Desired gain level set by MCU
MLEVEL = [Measured_Level3..Measured_Level0];"Measured gain level latched by IMCs
STATE_MACHINE = [STATE1..STATE0];
X = .x.;"Don’t care symbol
C = .c.;"Clock symbol
page = [pgr1,pgr0];
address = [a15..a0];"De-muxed microcontroller address signals
```

EQUATIONS

```
"*************************       DPLD equations       *************************

// Generate active high chip selects for the main Flash segments. Each segment is
16K bytes
// for the M88x3Fxx devices.
fs0 = ((address >= ^h8000) & (address <= ^hBFFF) & (page == 3) & !swap)
  # ((address >= ^h0000) & (address <= ^h3FFF) & (page == X) & swap);
fs1 = (address >= ^h4000) & (address <= ^h7FFF) & (page == X);
fs2 = (address >= ^h8000) & (address <= ^hBFFF) & (page == 0);
fs3 = (address >= ^hC000) & (address <= ^hFFFF) & (page == 0);
fs4 = (address >= ^h8000) & (address <= ^hBFFF) & (page == 1);
fs5 = (address >= ^hC000) & (address <= ^hFFFF) & (page == 1);
fs6 = (address >= ^h8000) & (address <= ^hBFFF) & (page == 2);
fs7 = (address >= ^hC000) & (address <= ^hFFFF) & (page == 2);


// Generate active high chip selects for the EEPROM segments. Each segment is 8K
bytes for
// the M8813F1x devices.
ees0 = ((address >= ^h0000) & (address <= ^h1FFF) & (page == X) & !swap)
   # ((address >= ^h8000) & (address <= ^h9FFF) & (page == X) & swap &
!enable_data_half);
ees1 = ((address >= ^h2000) & (address <= ^h3FFF) & (page == X) & !swap)
   # ((address >= ^hA000) & (address <= ^hBFFF) & (page == X) & swap &
!enable_data_half);
ees2 = (address >= ^hC000) & (address <= ^hDFFF) & (page == X) & swap &
enable_data_half;
ees3 = (address >= ^hE000) & (address <= ^hFFFF) & (page == X) & swap &
enable_data_half;


//Generate active high chip select for the PSD SRAM (2K bytes).
rs0 = (address >= ^h0100) & (address <= ^h08FF) & (page == X);


// Generate active high chip select for the PSD control registers. 256 contiguous
bytes must be
// decoded for all M88x3Fxx devices.
csiop = (address >= ^h0900) & (address <= ^h09FF) & (page == X);


// Enable the JTAG port when the JTAG Chip Enable (JCEn) Signal is active
```

```
jtagsel = !JCEn;


"*************************     GPLD/ECSPLD equations
*************************

// IMPORTANT NOTE: Comment these next four equations out for the ABEL simulation
only. The
//        PSDsilosIII Simulator requires the equations (and they are functionally
//     correct). The problem is that the MCU presets (loads) and clears these
//     registers, and the value is not registered through the D input. However,
//     the ABEL simulator does not reconize any "dot" extentions (As these would
//     normally be set up through equations). The basic functionality can still
//        be properly tested, but how it is actually implemented in hardware is
//     slightly different.
//     So, if you intend to use the ABEL Simulator, comment out the following
//     four lines so that the test vectors at the end of the file will work
//     properly.
DLEVEL.ck = 0;
DLEVEL := 0;
begin_cycle.ck = 0;
begin_cycle := 0;

mxord3 = Measured_Level3 $ !Desired_Level3;

// Trim the gain when the Measured signal level is greater than the desired signal
level.
// Trim = MLEVEL > DLEVEL
Trim = (Measured_Level3 & !Desired_Level3)
    # ((Measured_Level2 & !Desired_Level2) & mxord3)
    # ((Measured_Level1 & !Desired_Level1) & mxord3 & (Measured_Level2 $
!Desired_Level2))
    # ((Measured_Level0 & !Desired_Level0) & mxord3 & (Measured_Level2 $
!Desired_Level2)
     & (Measured_Level1 $ !Desired_Level1));

// Boost the gain when the Measured signal level is less than the desired one.
meqd = (MLEVEL == DLEVEL);
Boost = !meqd & !Trim;

// Generate the chip select
!RTCcsn = ((address >= ^h0a00) & (address <= ^h0aff));

// Loading of the various registers
```

```
MLEVEL.ld = !clkin;

// State machine which controls the conversion start of the ADC, the interrupt to
the MCU,
// and the strobing of the IMCs

STATE_MACHINE.ck = clkin;
STATE_MACHINE.re = !reset;

state_diagram STATE_MACHINE;

state 0:
   Start_Conv = 0;
   Intrn = 1;
   if (begin_cycle == 1) then 1 else 0;
state 1:
   Start_Conv = 1;
   goto 2;
state 2:
   Start_Conv = 0;
   goto 3;
state 3:
   !Intrn = Trim # Boost; "Interrupt when Measured not equal to Desired
   goto 0;


Test_Vectors
// Test the state machine, trim, and boost signals
([clkin, reset, begin_cycle, MLEVEL, DLEVEL] ->
 [Start_Conv, Intrn, STATE1, STATE0, Trim, Boost])
 [ X, 0, X, ^h3, ^h4 ] -> [ X, X, X, X, 0, 1 ];"system in reset
 [ 0, 1, X, ^h4, ^h4 ] -> [ 0, 1, 0, 0, 0, 0 ];"system not in reset
 [ C, 1, 1, ^h5, ^h4 ] -> [ 1, 1, 0, 1, 1, 0 ];
 [ C, 1, 1, ^h5, ^h4 ] -> [ 0, 1, 1, 0, 1, 0 ];
 [ C, 1, 1, ^h5, ^h4 ] -> [ 0, 0, 1, 1, 1, 0 ];
 [ C, 1, 1, ^h4, ^h4 ] -> [ 0, 1, 0, 0, 0, 0 ];
 [ C, 1, 0, ^h4, ^h4 ] -> [ 0, 1, 0, 0, 0, 0 ];

end
```

APPENDIX B: STIMULUS FILE "*TUTOR8XX.STL*"

The *tutor8xx.stl* file consists of four sections:

1. Parameter Definitions: each of the M88 FLASH+PSD control registers has an I/O address (offset from the CSIOP base address). The parameters make the stimulus file easier to read.
2. User-defined tasks: these are used to define and implement the microcontroller bus cycles. In each task, the timing of the control signals and address or data bus should follow that of the microcontroller, but they do not have to be exact; they just have to be to scale. The PSD Simulator will simulate a bus cycle every time a **read**, **write**, or **psen** task is called.
3. Signal Initialization: you must specify the initial logic level of all the input signals before simulation. The output signals that you want to simulate should be initialized to a high impedance state.
4. The stimulus inputs: here the stimulus inputs are needed to perform MCU read or write bus cycles to access the Flash, EEPROM, SRAM or I/O ports. Inputs can also be generated to exercise the CPLD functions.

```
//Title:tutor8XX.stl
//Function:Simulation file for the M88x3Fxx Tutorial
//Designed by:Dan Harris
//Design Date:6-23-98
//Description:This file is intended to be used in the PSDsilosIII environment as a
//    stimulus file for the M88x3Fxx Tutorial. The idea of this file is not
//    to show how the Verilog-HDL language works, but rather the format of
//    a .stl file, and how it applies to this tutorial example.
//    The main parts of this file are:
//    * Parameter declarations which make the file more readable
//    * Read, write and "PSEN/" bus cycle tasks for the 80C31
//    * An area where the user may wish to add to the file in order to
//     test more functions
//    * The actual stimulus of the design

//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++
// Parameters declarations for the address offsets for the CSIOP address space
//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++

//Port A
parameter Port_A_Dir_Reg='h0906,Port_A_Cntl_Reg ='h0902;
parameter Port_A_Dout_Reg='h0904,Port_A_Din_Reg ='h0900;
parameter Port_A_IMC='h090A,Port_A_Drive_Sel = 'h0908;
parameter Port_A_En_Out='h090C;

//Port B
```

```
parameter Port_B_Dir_Reg=’h0907,Port_B_Cntl_Reg =’h0903;
parameter Port_B_Dout_Reg=’h0905,Port_B_Din_Reg =’h0901;
parameter Port_B_IMC=’h090B,Port_B_Drive_Sel = ’h0909;
parameter Port_B_En_Out=’h090D;

//Port C
parameter Port_C_Dir_Reg=’h0914,Port_C_En_Out =’h091A;
parameter Port_C_Dout_Reg=’h0912,Port_C_Din_Reg =’h0910;
parameter Port_C_IMC=’h0918,Port_C_Drive_Sel = ’h0916;

//Port D
parameter Port_D_Dir_Reg=’h0915,Port_D_Drive_Sel =’h0917;
parameter Port_D_Dout_Reg=’h0913,Port_D_Din_Reg =’h0911;
parameter Port_D_En_Out=’h091B;

//Port AB OMCs
parameter Port_AB_OMC=’h0920,Port_AB_OMC_Mask = ’h0922;

//Port BC OMCs
parameter Port_BC_OMC=’h0921,Port_BC_OMC_Mask = ’h0923;

//Other control registers
parameter FLASH_Protect=’h09C0,EEPROM_Protect = ’h09C2;
parameter PMMR0_Reg=’h09B0,PMMR1_Reg =’h09B2;
parameter PMMR2_Reg=’h09B4,JTAG_En = ’h09C4;
parameter Page_Reg=’h09E0,VM_Reg  =’h09E2;


//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++
// Defining tasks to simulate 80C31 bus cycles (read, write and psen bus cycles).
// Note that the cycles are shortened for simulation purposes, but the
functionality
// remains the same.
//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++

//The "write task" implements the 80C31 write bus cycle

task write;
input [15:0] addr_bus;
input [7:0] data_in;
```

```
begin
#20 ale = 1;//Latch the address lines
#20 adio = addr_bus;//Read the valid address (adio defined in .top file)
#20 ale = 0;//Ale inactive
#20 adio[7:0] = data_in;//Write operation
#40 wr = 0;//Write pulse
#100 wr = 1;//Write ends
#10 adio[7:0] = Z8;//Z16 defined in .top file
end

endtask


//The "read task" implements the 80C31 read bus cycle timing

task read;
input [15:0] addr_bus;

begin
#20 ale = 1;//Latch the address lines
#20 adio = addr_bus;//Read the valid address
#20 ale = 0;//Ale inactive
#20 adio[7:0] = Z8;//Float address bus (Z8 defined in .top)
#40 rd = 0;//Read pulse
#100 rd = 1;//Read ends
end

endtask


//The "psen task" implements the 80C31 psen program fetch bus cycle

task psen;
input [15:0] addr_bus;

begin
#20 ale = 1;//Latch the address lines
#20 adio = addr_bus;//Set-up the right address
#20 ale = 0;//Ale inactive
#20 adio[7:0] = Z8;//Float address bus
#40 psen = 0;//Read pulse
#100 psen = 1;//Read ends
end
```

```
endtask


//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++
// Define some busses here to make the program easier to read.
//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++

//adrout is the latched address output on Port A

reg [4:0] adrout;
reg Addr_Out4, Addr_Out3,Addr_Out2, Addr_Out1, Addr_Out0;
assign {Addr_Out4, Addr_Out3, Addr_Out2, Addr_Out1, Addr_Out0} = adrout;

reg [3:0] measured_value;
reg Measured_Level3, Measured_Level2, Measured_Level1, Measured_Level0;
assign {Measured_Level3, Measured_Level2, Measured_Level1, Measured_Level0} =
measured_value;

reg [3:0] desired_value;
reg Desired_Level3, Desired_Level2, Desired_Level1, Desired_Level0;
assign {Desired_Level3, Desired_Level2, Desired_Level1, Desired_Level0} =
desired_value;

reg [3:0] PGA_data;
reg PGA_Din3, PGA_Din2, PGA_Din1, PGA_Din0;
assign {PGA_Din3, PGA_Din2, PGA_Din1, PGA_Din0} = PGA_data;

reg [2:0] cntrl;
reg Control2, Control1, Control0;
assign {Control2, Control1, Control0} = cntrl;


//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++
// Stimulus starting point
//
// Initialize all the I/O first. Then proceed with the rest of the simulation.
```

```
//
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++

initial
begin

//Initialize the signals first
wr = 1; rd = 1;
reset = 0;adio = 'h0000;
ale = 0;psen = 1;
adrout = Z8;
measured_value = 'h0;
desired_value = 'h0;
PGA_data=Z4;cntrl = Z3;
Intrn = Z1;Start_Conv = Z1;
Trim = Z1;Boost = Z1;
JCEn = 1;
#100 reset = 1; // Take the PSD out of reset after 100ns


//We are now ready to do some configuration of the PSD

//Port A configuration
//Configure Port A, pins pa4 to pa0 to output the latched address, and the rest
//of the port will output control information in MCU I/O mode.

//Writing "1F" to the Port A control register enables latched address output on
//pins pa4 to pa0, and the rest of the port to output MCU I/O.
write(Port_A_Cntl_Reg, 'h1f);

//Writing "FF" to Port A's direction register sets up Port A pins to be outputs.
write(Port_A_Dir_Reg,'hff);

//Port B configuration
//Since there is no latched address output on Port B, and its control register
//defaults to MCU I/O mode output, only the direction register needs to be setup.
//Only pins pb3 to pb0 will be outputting data, and the rest will be receiving
//input
write(Port_B_Dir_Reg,'h0f);

//All of Port C is output (with the exception of the Vstby input
write(Port_C_Dir_Reg,'hfb);
```

```
//There is only one output on Port D (RTCcs/), so the direction register is
//setup as follows:
write(Port_D_Dir_Reg,'h04);

//Set up the mask registers so that only the desired portion of the OMCs get
//written. Only the desired value (MCELLAB[7:4]), and begin (MCELLBC7) can be
//written to.
write(Port_AB_OMC_Mask, 'h0f);
write(Port_BC_OMC_Mask, 'h7f);


//Write the EEPROM segment ees0, and the Flash segment fs1
//then read the SRAM

write('h0020,'h5a);//write 5a to ees0
write('h5A00,'ha5);//write a5 to fs1
read ('h07FE); //read the internal SRAM

// Wait, then initialize the gain to one and output the data on the pins
// pb2 to pb0.
#40 write(Port_B_Dout_Reg,'h01);


// Assume a small value for the output of the ADC since the gain is set
// to one
measured_value='h3;


// Load 5 into the desired value register
write(Port_AB_OMC, 'h50);


// Take the state machine out of the idle state and generate the ADC
// chip select.
#20 write(Port_BC_OMC, 'h80);



// Since the measured value is less than the desired one, the gain would
// be boosted after the interrupt was generated (3 cycles after the start
// of the state machine). The MCU should increment the gain by 1 at that
// time.
#400 write(Port_B_Dout_Reg, 'h02);
#10 $finish;
```

```
end

initial
begin

// Generate a 10 MHz system clock used by the state machine, etc.
// Note the time scale is set in the psdsoft.run file.
clkin=0;
forever #100 clkin=~clkin;

end //stimulus ends here
```

APPENDIX C: LIST OF M88 FLASH+PSD SIMULATION SIGNALS

Figure 61 gives a list of signals from the Explorer that can be viewed using the Data Analyzer. This list is based on the *tutor8XX.abl* file, and predefined signals. The list will vary depending on the names in your .abl file, but most of the signals will be the same.

Figure 61. List of Simulation Signals

AI03198

Any of the above signals can be dragged to the Data Analyzer window for viewing. Once there, the signals can be made into buses. For more information on the Explorer or Data Analyzer, see PSDsilosIII's on-line help, and the *PSDsilosIII User Manual*.

Table 3 contains all of the viewable predefined signal names, along with a brief description of each. The conventions used in the table are:

- n represents a number
- x represents a letter

See the list above to determine which letters or numbers apply to the respective signal.

Table 3. Table C1–Predefined Signal Names and their Descriptions

Signal/Bus Name	Description
VM	VM register
adioh[15:8]	Address/Data bus high byte
adiol[7:0]	Address/Data bus low byte
ctrl_x	Port x control register
data[7:0]	Non-multiplexed 8-bit data bus
din_x	Port x data in register
dirff_x	Port x direction register
dout_x	Port x data out register
drive_x	Port x drive register
ecsdn	External chip select output n
ee_boot_oe	EEPROM output enable
ee_power_down	EEPROM power down signal
ee_protection[3:0]	PSD security and EEPROM sector protection
ee_ready_busy_N	EEPROM ready/busy signal
ee_sdp_disable	EEPROM software data protection disable bit
ee_sdp_enable	EEPROM software data protection enable bit
ee_toggle	EEPROM toggle signal
eesel_f	EEPROM final chip select
enable_x	Enable to port x driver
f_protection[7:0]	Flash sector protection register (read only)
flash_oe	Flash output enable
flash_polling	Flash data polling bit
flash_ready_busy	Flash ready/busy signal
flash_toggle	Flash toggle bit
flsel_f	Flash final chip select
jtag	JTAG enable register
mask_mcab	Mask AB register outputs
mask_mcbc	Mask BC register outputs
mcellabn	Micro ∫Cell AB n output
mcellabn_clk	Output Micro ∫Cell AB n clock input

Signal/Bus Name	Description
mcellabn_pr	Output Micro ⌠Cell AB n preset input
mcellabn_reg	Output Micro ⌠Cell AB n register input
mcellabn_re	Output Micro ⌠Cell AB n reset input
mcellbcn	Output Micro ⌠Cell BC n output
mcellbcn_clk	Output Micro ⌠Cell BC n clock input
mcellbcn_pr	Output Micro ⌠Cell BC n preset input
mcellbcn_reg	Output Micro ⌠Cell BC n register input
mcellbcn_re	Output Micro ⌠Cell BC n reset input
nib_xn	Product term control port x[7:4] or x[3:0] input Micro ⌠Cell
out_mcab[7:0]	Output registers for Micro ⌠Cell AB
out_mcbc[7:0]	Output registers for Micro ⌠Cell BC
pxn	Port x, pin n
pxn_imc	Port x, Input Micro ⌠Cell n
pxn_oe	Port x, output enable n product term
pdn	Power down signal
pgr7_0	Page register outputs
pmmrn	Power management mode register n
pseln	Port n peripheral select
rd_bsy	PSD internal ready/busy status signal
sram_oe	SRAM output enable signal

APPENDIX D: DESIGN FILE FOR EPM7064S (U2 OF FIGURE 2)

```
-- Title:8XX Tutorial--Discrete Solution
-- Function:Replacement for the programmable logic portions of the M88x3Fxx
-- Designed by:Dan Harris
-- Design date:6/15/98
-- Description:This design shows what chip and logic would be required to replace
the
--    programmable logic portions of the M8813F1x. This chip will be responsible
--    for the following tasks:
--    * Latching the address generated by the 80C31 MCU.
--    * Decoding the address and generating internal/external chip selects.
--    * Storing control/status information in internal registers.
--    * Address translation for memory pageng.
--    * Interfacing to and controlling of the PGA in the Receiver circuit, and
the ADC
--     RTC, SRAM, EEPROM, FLASH, and MCU.
--    * Interfacing to a JTAG-compatible port for ISP.
-- Convention:The tilde (~) is used throughout this design to indicate active low
signals.

CONSTANT PAGE_REG_ADDR   = H"09E0";
CONSTANT VM_REG_ADDR    = H"09E2";
CONSTANT MCU_IO_OUT_ADDR  = H"0902";
CONSTANT DESIRED_REG_ADDR = H"0920";
CONSTANT GAIN_REG_ADDR   = H"0901";
CONSTANT START_SIG_ADDR  = H"0921";

subdesign 8XXtutor
(
-- The following signals are generated by the MCU (U1):
A/D[7..0]: BIDIR;-- Multiplexed address (lower byte)/data bus
A[15..8]: INPUT;-- Upper byte of the addr bus
RD~   : INPUT;-- Read strobe
WR~   : INPUT;-- Write strobe
ALE   : INPUT;-- addr latch enable signal
PSEN~ : INPUT;-- Program store enable

-- System-level inputs:
Reset~: INPUT;-- System reset
Clock : INPUT;-- System clock

-- The following signals are generated for the MCU (U1):
AGC_Interrupt~: OUTPUT;-- Interrupt the MCU when the desired and measured signal
levels don’t match
```

```
Trim  : OUTPUT;-- True when the measured level is greater than the desired one
Boost : OUTPUT;-- Opposite of Trim

-- The chip select output for the RTC (U5):
RTC_CS~: OUTPUT;

-- This signals are to/from the ADC (U7):
Start_Conversn: OUTPUT; -- Indicates when the ADC should start its analog-to-
digital conversion
ADC_Out[3..0]: INPUT; -- The measured signal strength

-- The bus is used to set the gain on the PGA (part of U8)
PGA_Din[2..0]: OUTPUT;

-- The following are outputs to the external memories:
-- Chip selects
FLASH_CS~: OUTPUT;
EEPROM_CS~: OUTPUT;
SRAM_CS~: OUTPUT;
-- Output enables
FLASH_OE~: OUTPUT;
EEPROM_OE~: OUTPUT;
SRAM_OE~: OUTPUT;
-- Upper address bits
FLASH_A[16..14]: OUTPUT;-- MS addr bits for the 128K FLASH - for segmentation
EEPROM_A[14..13]: OUTPUT;-- MS addr bits for the 32K EEPROM - for segmentation

-- Latched/demultiplexed address output
Addr_Out[7..0]: OUTPUT;-- outputs to the external memories

-- Control Output for MCU I/O mode
Control[2..0]: OUTPUT;
)

VARIABLE

A/D[7..0]: TRI;-- Needed to drive the data output onto the data bus
la[7..0] : LATCH;-- Must demux lower byte of addr
page_reg[7..0]: DFFE;-- Page register
vm_reg[7..0]: DFFE; -- Used for memory mapping in combined memory space mode
desired_reg[3..0]: DFFE;-- Register to store the desired signal level (set by the
MCU)
gain_reg[2..0]: DFFE; -- Register to store the gain level (set by the MCU)
begin_comparrison: DFFE; -- takes state machine out of idle state (s0)
```

```
cntrl_port_reg[2..0]: DFFE;-- MCU I/O mode control register
addr[15..0]: NODE;-- Demultiplexed addr
fs[7..0] : NODE;-- FLASH segment enable signals
ees[3..0]: NODE; -- EEPROM segment enable signals
swap     : NODE; -- bit 7 of the page register
enable_data_half: NODE;-- bit 6 of the page register
measured[3..0]: NODE;-- Output from the ADC
desired[3..0]: NODE;-- Input from the MCU
meqd     : NODE;-- True when the measured value equals the desired one
sm       : MACHINE WITH STATES (s0, s1, s2, s3);

BEGIN

-- Right now, there is nothing to output to the MCU on the A/D lines
 A/D[] = GND;

-- Latch in the addr[]
 la[] = A/D[];
 la[].ena = ALE;
 addr[7..0] = la[];
 addr[15..8] = A[];
 Addr_Out[] = la[];

 begin_comparrison = A/D7;
 begin_comparrison.clk = Clock;
 begin_comparrison.clrn = Reset~;
 begin_comparrison.ena = !WR~ & (addr[] == START_SIG_ADDR);

 desired_reg[] = A/D[7..4];
 desired_reg[].clk = Clock;
 desired_reg[].clrn = Reset~;
 desired_reg[].ena = !WR~ & (addr[] == DESIRED_REG_ADDR);

 gain_reg[] = A/D[2..0];
 gain_reg[].clk = Clock;
 gain_reg[].clrn = Reset~;
 gain_reg[].ena = !WR~ & (addr[] == GAIN_REG_ADDR);

 cntrl_port_reg[] = A/D[2..0];
 cntrl_port_reg[].clk = Clock;
 cntrl_port_reg[].clrn = Reset~;
 cntrl_port_reg[].ena = !WR~ & (addr[] == MCU_IO_OUT_ADDR);

 page_reg[] = A/D[];
```

```
 page_reg[].clk = Clock;
 page_reg[].clrn = Reset~;
 page_reg[].ena = !WR~ & (addr[] == PAGE_REG_ADDR);

 vm_reg[] = A/D[];
 vm_reg[].clk = Clock;
 vm_reg[].clrn = Reset~;
 vm_reg[].ena = !WR~ & (addr[] == VM_REG_ADDR);

 measured[] = ADC_Out[];
 desired[] = desired_reg[];
 PGA_Din[] = gain_reg[];
 Control[] = cntrl_port_reg[];

-- Memory Section
 swap = page_reg7;
 enable_data_half = page_reg6;

 fs0 = ((addr[] >= H"8000") & (addr[] <= H"BFFF") & (page_reg[] == 3) & !swap)
    # ((addr[] >= H"0000") & (addr[] <= H"3FFF") & swap);
 fs1 = (addr[] >= H"4000") & (addr[] <= H"7FFF");
 fs2 = (addr[] >= H"8000") & (addr[] <= H"BFFF") & (page_reg[] == 0);
 fs3 = (addr[] >= H"C000") & (addr[] <= H"FFFF") & (page_reg[] == 0);
 fs4 = (addr[] >= H"8000") & (addr[] <= H"BFFF") & (page_reg[] == 1);
 fs5 = (addr[] >= H"C000") & (addr[] <= H"FFFF") & (page_reg[] == 1);
 fs6 = (addr[] >= H"8000") & (addr[] <= H"BFFF") & (page_reg[] == 2);
 fs7 = (addr[] >= H"C000") & (addr[] <= H"FFFF") & (page_reg[] == 2);

 ees0 = ((addr[] >= H"0000") & (addr[] <= H"1FFF") & !swap)
    # ((addr[] >= H"8000") & (addr[] <= H"9FFF") & swap & !enable_data_half);
 ees1 = ((addr[] >= H"2000") & (addr[] <= H"3FFF") & !swap)
    # ((addr[] >= H"A000") & (addr[] <= H"BFFF") & swap & !enable_data_half);
 ees2 = (addr[] >= H"C000") & (addr[] <= H"DFFF") & swap & enable_data_half;
 ees3 = (addr[] >= H"E000") & (addr[] <= H"FFFF") & swap & enable_data_half;

 -- FLASH upper 3 and EEPROM upper 2 address bit encoding

 FLASH_A16 = fs7 # fs6 # fs5 # fs4;
 FLASH_A15 = fs7 # fs6 # fs3 # fs2;
 FLASH_A14 = fs7 # fs5 # fs3 # fs1;
 EEPROM_A14 = ees3 # ees2;
 EEPROM_A13 = ees3 # ees1;

-- Chip Selects and Output Enables
```

```
-- SRAM has highest priority, followed by EEPROM, and then FLASH
 !FLASH_CS~ = (fs0 # fs1 # fs2 # fs3 # fs4 # fs5 # fs6 # fs7) & (EEPROM_CS~ #
SRAM_CS~);
 !EEPROM_CS~ = (ees0 # ees1 # ees2 # ees3) & SRAM_CS~;
 !SRAM_CS~ = ((addr[] >= H"0100") & (addr[] <= H"08FF"));
 !RTC_CS~ = ((addr[] >= H"0A00") & (addr[] <= H"0A1F"));

 !SRAM_OE~ = !(!RD~ # (!PSEN~ & vm_reg0));
 !EEPROM_OE~ = !((!PSEN~ & vm_reg1) # (vm_reg3 & !RD~));
 !FLASH_OE~ = !((!PSEN~ & vm_reg2) # (vm_reg4 & !RD~));

-- Comparator I/O
 Trim = (measured[] > desired[]);
 meqd = (measured[] == desired[]);
 Boost = !Trim & !meqd;


-- State Machine
 sm.clk = Clock;
 sm.reset = !Reset~;

 CASE sm IS
   WHEN s0 =>
   Start_Conversn = GND;
    AGC_Interrupt~ = VCC;
    IF (begin_comparrison) THEN
      sm = s1;
    ELSE
      sm = s0;
    END IF;
   WHEN s1 =>
    Start_Conversn = VCC;
    sm = s2;
   WHEN s2 =>
    Start_Conversn = GND;
    sm = S3;
   WHEN s3 =>
    !AGC_Interrupt~ = Trim # Boost;-- Interrupt when Measured not equal to
Desired
    sm = s0;
 END CASE;

END;
```

APPENDIX E: COMPARING THE DISCRETE AND INTEGRATED SOLUTIONS

This appendix compares the two circuits of Figure 2 and Figure 3 in the following categories:

- Cost
- Average Current Usage
- Board Space Usage
- Time to market

(Only the major ICs are compared here.)

COST

The 150 ns M8813F1x in the PLCC package can be purchased at a significantly lower price than the total cost of the individual EEPROM, Flash, SRAM, and CPLD devices.

AVERAGE CURRENT USAGE

The M88x3Fxx would typically use 4.29 mA according to the “Example of M88x3Fxx Typical Power Calculation at V_{CC} = 5.0 V” in the “AC and DC Parameters” sections of the *M88 FLASH+PSD Data Sheet*. If we take the total average current of the devices in the discrete solution, we get 32.4 mA (with the EPM7064S *not* in turbo mode). This shows that the discrete solution uses 755% more current than the PLD.

BOARD SPACE USAGE

The M8813F1x in the PLCC package takes up 400 mm^2. The chips that make up the discrete solution take up a combined 1493 mm^2. That equates to 373% more board space. (All calculations have been based on PLCC packages.) This calculation does not reflect the extra board space, complexity, and noise associated with routing the signals in the discrete solution.

TIME TO MARKET

The time to market will be reduced significantly for many reasons:

- The integrated PSD solution involves one complex integrated circuit, not four.
- There are templates and predefined routines that, when used in conjunction with our user-friendly PSDsoft, help you with every step of your design process.
- Issues related to concurrent memory, memory mapping, and MCU assisted ISP are simplified.
- C code is generated for you.
- The JTAG interface is one of the greatest benefits and time savers. It allows you to program, configure, and test the entire PSD, and leave it soldered to the board the whole time.
- There are just fewer places to go wrong, and fewer things to debug when you have this level of integration.

APPENDIX F: SYSTEM MEMORY MAP AND UART ISP

INTRODUCTION

A system memory map was developed for this tutorial to take full advantage of the memory available in the M8813F1x, and to expand beyond the 64 KByte address space limitation of the 8031 MCU. This memory map facilitates the downloading of firmware from a host computer to the Flash memory in the PSD using the 8031 UART. The 8031 boots from the PSD EEPROM, concurrently downloads to PSD Flash memory, and then 8031 execution jumps from EEPROM to Flash memory. After this jump, the EEPROM in the boot area address space is replaced with Flash memory by a special register within the PSD (the VM Register). After that, the entire Flash memory is available to the 8031.

This system memory map also allows the concurrent downloading of boot code into the PSD EEPROM while executing code out of PSD Flash memory. This is not possible in non-PSD systems that use PROM for boot code.

The total memory available to the 8031 as defined in this system is:

- 128 KBytes Flash
- 16 KBytes EEPROM for boot code
- 16 KBytes EEPROM for data storage
- 2 KBytes battery-backed SRAM (in addition to the 256 bytes SRAM resident on the 8031)

SYSTEM MEMORY MAP

The system memory map is shown in Figure 62, Figure 63, Figure 64, and Figure 65. The labels FSx and EESx are the names of internal memory segments within the M8813F1x device. FSx represents 16 Kbyte Flash segments, EESx represents 8 Kbyte EEPROM segments.

In this design, paging is used because the system contains more memory than the 8031 can address linearly. The M8813F1x facilitates paging by using a page register, which the 8031 can access. Because paging is used, a common memory area is needed for firmware routines that must be accessible regardless of what page the MCU is executing from. This common area resides in the lower half of each memory page in program space (shown in Figure 62, Figure 63, Figure 64, and Figure 65). It should contain routines that handle initialization, interrupts, implement page switching, and drive physical devices. It is also used to keep critical data space items available at all times. For example, in this design, the PSD control registers, I/O, and system SRAM for the stack and global variables are available on any memory page (see Figure 62, Figure 63, Figure 64, and Figure 65).

There are two fundamental modes of operation: one is boot/download mode, and the other is normal operation. Figure 62, Figure 63, Figure 64, and Figure 65 show the memory map during the transition from boot/download mode to normal operation mode.

Figure 16 represents the memory map at power-on (boot). The system boots up from EEPROM, and then facilitate a download to the main Flash memory (if needed) using the 8031 UART. At this point, all of the PSD Flash memory is in the 8031 “data space” and all of the EEPROM is in the 8031 “program space”. This is due to the “MCU Bus Configuration” that was performed in step 2 of the section entitled “PSDsoft Configuration” (on page 20), and shown in Figure 16. This step of the configuration automatically sets the VM register to 12h. Please refer to the *M88 FLASH+PSD Data Sheet* for information on the VM register settings.

It is very important to note that the PSD Configuration utility initialises the VM register (located in the CSIOP space at offset E2h), and that it can only be changed by the MCU after it has booted. After the Flash has been programmed or validated, the Flash memory is moved from the 8031 data space to the

8031 program space by the MCU writing 06h to the VM register (while still executing out of PSD EEPROM).

Figure 62 represents the memory map after the Flash memory has been moved to the program space. This is an intermediate step that is a result of writing to the VM register.

Next, the 8031 execution jumps from PSD EEPROM to PSD Flash memory. While executing from PSD Flash memory, the 8031 sets a bit in the PSD page register that we call “SWAP”. The EEPROM that the MCU booted from, during power-up, is replaced with Flash memory that contains application vectors and code, as shown in Figure 64. The transition between the two maps of Figure 63 and Figure 64 is under the control of the 8031 by setting the “SWAP” bit inside the PSD (defined in the PSDabel, *tutor8XX.abl*, file). Again, the state of the memory map, shown in Figure 64, is an intermediate step.

Individual bits within the 8-bit PSD page register may be used for functions other than memory page definition. For example, in this tutorial, two of the eight PSD page register bits are use to define four memory pages, and one of the page register bits is used as the “SWAP” bit, as described above.

Finally, while executing from the PSD Flash memory, the 8031 must write 0Ch to the VM register in the PSD to move the PSD EEPROM from the 8031 program space to the 8031 data space. This finalizes the memory map, as shown in Figure 65. Now, all 128 KBytes of PSD Flash memory are in the program space, with 32 KBytes in a common area and 96 KBytes spread across three memory pages. Also, the EEPROM is now in the data space, and is accessible from any memory page. Notice that two more PSD EEPROM segments (EES2 and EES3) appear in Figure 65. These two segments are for general data use while the other two EEPROM segments (EES0 and EES1) contain the 8031 power-on boot code.

Now that the system memory map looks like that of Figure 65, another feature becomes available. Besides the mechanisms mentioned, there is one more memory mapping control bit used in this tutorial design. This bit, “ENABLE_DATA_HALF”, is another PSD page register bit used to protect the boot code in EES0 and EES1 from inadvertent writes. At the same time, it enables the other half of the EEPROM (EES2 and EES3) to be accessed for general data. For example, to update the boot code in EES0 and EES1 with new code downloaded over the UART, the 8031 would leave ENABLE_DATA_HALF at logic zero, perform the update by writing to EES0 and EES1, then set ENABLE_DATA_HALF to logic one. Now the new boot code is inaccessible (protected while not booting), and the data half of EEPROM is accessible.

Figure 62. System Memory Map for 8031-M8813F1x, boot/download POWER-UP (VM Register = 12h)

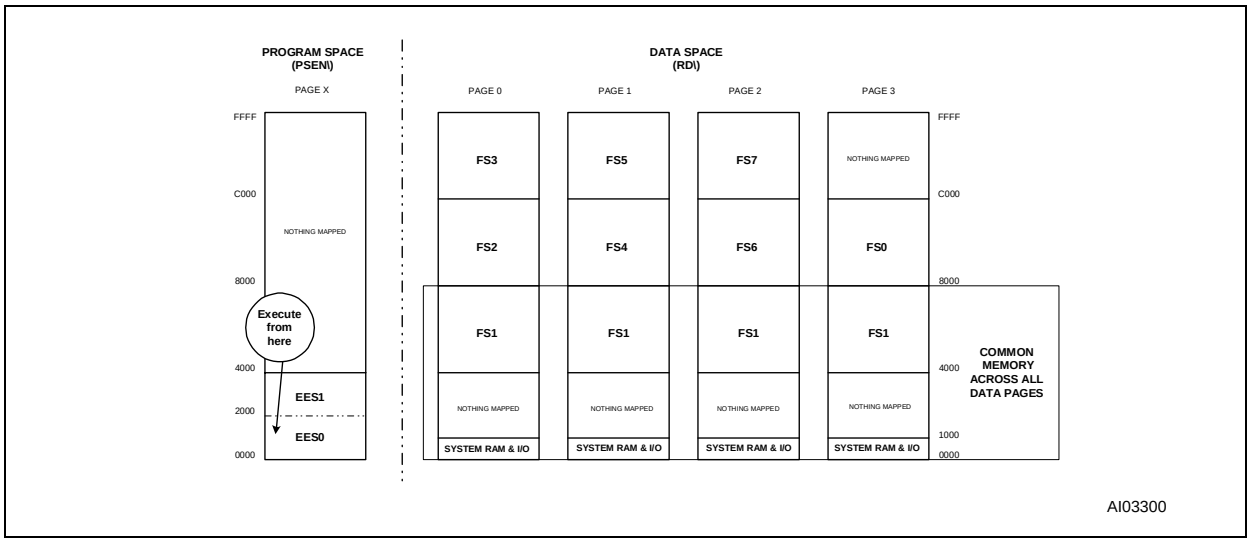


Figure 63. System Mem Map for 8031-M8813F1x, move Flash to program space WRITE 06h TO THE VM REGISTER

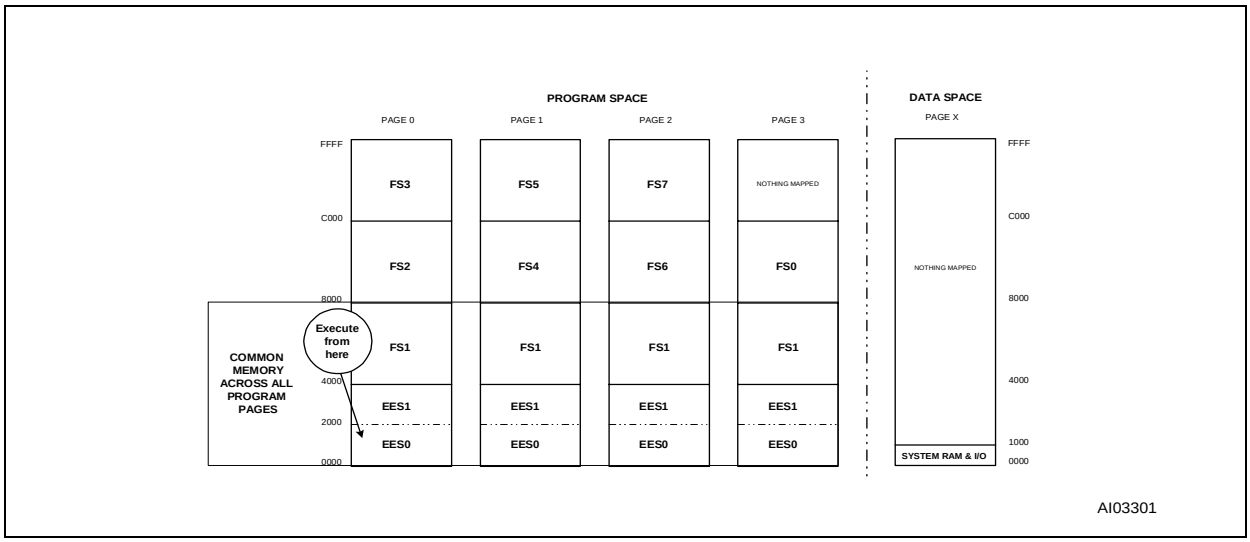


Figure 64. System Memory Map for 8031-M8813F1x, swap boot EEPROM with Flash segment SET SWAP BIT = 1

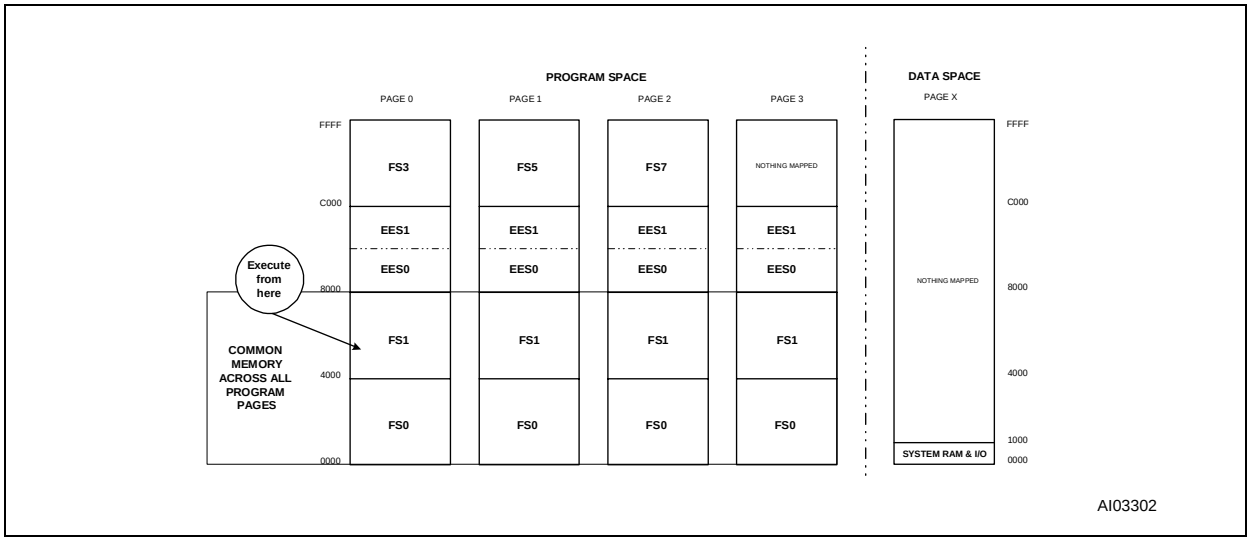
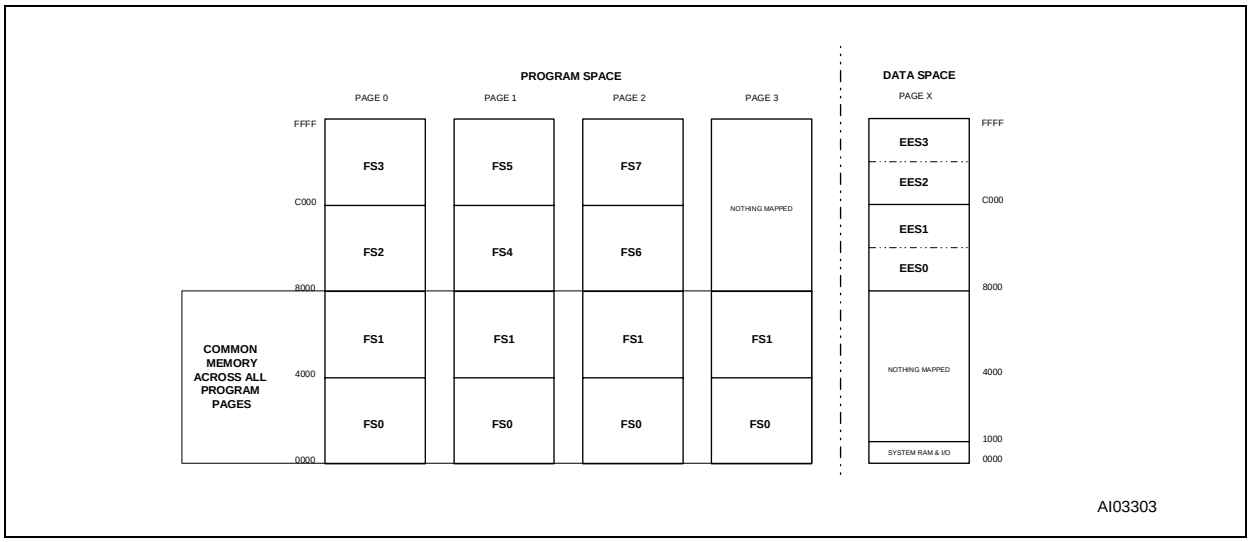


Figure 65. Final Sys Mem Map for 8031-M8813F1x, move EEPROM to data space WRITE 0Ch TO THE VM REGISTER



CODE PARTITIONING IN THE FLASH MEMORY PAGES

Ultimately, the MCU will be executing from Flash memory since the EEPROM is used for boot-up and ISP in this design. Let us assume that we have 128 KBytes of program space in Flash memory, as shown in Figure 65. The 128 KBytes of code resides in four areas: 32 KBytes in the common area (FS0 and FS1, accessible from any page), 32 KBytes on page zero (FS2 and FS3), 32 KBytes on page one (FS4 and FS5), and 32 KBytes on page two (FS6 and FS7).

If the 8031 never leaves page zero while executing, it can access 64 KBytes of Flash memory in FS0 through FS3 as well as all of the SRAM and I/O. If the 8031 execution jumps to Flash memory on pages one or two from a call on the upper half of page zero (FS2 or FS3), care must be taken to leave a path to return to page zero again. However, if the call to page one or two is from a routine in the lower half of page zero (the common area, FS0 or FS1), there is no problem returning from the call.

When placing code in the Flash memory on the upper half of pages zero, one, or two, the software designer must break tasks into logical groups. These groups should not need to access code frequently on other pages. (Most software can be split in this manner and is a result of a good modular design.) Since system SRAM is available on any page, firmware routines that reside on different pages may pass data using global variables or the stack. The designer can create page-switching algorithms to jump between tasks on different pages. There are many ways to implement a paging scheme: one method involves the use of a table that contains addresses and page numbers of all program tasks, which may be called from page to page. The table and algorithms must reside in the portion of Flash memory that is located in the common area. This provides a very clean paging solution, which may be implemented using a high-level compiler. (The compiler from Keil supports this directly, and creates the tables for you.) The only penalty when using this method is the overhead experienced when switching from one page to another.

For this tutorial design, five different files from an MCU cross-compiler and linker are used to program the NVM sections of the M8813F1x. These are dummy files with no code in them, but are present to illustrate the merging of MCU firmware with the PSD configuration during the Address Translate operation. If this were a real design, the file *common.hex* would contain all of the common functions and interrupt vectors, and would be programmed into FS0/FS1. Three more files from the MCU linker, *page_0.hex, page_1.hex,* and *page_2.hex* would contain the partitioned code described above. As such, these three files would be programmed into segments FS2/FS3, FS4/FS5, and FS6/FS7, respectively. Finally, the file *boot.hex*,

containing the power-up boot code and programming algorithms for Flash memory, would be programmed into EES0/EES1.

START-UP SEQUENCES, UART DOWNLOADS

Let us assume that a PC or lap-top is to be used as a host to download firmware to this embedded system over an RS-232 UART channel (instead of JTAG). These download actions can program the main Flash memory for the very first time; can update the main Flash after it has been programmed once; or can update the boot code after being programmed for the first time by a device programmer or JTAG link.

There are six valid boot-up arrangements (labelled respectively: a, b, c, d, e and f) that must be handled by the system at power-up (reset). The default setting of the VM register at power-up places the main Flash memory in the data space and the EEPROM in the program space. Please refer to the memory maps in Figure 62, Figure 63, Figure 64, and Figure 65.

a. RS-232 cable not attached, main Flash valid

```
8031 action:
   Boot from EES0/EES1
   Run a checksum on the Flash memory
   Check the UART for a pending host download request of main Flash (Figure 62)
   Set a bit in the PSD VM register to put main Flash into program space (Figure 63)
   Set the SWAP bit in PSD, which swaps EES0/EES1 with FS0 (Figure 64)
   Set a bit in the PSD VM register to put the EEPROM into data space (Figure 65)
Now, the system is in normal operating mode. More 8031 action:
   Check the UART for a host download request of boot memory
   Set the ENABLE_DATA_HALF bit in the PSD if no boot download request exists
Normal application code can now be executed from main Flash memory.
```

b. RS-232 cable attached, main Flash valid, no download demands from host

Action: same as step "a.", above.

c. RS-232 cable attached, main Flash valid, download of main Flash is demanded by host

```
8031 action:
   Boots from EES0/EES1
   Run a checksum on the Flash memory
   Check the UART for a pending host download request of main Flash (Figure 62)
   Program the main Flash memory with data from the UART
   Set a bit in the PSD VM register to put main Flash into program space (Figure 63)
   Set the SWAP bit in PSD, which swaps EES0/EES1 with FS0 (Figure 64)
   Set a bit in the PSD VM register to put the EEPROM into data space (Figure 65)
Now, the system is in normal operating mode. More 8031 action:
   Check the UART for a host download request of boot memory
   Set the ENABLE_DATA_HALF bit in the PSD if no boot download request exists
Normal application code can now be executed from main Flash memory.
```

d. RS-232 cable not attached, main Flash is blank or invalid

8031 action:

Boot from EES0/EES1
Run a checksum on the Flash memory
Check the UART for a pending host download request of main Flash (Figure 62)
Wait until any UART traffic is present (Figure 62)

e. RS-232 cable attached, main Flash is blank or invalid

8031 action:

Boot from EES0/EES1
Run a checksum on the Flash memory
Check the UART for a pending host download request of main Flash (Figure 62)
Program the main Flash memory with data from the UART
Set a bit in the PSD VM register to put main Flash into program space (Figure 63)
Set the SWAP bit in PSD, which swaps EES0/EES1 with FS0 (Figure 64)
Set a bit in the PSD VM register to put the EEPROM into data space (Figure 65)

Now, the system is in normal operating mode. More 8031 action:

Check the UART for a host download request of boot memory
Set the ENABLE_DATA_HALF bit in the PSD if no boot download request exists

Normal application code can now be executed from main Flash memory.

f. RS-232 cable attached, main Flash is valid, system requests a download of boot memory

8031 action:

Boot from EES0/EES1
Run a checksum on the Flash memory
Check the UART for a pending host download request of main Flash (Figure 62)
Set a bit in the PSD VM register to put main Flash into program space (Figure 63)
Set the SWAP bit in PSD, which swaps EES0/EES1 with FS0 (Figure 64)
Set a bit in the PSD VM register to put the EEPROM into data space (Figure 65)

Now, the system is in normal operating mode. More 8031 action:

Check the UART for a host download request of boot memory
Program the EEPROM boot memory in EES0 and EES1 with data from the UART
Run a checksum on EES0 and EES1
Set the ENABLE_DATA_HALF bit in the PSD
to protect the boot code in EES0 and EES1 from inadvertent writes
Enable data access of EES2 and EES3

Normal application code can now be executed from main Flash memory.

For of any of these host UART download options, it is assumed that the normal boot (EES0/EES1) area is programmed the very first time by a device programmer before the PSD is installed on the circuit card or by the JTAG interface while the PSD is in-system.

For current information on M88 FLASH+PSD products, please consult our pages on the world wide web:
www.st.com/flashpsd

If you have any questions or suggestions concerning the matters raised in this document, please send them to the following electronic mail addresses:

apps.flashpsd@st.com (for application support)
ask.memory@st.com (for general enquiries)

Please remember to include your name, company, location, telephone number and fax number.