AN1356
APPLICATION NOTE

PSDsoft Express and PSD4235G2 Design Guide

CONTENTS

*Easy*FLASH™ PSD4X35G2 devices are members of a family of Flash memory-based peripherals for use with embedded microcontrollers (MCUs) or microprocessors (MPUs). These Programmable System Devices (PSDs) consist of memory, logic, and I/O. When coupled with a low-cost, ROM-less MCU/MPU, the PSD forms a complete embedded Flash memory system that is 100% In-System-Programmable (ISP). There are many features in the PSD silicon and in the PSDsoft *Express*™ development software that make ISP easy for you, regardless of how much experience you have in embedded Flash memory design.

This document offers three designs using an ST PSD4235G2 and a Philips P51XA MCU. Note that a variety of 16-bit MCU/MPUs can be used in place of the Philips part. Although the specifics of this document are based on the P51XA-G30, this document can be used as a guide for other MCU/MPU applications. The first design is a simple system to get up and running quickly for basic applications, or to check out prototype hardware. The second design illustrates the use of concurrent memory operation for field-updates and includes the use of programmable logic. The third design highlights advanced concurrent memory operation. You can start with the first design and migrate to the second and third as your requirements grow. Another member of the PSD4X35G2 family, the PSD4135G2, is a lower cost device with a subset of features of the PSD4235G2. See data sheets and *AN1426* for details.

In-System Programming and In-Application re-Programming

Our industry uses the term In-System Programming, or ISP, in a general sense. ISP is applicable to programmable logic, as well as programmable Non-Volatile Memory (NVM). An additional term is used in this document: In-Application re-Programming (IAP). There are subtle yet significant differences between ISP and IAP when microcontrollers are involved. ISP of memory means that the MCU is off line and not involved while memory is being programmed. IAP of memory means that the MCU participates in programming memory, which is important for systems that must be online while updating firmware. Often, ISP is well suited for manufacturing, while IAP is appropriate for field updates. PSD4X35G2 devices provide

Rev. 01A

both ISP and IAP. Keep in mind that IAP can only program the memory sections of the PSD, not the configuration and programmable logic portions. ISP can program all areas of the PSD.

The IAP Problem

Typically, a host computer downloads firmware into an embedded Flash memory system through a communication channel that is controlled by the MCU. This channel is usually a UART, but any communication channel that the MCU supports will do (CAN, MODEM, USB, J1850, etc). The MCU must execute the code that controls the IAP process from an independent memory array that is not being erased or programmed. Otherwise, boot code and Flash memory programming algorithms (IAP loader code) will be unavailable to the MCU. It is absolutely necessary to use an alternate memory array (an independent memory that is not being programmed) to store the IAP loader code.

A system designer must choose the type of alternate memory to store IAP loader code (ROM, SRAM, Flash memory, or EEPROM) as each type has advantages and disadvantages. This alternate memory may reside external to the MCU or on-board the MCU. A top-level view of an embedded IAP Flash memory system with external memory is shown in Figure 1.

**Figure 1. Embedded Flash memory system capable of IAP (5 devices)**
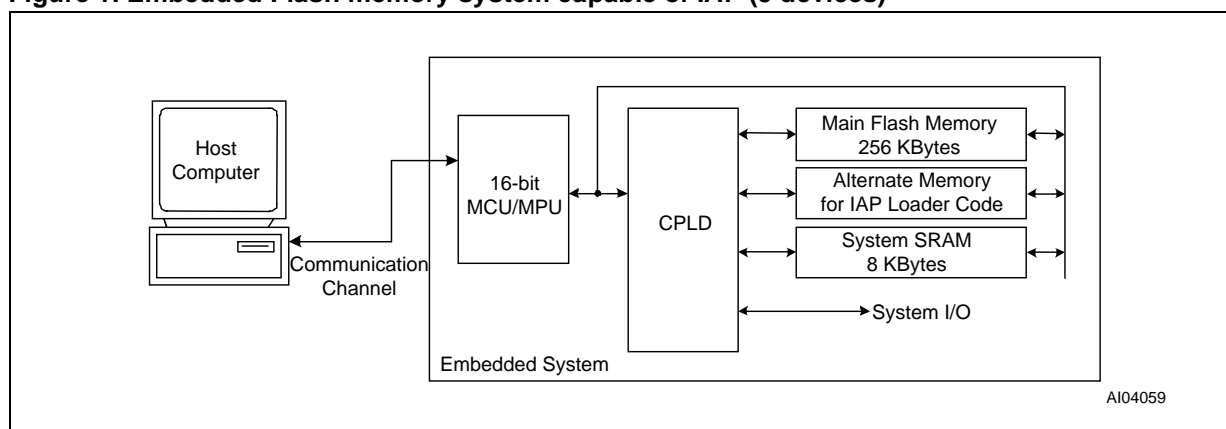
A Common Solution

Without a PSD device, implementing IAP with the P51XA and most other 16-bit MCUs can be difficult, expensive, and time consuming. Many P51XA designers will use external or internal PROM to implement a boot-loader using the P51XA UART to download code from a host computer into P51XA SRAM (Philips application note AN97019). P51XA execution then jumps to the SRAM to execute the remainder of the download process to program Flash memory. This is a cumbersome and error prone exercise using re-locatable code in volatile memory which is difficult to debug, vulnerable to power outages, and not supported by all emulators. Additionally, it is an expensive task to update the IAP loader code that is stored in PROM.

A Better, Integrated Solution

Figure 2 shows a two-chip solution using an *Easy*FLASH PSD4235G2. This system has ample main Flash memory, a second alternate Flash memory to hold the IAP loader code and general data, and more SRAM. All three of these memories can operate independently and concurrently; meaning the MCU can operate from one memory while erasing/writing the other. This allows the MCU to continue normal operation (at possibly a reduced level) during IAP, which is crucial for some applications. This system also has programmable logic, expanded I/O, and design security. The two-chip solution is 100% programmable in the factory or in the field.

Figure 2. Embedded Flash memory system capable of ISP and IAP (2 devices)
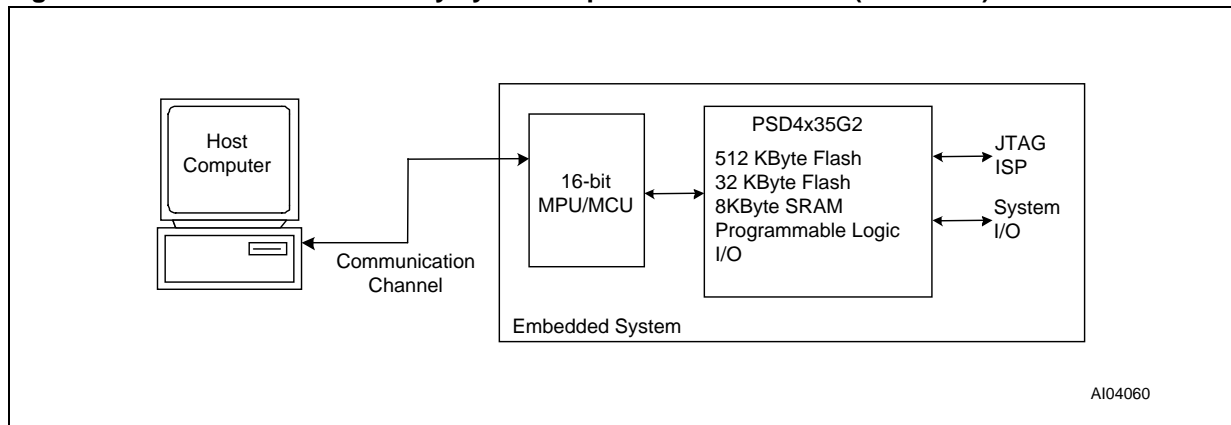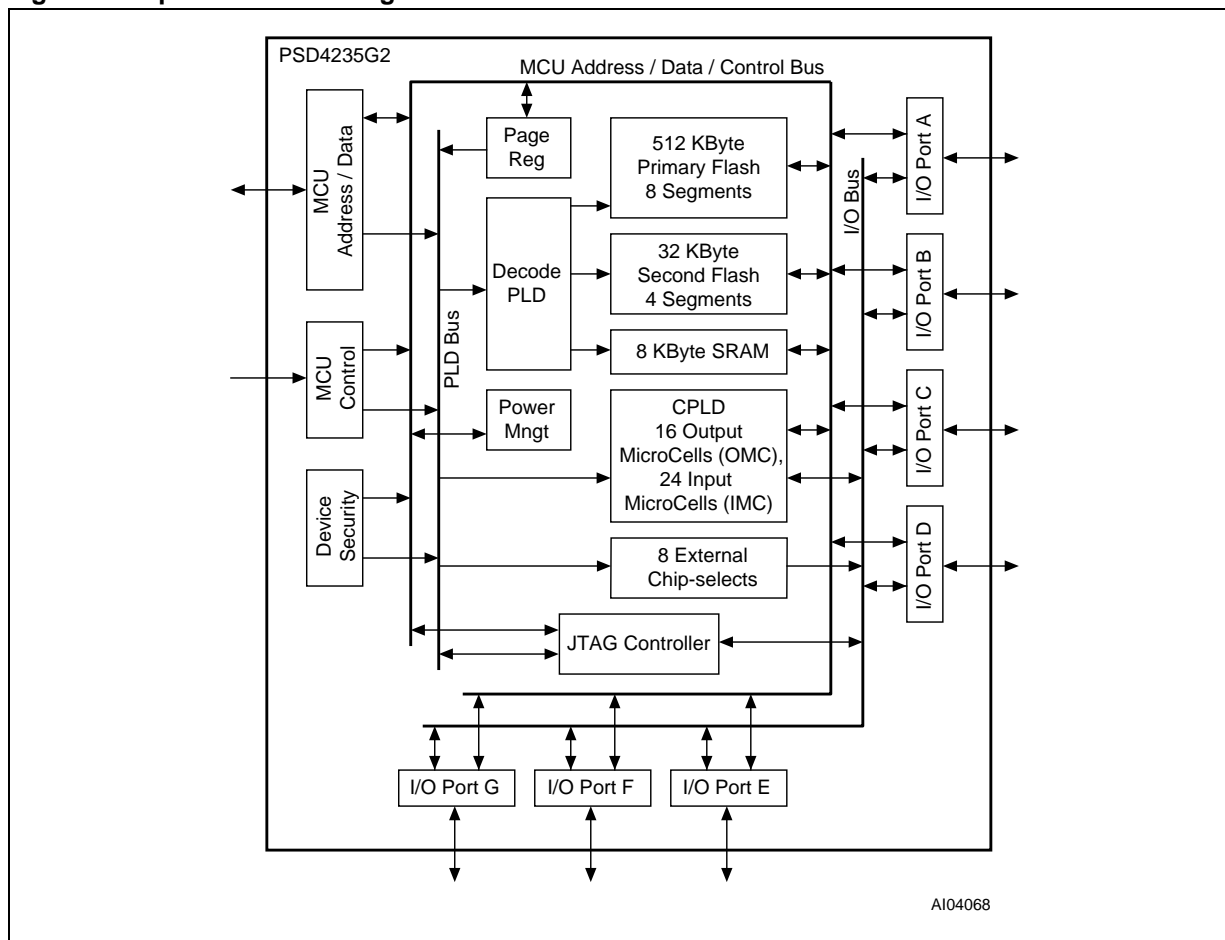

By design, the IAP method described above requires MCU participation to exercise a communication channel to implement a download to the main Flash memory. The PSD4235G2 also offers an alternative method (ISP) to program the PSD using a built-in IEEE-1149 JTAG interface requiring no MCU participation. This means that a completely blank PSD can be soldered into place and the entire chip can be programmed in-system in just a few seconds using ST's FlashLINK™ JTAG cable and PSDsoft development software. No P51XA firmware needs to be written, just plug in the FlashLINK™ cable to your PC parallel port and begin programming memory, logic, and configuration. This is a powerful feature of the PSD4235G2 that allows immediate development of application code in your lab, smart manufacturing techniques, and easy field updates.

PSDsoft *Express* is available from our website. The availability of the FlashLINK™ cable is also detailed there.

Let's take a quick look inside the EasyFLASH™ PSD4235G2, as shown in Figure 3. You can see the three independent memory arrays, which are selected on a segment basis when the proper MCU address is decoded in the Decode PLD. The page register participates in memory decoding, which greatly simplifies paging. The MCU address, data, and control signals are routed throughout the chip and can be used within the Complex PLD (CPLD). The CPLD has 16 Output Microcells (OMCs), each containing a flip-flop and combinatorial logic. The CPLD also has 24 Input MicroCells (IMCs) used for conditioning incoming signals. The MCU has direct memory-mapped access to both OMCs and IMCs. Additionally, the CPLD contains 8 programmable external chip-select outputs. There are 52 I/O pins that can be individually configured for many different functions. A power management scheme can selectively shut down parts of the chip and tailor special power saving mechanisms on-the-fly. The security feature can block access to all areas of the chip from a device programmer/reader. Finally, the self-contained JTAG-ISP controller allows programming of all areas of the chip.

In the second design example of this document, you will see how to use the CPLD to implement a loadable counter, a state machine, combinatorial logic, and other functions using OMCs, IMCs, the page register, and external chip-selects.

**Figure 3. Top Level Block Diagram of PSD4235G2**

PSD4235G2

MCU Address / Data / Control Bus

MCU Address / Data

MCU Control

Device Security

PLD Bus

Page Reg

Decode PLD

Power Mngt

512 KByte Primary Flash 8 Segments

32 KByte Second Flash 4 Segments

8 KByte SRAM

CPLD 16 Output MicroCells (OMC), 24 Input MicroCells (IMC)

8 External Chip-selects

JTAG Controller

I/O Bus

I/O Port A

I/O Port B

I/O Port C

I/O Port D

I/O Port G   I/O Port F   I/O Port E

AI04068

PHYSICAL CONNECTION

Connect your P51XA to the PSD4235G2 as shown in Figure 4. The JTAG programming channel, LCD module, system I/O, MCU I/O signals, and battery back up are optional. They are present in this application note to illustrate PSD functions.

There are four unused PSD I/O pins in this example. Unused pins should be pulled to Vcc with a 100K resistor or tied to GND. Also, see Application Note 54 for more information on the JTAG-ISP connection options.

Figure 4. Physical Connections, P51XA and PSD4X35G2

AI04071
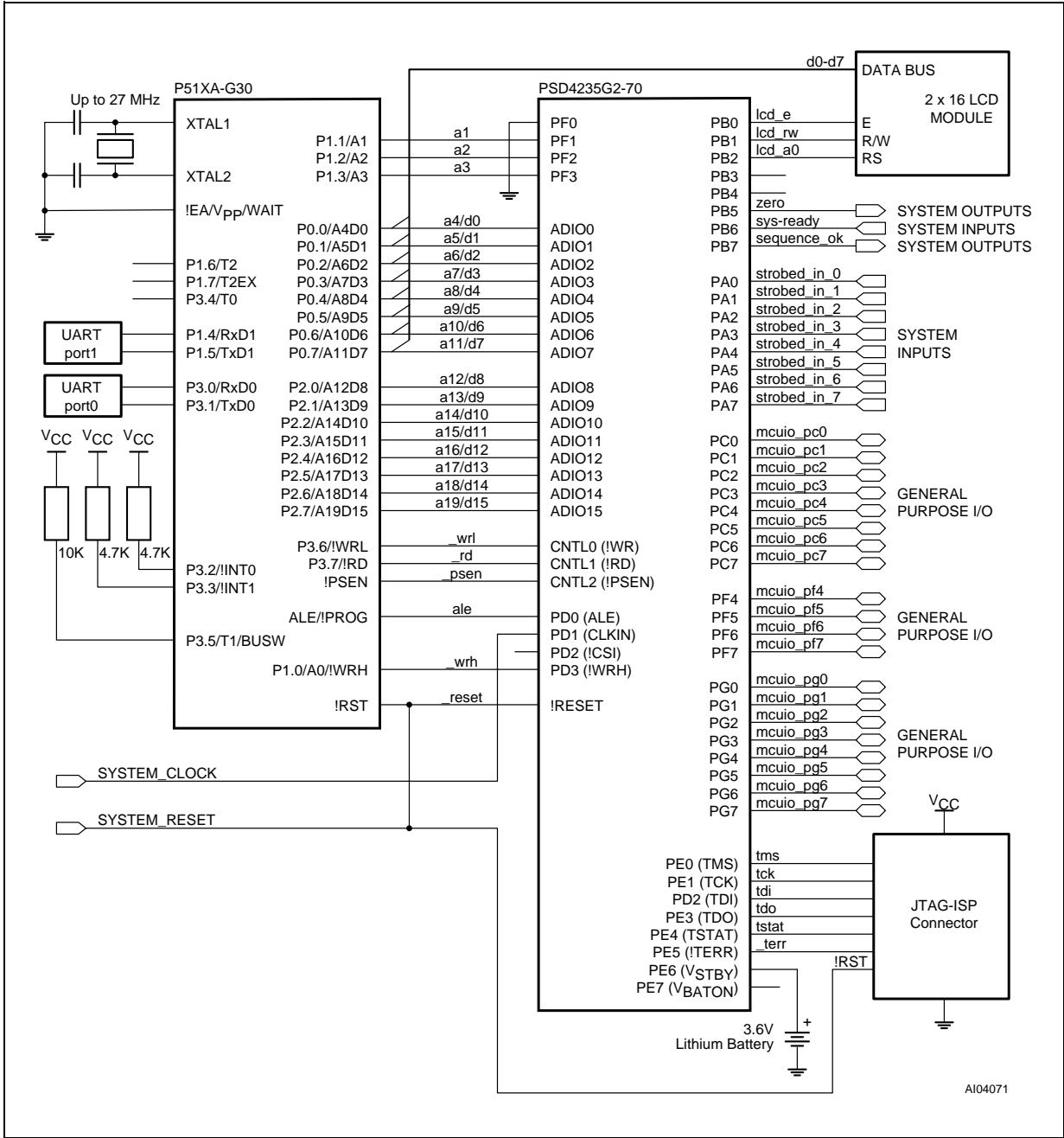
Note: Pullup (100K) or ground all unused inputs.
P51XA internal bus control register settings for 70ns

FIRST DESIGN EXAMPLE - ISP CAPABLE SYSTEM, LIMITED IAP

The first design example is capable of ISP and limited IAP. It outlines the steps required to get a Flash memory P51XA system up and running quickly. The 32 KBytes of PSD secondary Flash memory will be programmed with P51XA firmware (over the JTAG-ISP channel) that will execute low-level system hardware tests. This firmware is also able to access 512 KBytes of main PSD Flash memory, used as data only — not program space. This provides a way to develop code to erase and write to main PSD Flash memory while executing from secondary Flash memory. The second and third design examples take full advantage of concurrent memory operation and IAP, by allowing program execution from main Flash memory in addition to writing to it. You should become familiar with this first design before using the second and third.

Memory Map

For this first simple design, a PSD4235G2 is used with the following memories:

■ 512 Kbytes main Flash memory, broken into eight 64 Kbyte segments denoted $fs_{i\ (i\ =\ 0-7)}$

■ 32 Kbytes secondary Flash memory, broken into four 8 Kbyte segments denoted $csboot_{j\ (j\ =\ 0-3)}$.

■ 8 Kbyte SRAM denoted rs0

■ 256-byte PSD4235 control registers denoted csiop.

Note: PSD memory segment address locations are defined using PSDsoft *Express*™.

We'll use the PSD's secondary Flash memory to hold the boot code, P51XA interrupt vectors, hardware drivers, and common functions including routines that erase/program main PSD Flash memory. For this example, we'll execute from the PSD's secondary Flash memory only and use the PSD's main Flash memory as data. See the memory map in Figure 5.

Figure 5. Memory Map: Simple P51XA/PSD4235G2 Design
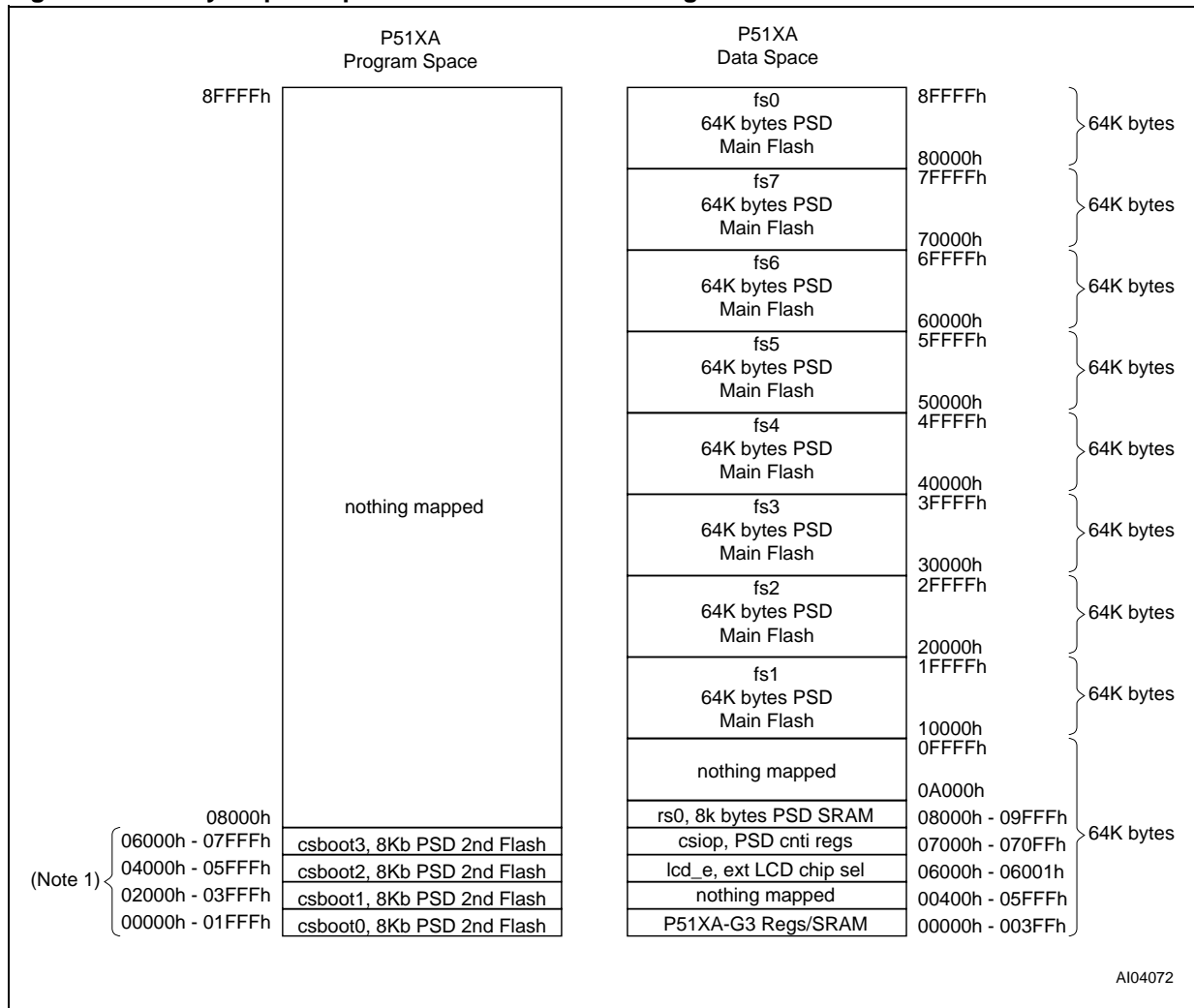
Note: 1. P51XA firmware gets programmed here by JTAG-ISP or conventional programmer tool.

PSDsoft *Express* Design Entry

Highlights of design entry are given here. Follow along using PSDsoft *Express* if you wish.

Open a New Project

■ Invoke PSDsoft *Express*.

■ Create a new project.

■ Select your project folder and name the project (in this example, name the project "simpleXA" in the folder PSDexpress\my_project).

■ Select an MCU. In this example, we're using a Philips P51XAG3x.

■ Select /WRL, /RD, /PSEN, /WRH, Burst Mode for the control signals.

■ Select the PSD4000 series for the PSD Family.

■ Select a PSD4235G2 and use the 80-pin TQFP package (U package).

■ Based on the above selections, the MCU bus will be automatically set to 16-bits multiplexed.

■ Select the main PSD Flash memory to reside in Data space upon power-up.

■ Select the secondary PSD Flash memory to reside in Program space upon power-up.

The selection of Program space or Data space for the Flash memories determines whether or not the P51XA signals, PSEN or RD respectively, will activate the output enables of the individual PSD Flash memory arrays upon power up. You will learn in the second and third designs that this setting can be changed by MCU firmware at runtime to implement IAP. Note that this applies only to MCUs with the Harvard architecture (separate address spaces for code and data). For MCUs with Von Nueman architecture (a single linear address space for code and data), the menu choice for Program and Data space does not apply and does not appear.

Now you have your project established based on a PSD4235G2 and a P51XAG3. The PSD will be compatible with the "burst mode" feature, unique to the P51XA, meaning that the special use of the lower four non-multiplexed address bits (a0..a3), the shifting of the upper address bits (a4..a19), and opcode reads with no ALE pulse are automatically supported by the PSD.

Although this document uses the Philips P51XA as a detailed example, the methods and examples within are very similar for other MCU/MPUs. PSD silicon adapts to many different MCU/MPU interfaces automatically based on selections in PSDsoft.
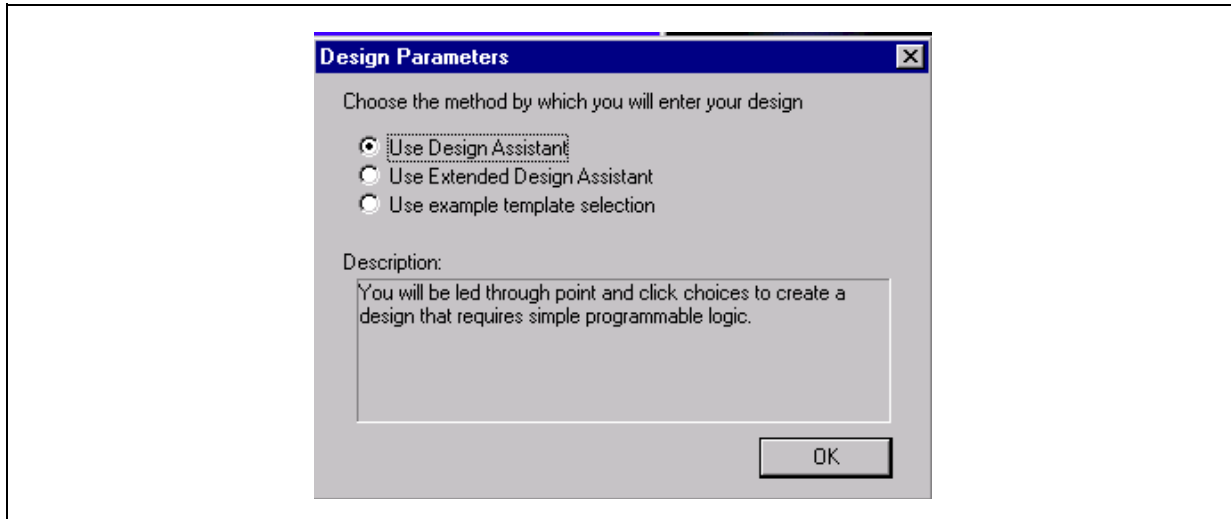
MCU and PSD Selection

This is what the screen should look like after you've made the selections:

Figure 6. MCU and PSD Selection
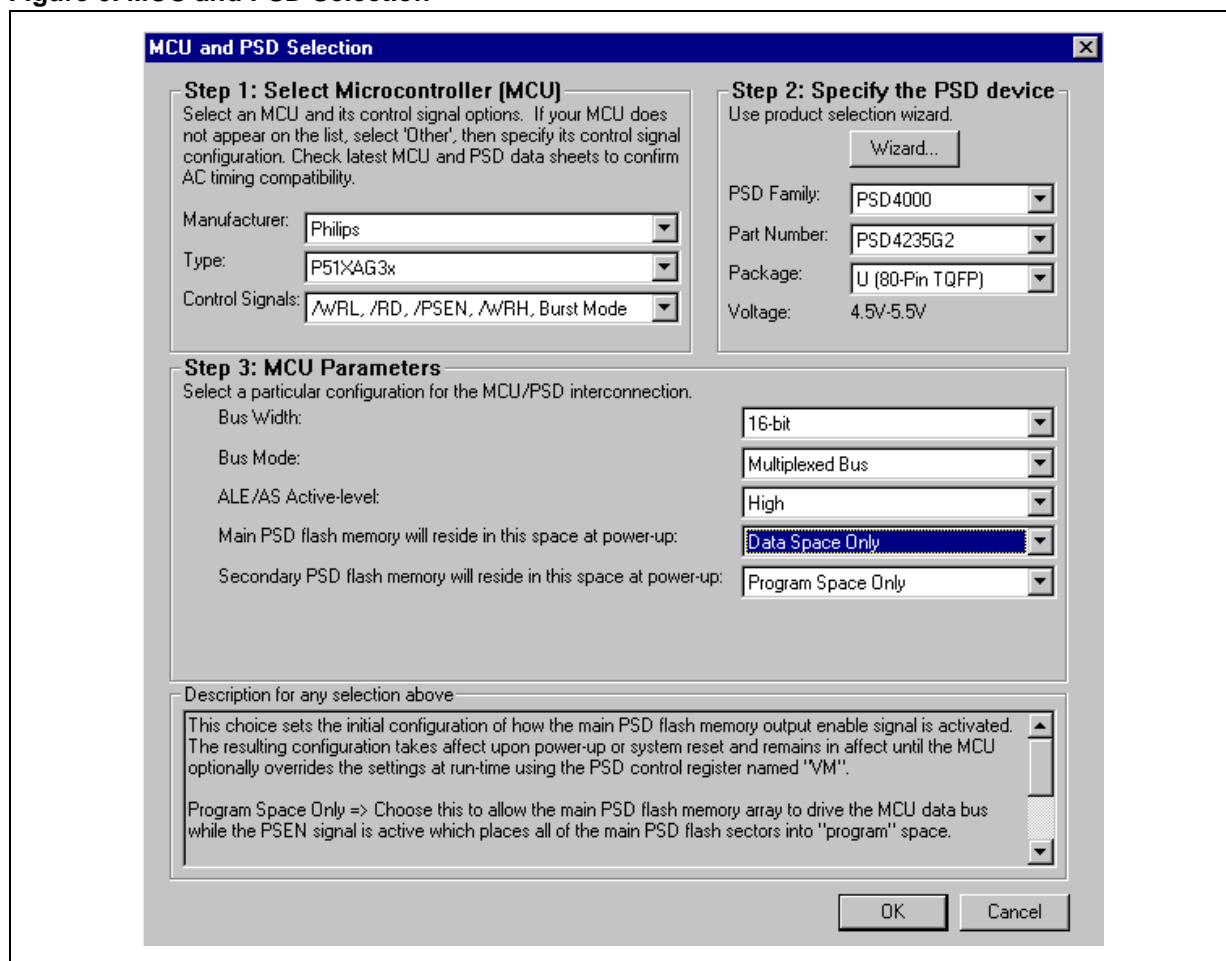
Click **OK**. Now you will be asked if you want to use the Design Assistant, Extended Design Assistant, or a example template as shown:

Figure 7. Design Assistant Selection

Choose Design Assistant. This will help you become familiar with most of the flow of PSDsoft *Express*. We'll use the Extended Design Assistant in the second design example.

For any of the three choices, ABEL HDL statements are automatically generated for you behind-the-scenes based on your point-and-click design entry. These statements include pin, node, and signal declarations as well as logic equations.

The **Design Assistant** choice does not allow editing access[*] to these generated ABEL statements, which is typically not necessary for simpler designs.

It is sometimes necessary to edit or add statements to the generated ABEL file for more complicated PLD designs that use counters, shift registers, state machines, etc. In these cases, the **Extended Design Assistant** should be chosen, allowing you to add ABEL statements in designated sections of the generated file that will not be affected by subsequent design iterations in the point-and-click entry environment. You will learn how to do this in design number two.

In future designs you may choose to use a pre-defined **example template**, which will make many of the choices for you based on your selection of MCU and PSD — you just have to tailor the template to fit your design. But again, there is no ability to edit HDL language statements[*]. Use this mode to get a suggested memory map from ST based on the MCU/MPU and PSD combination that you have chosen. Note that not all MCU/MPU selections will produce a choice for a pre-defined example template in which case only two choices will be available: the Design Assistant and Extended Design assistant.

[*]At a later point in your design cycle, regardless of which of the three methods you have chosen, you may optionally "turn on" the ability edit ABEL equations. We'll see how to do this in the second design example. Note that this is not available for the PSD9XXF and PSD4135G2 devices that have a simple PLD section.

Pin Definitions

Next you are taken to the "Pin Definitions" screen, which allows you to define each PSD pin function one-by-one on a point-and-click basis. Notice that the PSD pins connecting to the MCU are already defined for you because their function is fixed. For this first simple design, you need only to define a few pins that are listed below. In the second design example we use all of the signals shown in the schematic of Figure 4.

■ Define an active-high chip select output on Port B pin pb0. Choose **External chip select – Active-HI** from the CPLD Output section and name it "lcd_e". Click **Add**.

■ Define a combinatorial CPLD output on Port B pin pb1. Choose **Combinatorial** from the CPLD Output section and name it "lcd_rw". Click **Add**.

■ Define a combinatorial CPLD output on Port B pin pb2. Choose **Combinatorial** from the CPLD Output section and name it "lcd_a0". Click **Add**.

■ Define an additional JTAG-ISP pin on Port E pin pe4. Choose **Dedicated JTAG – TSTAT** from the Other section. Click **Add**. Notice the name "tstat" is automatically included. Also notice that the signal "_terr" is automatically added to Port E pin pe5. These two signals work together as a pair to reduce JTAG-ISP programming time by 10% – 15%. See application note 54 on our web site for details.

■ Define a pin to accept a battery voltage input for PSD SRAM on Port E pin pe6. Choose **SRAM standby voltage input** from the Other section. Click **Add**. The name "vstby" is automatically included.

Your Pin Definition screen should now look like the screen capture below:

**Figure 8. Pin Definition Screen**
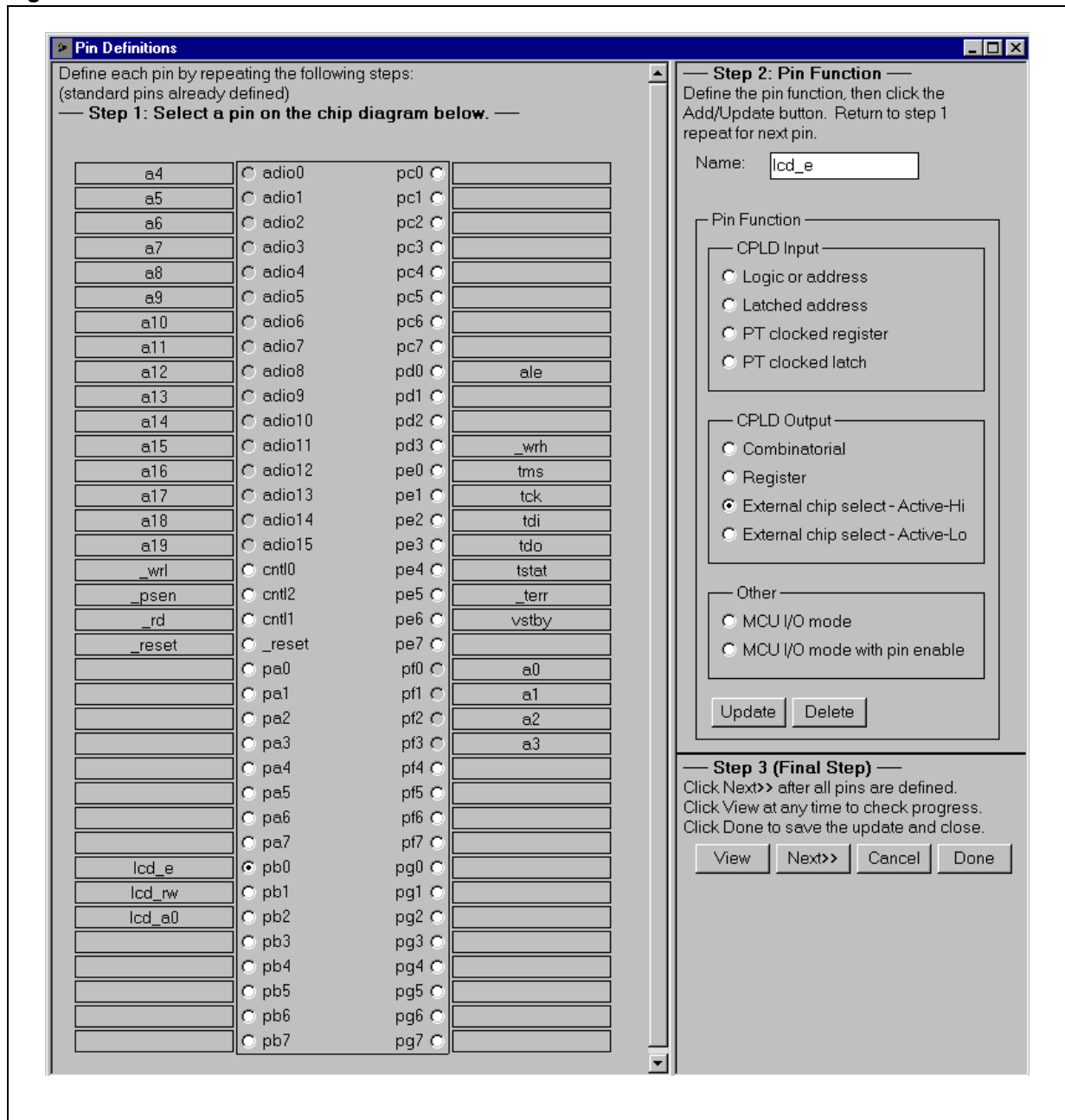
You can view a summary of your pin definitions by clicking the **View** button. When you are satisfied that you have defined all the pins correctly, click the **Next>>** button to be taken to the "Page Register Definition" screen as shown next.

Page Register Definition

Since 16-bit MCUs have an abundant number of address lines, memory paging is rarely needed for these MCUs. However, the PSD page register bits can be used for logic as well. You will learn how to do this in the second design example.

Figure 9. Page Register Definition
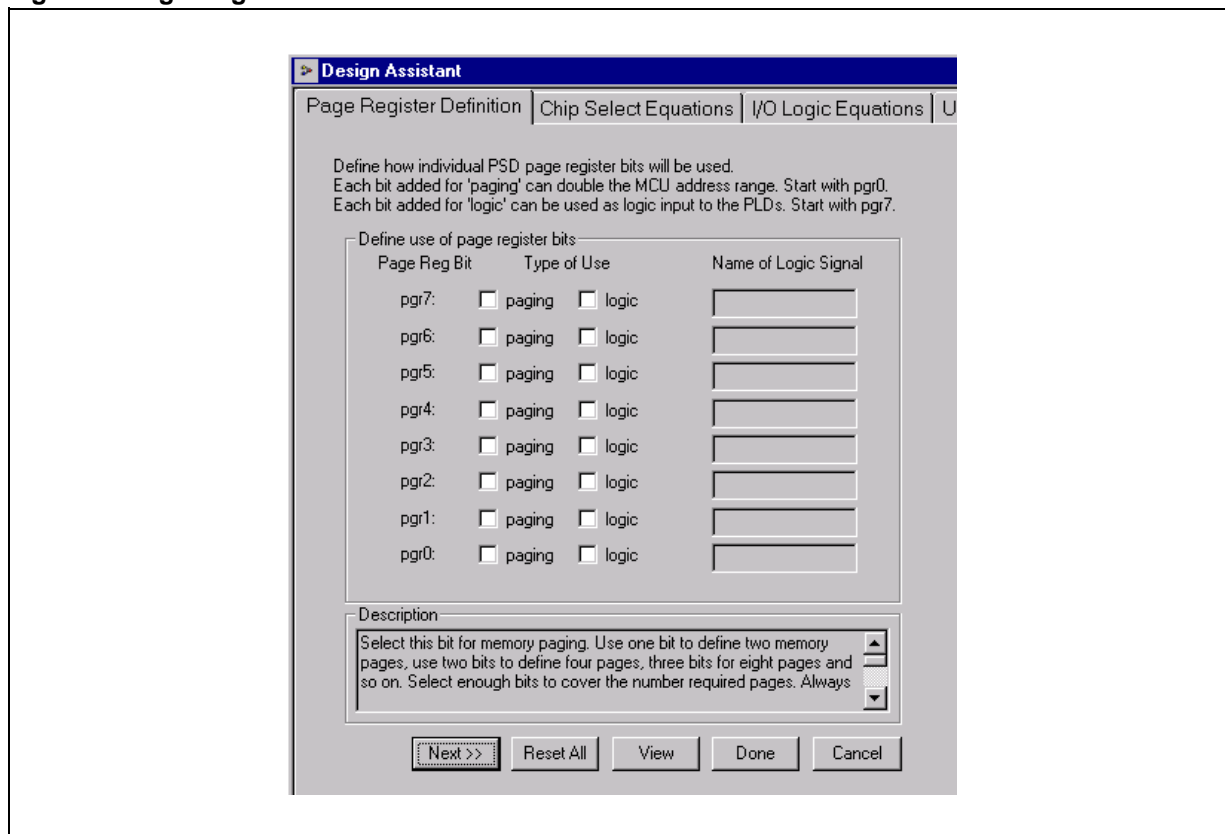
For this simple design, click **Next >>** or click on the "Chip Select Equations" tab.

**Chip Select Equations (system memory map)**

Now that the PSD pins are defined, you will need to define the system memory map. This is accomplished by defining all the chip-selects in the system (both internal to the PSD and external chip-selects).

The three memories inside the PSD are individually selected segment-by-segment when MCU addresses are presented to the Decode PLD (DPLD). Each internal PSD memory segment has its own individual chip-select name. For example, the main PSD Flash memory has eight individual chip-selects (one for each sector) named fs0 – fs7. See the PSD4235G2 data sheet for details. Each PSD memory segment must be defined in PSDsoft *Express* if it is to be accessed by the MCU.

We must define the internal PSD memory segment chip-selects: fs0 to fs7, csboot0 to csboot3, rs0, and csiop to match the memory map of Figure 5. The external chip-select for the LCD module, lcd_e, must also be defined, as shown in Figure 5.

Your screen should look like the following:

Figure 10. Chip Select Equations
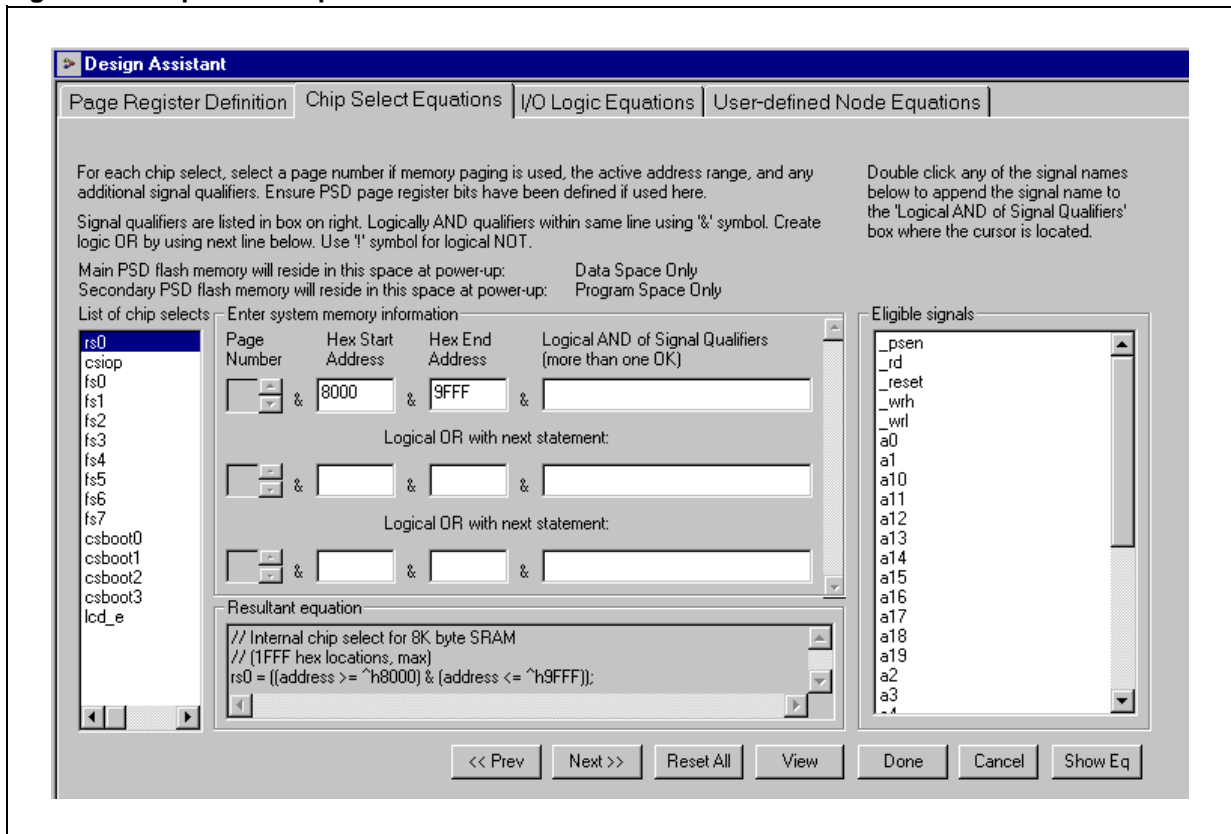
Start with the internal chip-select for the PSD SRAM, which is "rs0". Looking at the memory map of Figure 5, we see that 8 Kbytes (4 Kwords) of address space needs to be allocated to the PSD's internal SRAM. So, we enter the Start Address of 8000h and the End Address of 9FFFh as shown above. Notice that you do not have to qualify the rs0 chip-select with any MCU control signals (_rd, _wrh, _wrl, _psen, etc) because that is taken care of in silicon, just type in the addresses. This is true for all chip-selects of internal PSD memory — no MCU control signal qualifiers are necessary. Also notice that the 'Page Number' selection is grayed out since we defined no page register bits in the previous screen.

Next, define the chip-select for the internal PSD control registers by clicking on "csiop" on the left side of the screen. Enter its address range as shown:

Figure 11. CSIOP Address Range
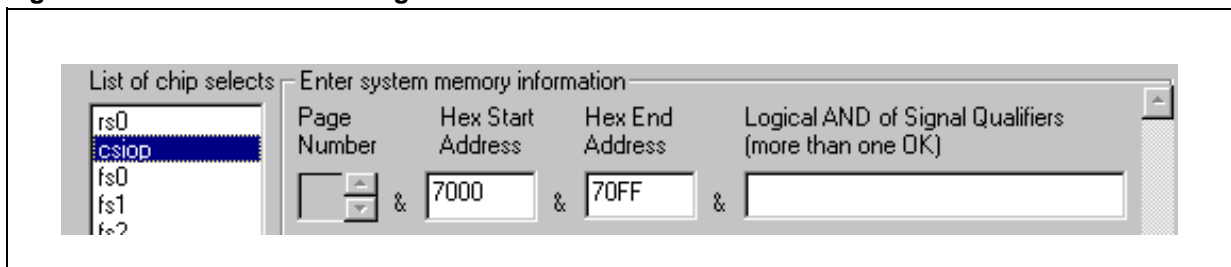
Continue to define internal PSD memory chip-selects for the main Flash memory segments fs0 to fs7, and then secondary Flash memory segments csboot0 to csboot3. Use Figure 5 as a guide for address ranges.

Again, no signal qualifiers are needed for internal PSD memory chip-selects. Here are a few examples of what the screen should like for these chip-selects:

Figure 12. FS0 Address Range

Figure 13. FS1 Address Range

Figure 14. FS7 Address Range

Figure 15. CSBOOT0 Address Range

Figure 16. CSBOOT7 Address Range
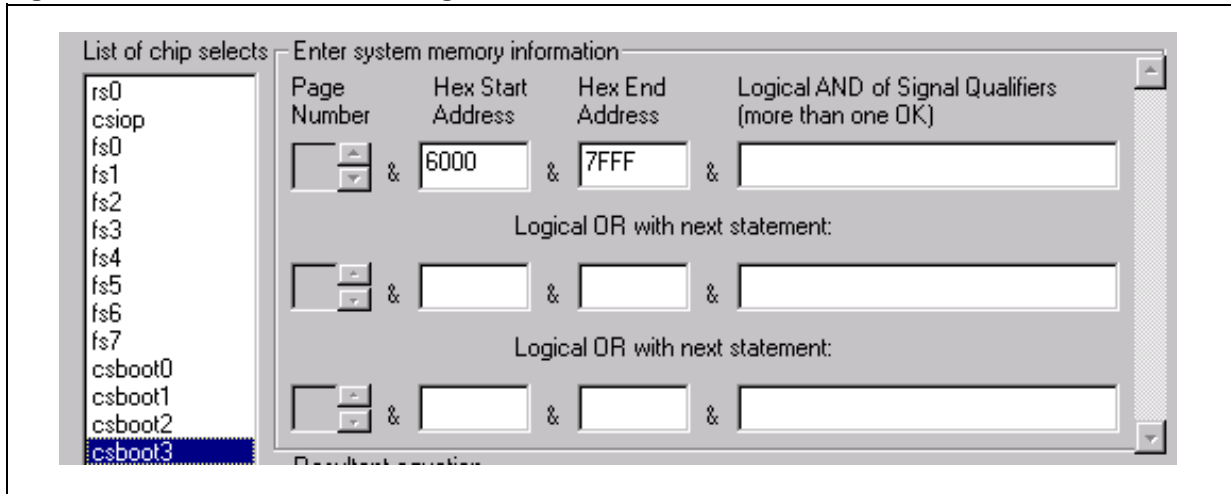
Finally, define the external chip-select for the LCD module, "lcd_e". This chip-select is different for two reasons. First, it is an external chip-select that does not activate any memory element inside the PSD because the signal "lcd_e" is output on a PSD I/O pin. And second, this chip-select requires qualifiers, meaning that this logic signal is true only for a given MCU address range AND only when one of two other another signals are active.

In this design, "lcd_e" is true only when the MCU presents an address in the range of 06000 to 06001h **AND** when either the P51XA control signal "_wrl" is true, **OR** when P51XA signal "_wrh" is true. To create this logic, enter information as shown in the screen below. Since both signals, "_wrl" and "_wrh", are active low as they leave the P51XA, the logical **NOT** operator (!) is used when they are specified as qualifiers.

Signal qualifiers may be added by parking the cursor where you want the signal name to go then just double-click on the signal name in the list of 'Eligible signals'.

Figure 17. Signal Qualifiers
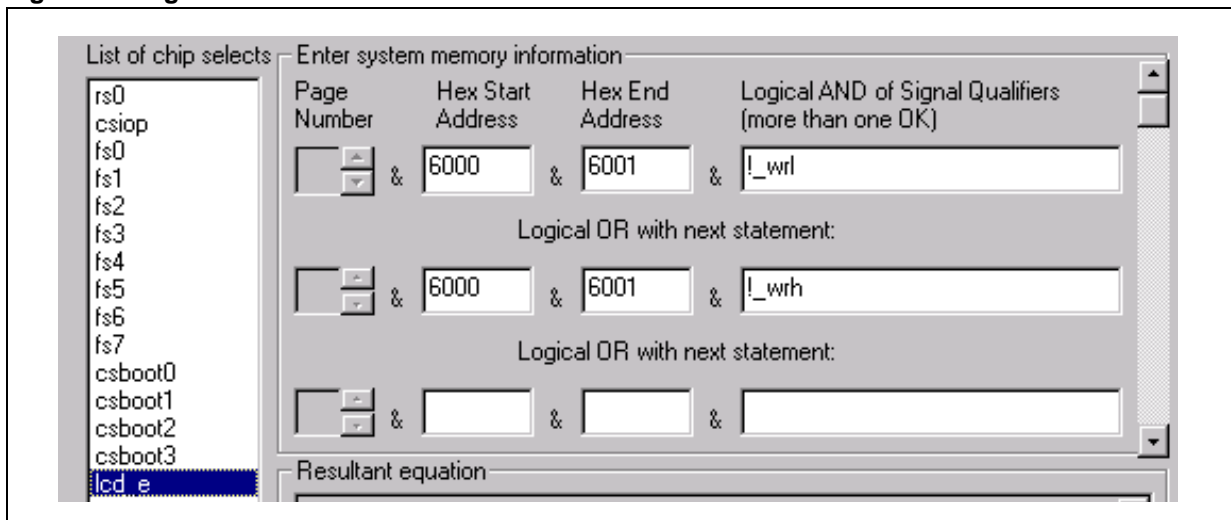
You can click the **View** button at any time to see a summary. Once you are satisfied with the results, click the **Next >>** button.

I/O Logic Equations

Now define the two combinatorial output signals "lcd_rw" and "lcd_a0". You should see the following screen:

Figure 18. I/O Logic Equations
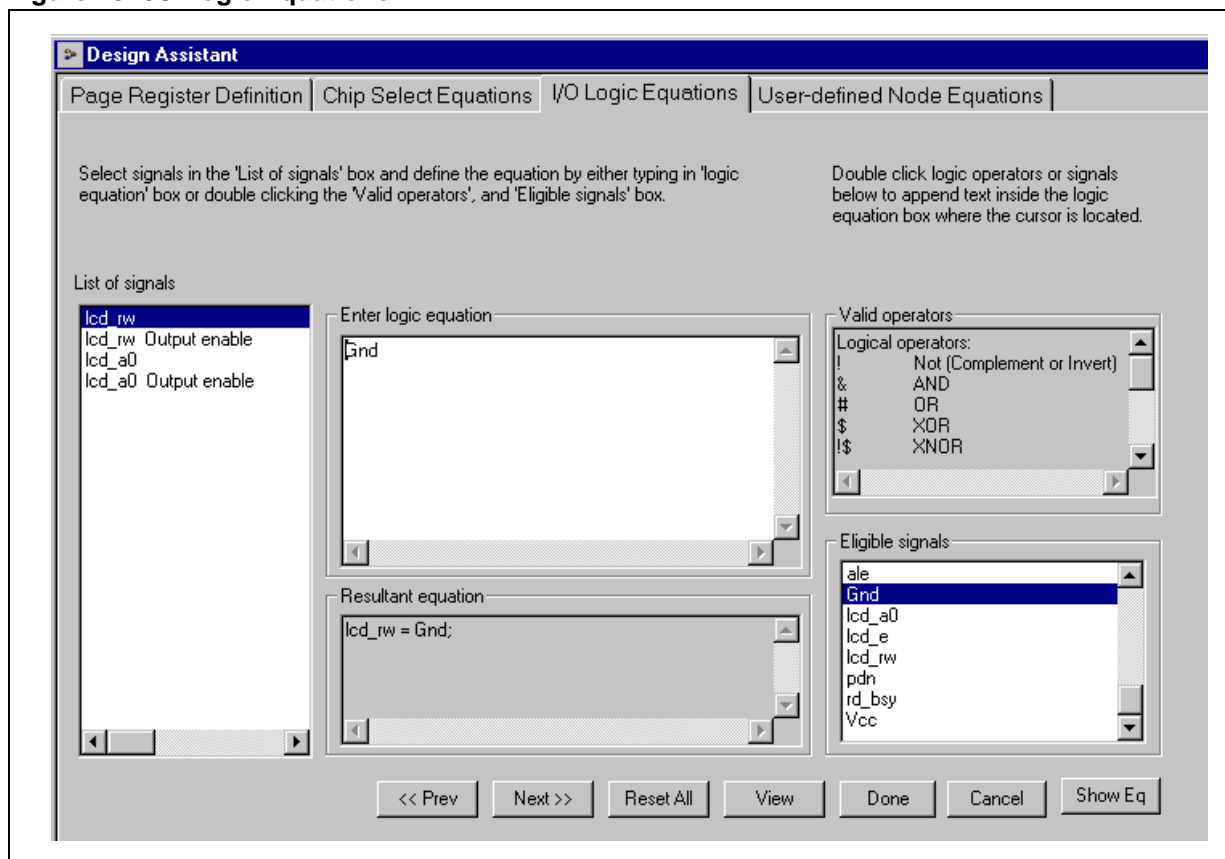
The signal "lcd_rw" should be a constant 0 volt output, so highlight the signal "lcd_rw" in the 'List of signals' box on the left. Then park your cursor in the 'Enter logic equation' box at the upper left corner. Now scroll down in the 'Eligible signals' box until you find the signal "Gnd". Double-click on "Gnd" and it will appear in the logic equation box as shown above. This is how you create equations for each of the I/O signals. You can also type the equations into the box.

Now set the output enable term for the signal "lcd_rw" to always active, or "Vcc" as shown:

Figure 19. LCD_RW Equation
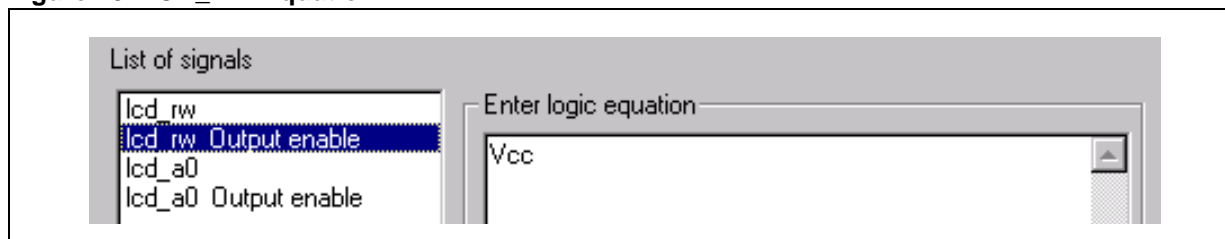
Next, define the signal "lcd_a0" as shown below:

Figure 20. LCD_A0 Equation
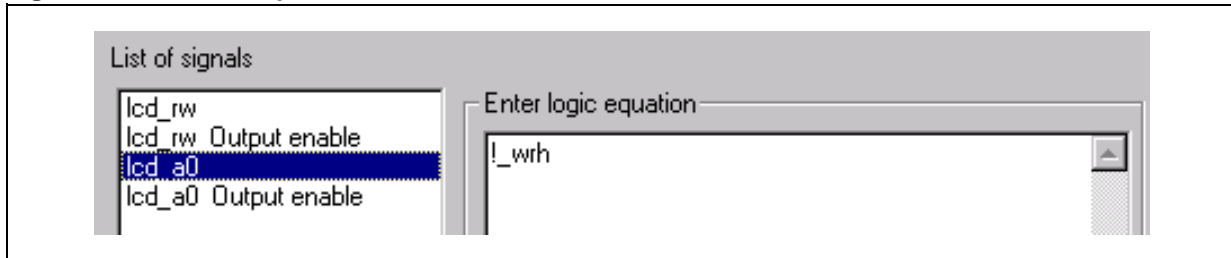
To do this, park your cursor in the 'Enter logic equation' box, then go to the 'Valid operators' box, and double-click on the "!" symbol. Now go to the 'Eligible signals' box and double-click on "_wrh". Lastly, set the output enable term for "lcd_a0" to "Vcc" just like "lcd_rw".

As an example of more complex logic, you can implement longer equations by adding signals and operators as shown in the following generic logic statement:

Figure 21. Generic Logic Statement
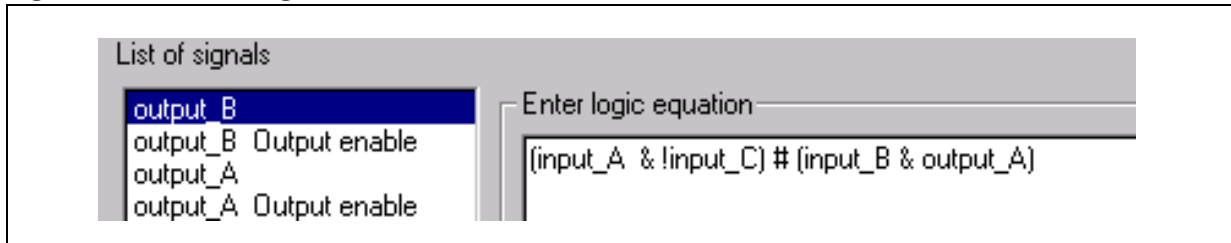
Notice that you can include other output signals (feedback) as part of the equation.

There are no 'User-Defined Nodes' in this simple design example, so click **Done**. This starts a preliminary resource and system check of the information you have entered. Analysis is performed to check for overlapping memory segments, problems with synthesizing the logic, and other problems. Any errors encountered will be indicated. An ABEL HDL file is generated.

Design Flow

Once you have clicked on **Done**, you are taken to the 'Design Flow' window. Use this window as your main navigational tool for PSDsoft *Express*™. Clicking on individual boxes within the flow diagram will invoke a process. A box shadowed in red identifies the next process that needs to be completed. The first three steps have been completed to this point. If you invoke a process that invalidates other processes downstream, the gray boxes indicate which processes must be invoked again and the red shadow indicates which process to invoke first.

The design flow should be in the following state:

Figure 22. Design Flow
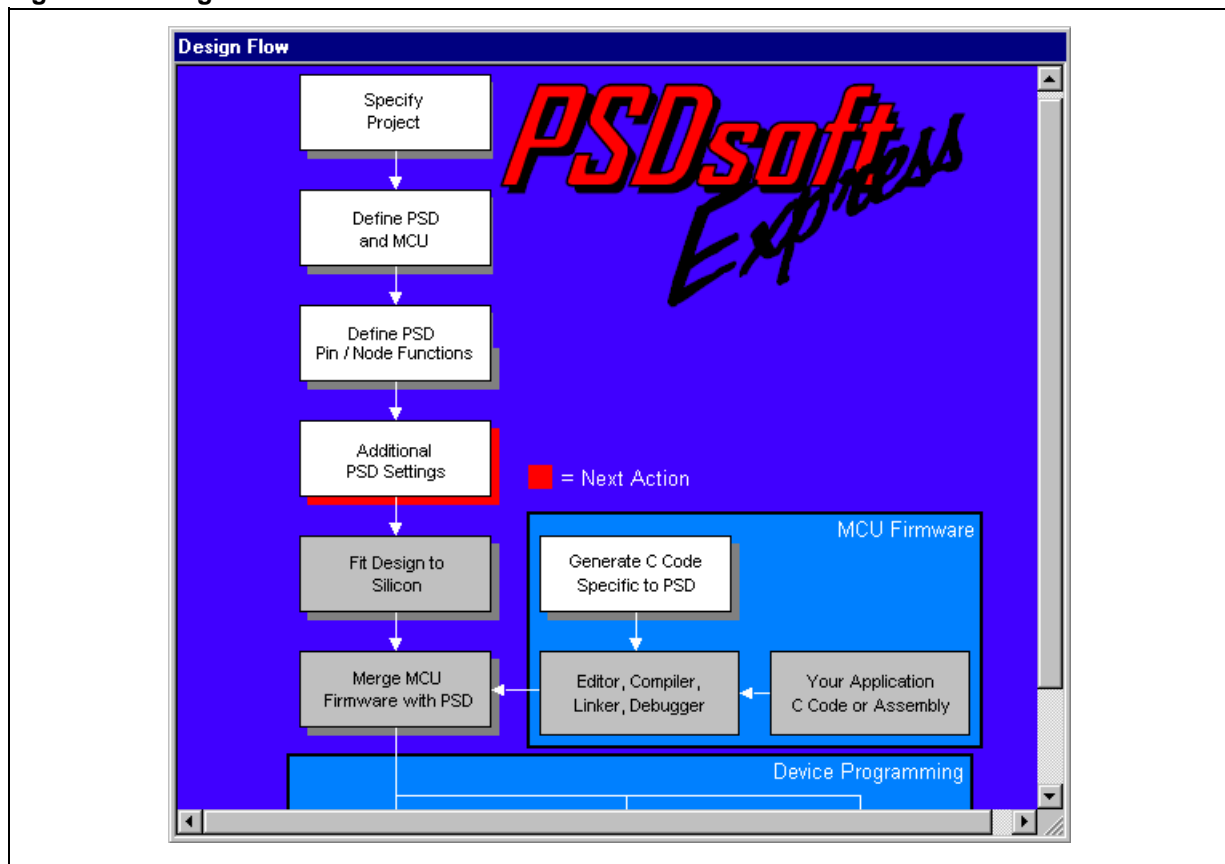
Additional PSD Settings

Click the 'Additional PSD Settings' box. This is where you may choose to set the security bit to prevent a device programmer from examining or copying the contents of the PSD. You can also click through the other sheets on this screen to set the JTAG IEEE 1149.1 USERCODE value and set sector protection on individual PSD non-volatile memory segments as desired.

Fit Design to Silicon

Click the 'Fit Design to Silicon' box. PSDsoft *Express* will input the generated ABEL file and all other configuration settings to synthesize the logic, creating reduced logic equations and a fusemap that fits the PSD4235G2 silicon elements. When this process is complete, a report will pop up that shows the resulting pin assignments and the resulting reduced equations. This is the "fitter report", which you can use to document your design.

PSD-Specific C Code Generation

You can take advantage of the provided low-level C code drivers for accessing memory elements within the PSD by clicking on the 'Generate C Code Specific to PSD' box in the design flow window. ANSI C code functions and headers are generated for you to paste into your C compiler environment. Simply tailor the code to meet your system needs and compile. C code generation can be performed anytime after a project is opened.

To generate ANSI C functions and headers, simply specify the folder(s) in which you want the header files and the C source file to be written, and name the C source file. Select the categories of functions that you would like to include, then click **Generate**. Three files will be written to your specified folder(s):

■ <your_specified_name>.c — ANSI-C source for all of the selected functions

■ psd4235g2.h–ANSI-C — header file to define PSD registers

■ map4235g2.h–ANSI-C — header file to define locations of system memory elements.

Notice that you do not have a choice to rename the two generated header files. This is because those header files are specified by name within the generated C function source file. If you edit the names of the generated header files, be sure to edit the generated C function source file to match the new header file names.

The three generated files may now be tailored and integrated into your compiler environment. The file psd4235g2.h contains a #define statement for each individual C function within the <your_specified_name>.c file. Edit psd4235g2.h and simply remove the comment delimiters (//) from the #define statement for each generated C function that you would like to be compiled with the rest of your C source code.

There are also coded examples available. Click on the 'Coded Examples' tab at the top of the C Code Generation screen. 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 toward various MCUs. You may copy these files to some folder to browse them for ideas, or cut and paste sections from the examples into your own MCU cross-compiler environment.

Merge MCU Firmware with PSD

Now that all PSD pins and internal configuration settings have been defined, compiled, and fitted, PSDsoft *Express*™ will create a single object file (.obj) that is a composite of your MCU firmware and the PSD configuration. FlashLINK™, PSDpro, and third party programmers can use this object file to program a PSD device. PSDsoft *Express* will create a file called "simpleXA.obj" for this first design example.

During this merging process, PSDsoft *Express* will input firmware files from your MCU compiler/linker in S-record or Intel hexadecimal format. It will map the content of these files into the physical memory segments of the PSD according to the choices you made in the "Chip Select Equations" screen. This mapping process translates the absolute system addresses inside specified firmware files into physical internal PSD addresses that are used by a programmer to program the PSD. This address translation process is transparent. All you need to do is type (or browse) the file names that were generated from your MCU linker into the appropriate boxes and PSDsoft *Express* does the rest. You can specify a single file name for more than one PSD chip-select, or a different file name for each PSD chip-select. It depends on how your MCU linker has created your firmware file(s). For each PSD chip-select in which you have specified a firmware file name, PSDsoft *Express* will extract firmware from that file only between the specified start and stop addresses, and ignore firmware outside of the start and stop addresses.

Click on "Merge MCU Firmware" in the main flow diagram and you will see the following:

Figure 23. Merge MCU Firmware
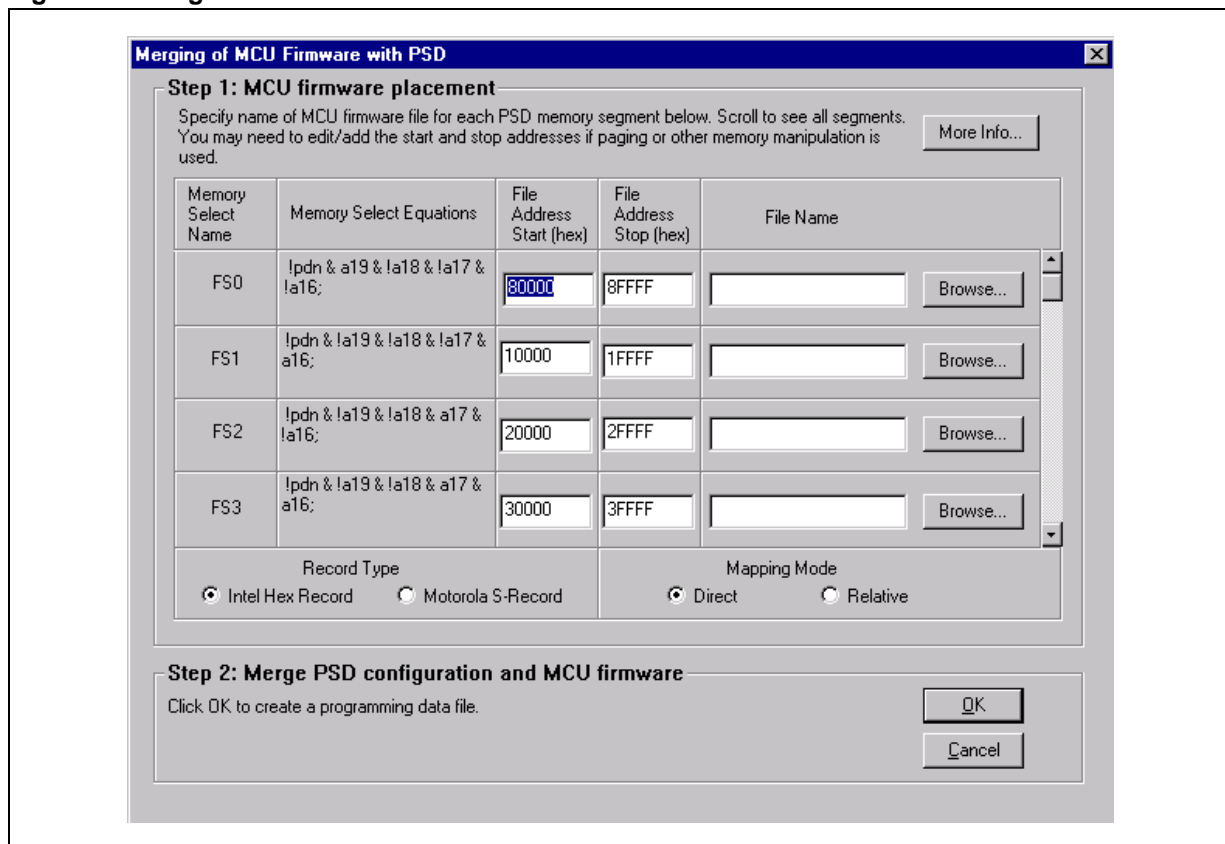
In the left column are individual PSD memory segment chip-selects (FS0, FS1, and so on). The next column shows the logic equations for selection of each internal PSD memory segment. These equations reflect the choices that you made while defining PSD internal chip-select equations in an earlier step. In the middle of the screen are hexadecimal start and stop addresses that PSDsoft *Express* has filled in for you based on your chip-select equations. On the right are fields to enter (browse) the MCU firmware files.

Select 'Intel Hex Record' for 'Record Type' as shown. Select 'Direct' for 'Mapping Mode'. This maps the MCU addresses residing inside the Hex file directly to the corresponding addresses within the range of the file start and stop addresses that are typed into the boxes. 'Direct' is the most typical setting. 'Relative' mode will place contents of the specified Hex file starting at the beginning of a physical PSD memory segment, in other words, no offset from the base of the physical memory segment. 'Relative' is used only for very unique applications.

Scroll all the way down to the bottom to get to the secondary Flash memory. Now, click **Browse…** for csboot0 and select the firmware file, PSDexpress\examples\boot_32K.hex. Repeat for csboot1, csboot2, and csboot3 specifying this same file name for each. Once you have filled in the file names, your screen should look like the one below:

Figure 24. Scroll to the Bottom to get to the Secondary Flash Memory
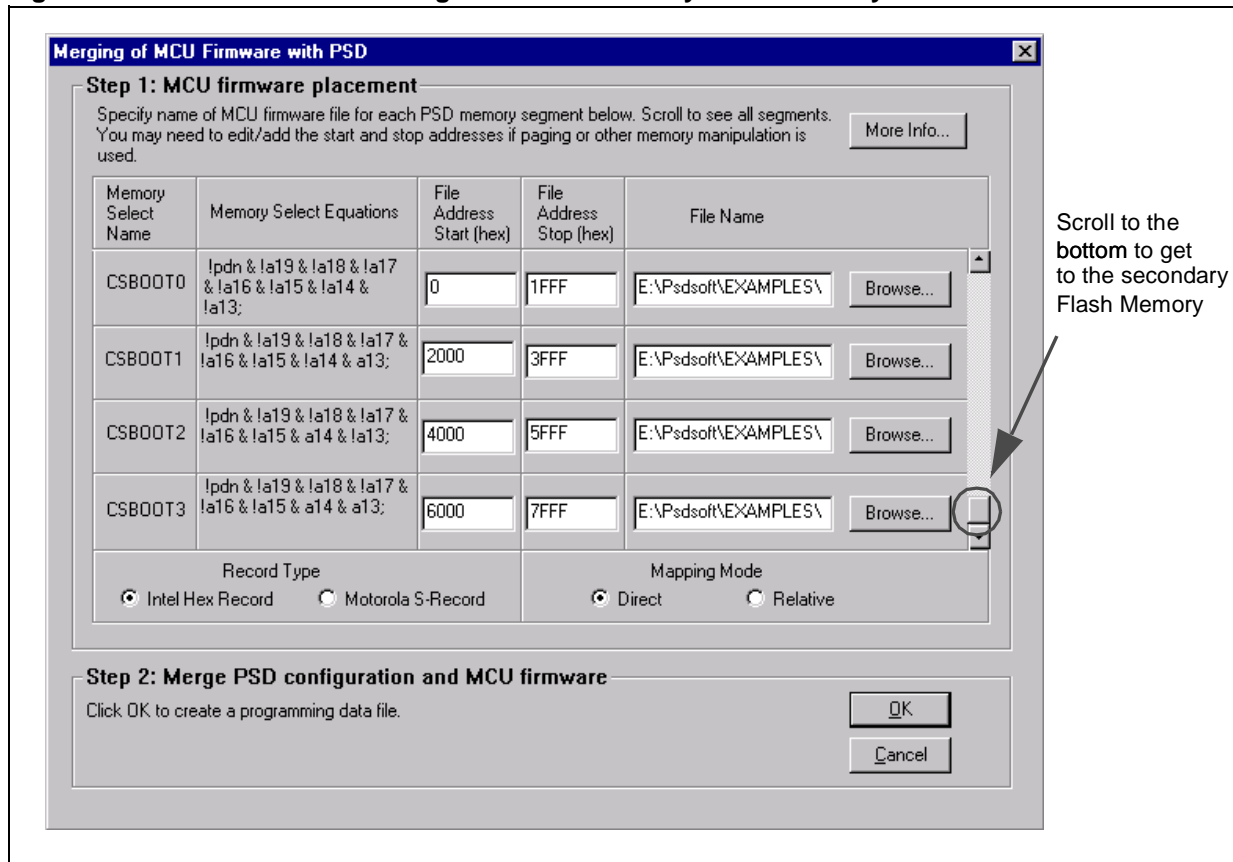
This specification places firmware in PSD secondary Flash memory segments csboot0 through csboot3. PSDsoft *Express* will extract any firmware that lies inside the file boot_32K.hex between MCU addresses 0000 and 7FFF and place it in appropriate PSD memory segment. Click **OK** to generate the composite object file, simpleXA.obj.

Note: the file boot_32K.hex does not contain P51XA firmware. It is used to illustrate the firmware merging process. Boot_32K.hex has a data pattern for each of the four segments of secondary PSD Flash memory. Csboot0 will receive AAh, csboot1 receives BBh, csboot2 receives CCh, and csboot3 receives DDh. The point is that although only one file name was specified for four different PSD memory segments, PSDsoft *Express* extracted the proper data for each segment based on the specified file start and stop addresses and the addresses contained inside the file boot_32K.hex. You may examine the contents of the file boot_32K.hex if you wish to better understand.

Programming the PSD

The file simpleXA.obj can be programmed into the PSD by one of three ways:

■ The ST FlashLINK™ JTAG cable, which connects to the PC parallel port.

■ The ST PSDpro device programmer, which also uses the PC parallel port.

■ Third-party programmers, from Stag, BP Micro, and others. See our website at *www.st.com* for list (PSD Products, Programming, then Programmers).

Programming with FlashLINK™

Connect the FlashLINK™ JTAG-ISP cable to your PC parallel port. Click the 'JTAG-ISP' box in the design flow window. You will be asked how many devices are in your JTAG chain. For this example, select 'Only One'. You would only select 'More than One' if you had more than one ISP device in your JTAG chain (even non-ST JTAG devices may be included in the chain). *You may choose to disable this question that appears each time you enter the JTAG screen, and then turn it back on later using the 'Preferences' menu choice from the 'Project' pull-down menu.* Click **OK** after your selection, you should see the following screen:

Figure 25. JTAG Chain Setup Window
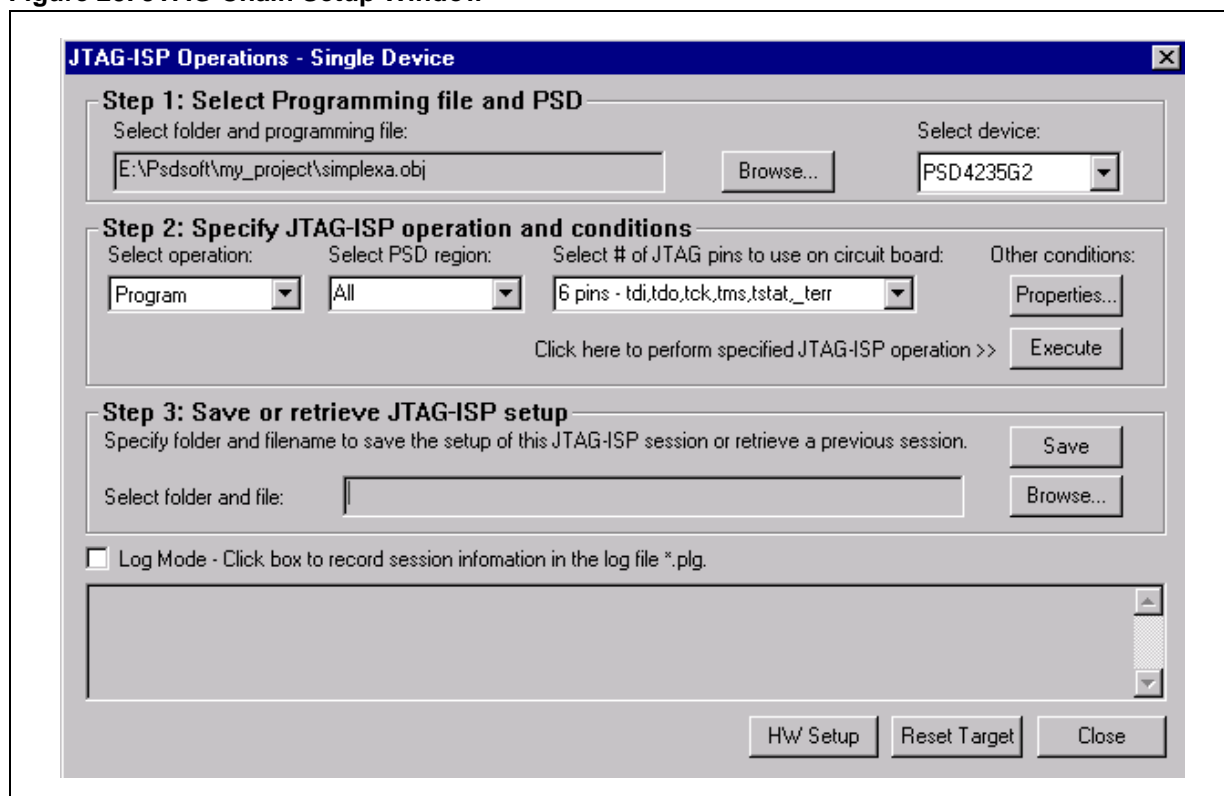
This window enables you to perform JTAG-ISP operations and also offers a loop back test for your FlashLINK™ cable. If this is your first use, test your FlashLINK™ cable and PC parallel port by clicking the **HW Setup** button, then click **LoopTest** button and follow the directions.

Now let's define our JTAG-ISP environment. PSDsoft *Express* should have filled in the folder and filename of the object file to program, the PSD device, and the JTAG-ISP operation, as shown in the screen above in 'Step 1'. For this design example, we have chosen to use all six JTAG-ISP pins (instead of four) so six pins is automatically filled in. Using all six pins reduces programming time by 10%-15%. Refer to Application Note 54 for details.

To begin programming, connect the JTAG cable to the target system, power-up the target system, and click **Execute** on the JTAG screen in 'Step 2'. The Log window at the bottom of the JTAG screen shows the progress. You can choose to save all log messages to a file by clicking the 'Log Mode' box.

There are optional choices available when the **Properties…** button is clicked. One choice includes setting the state of all non-JTAG PSD I/O pins during JTAG-ISP operations (make them inputs or outputs). The default state of all non-JTAG PSD I/O pins is "input", which is fine for this design example. The other

choice allows you to specify an IEEE 1149.1 USERCODE value to compare before any JTAG-ISP operation starts. This is typically used in a manufacturing environment. See the on-screen description for details.

After JTAG-ISP operations have completed, you can save the JTAG setup for this programming session to a file for later use. To do so, click on the **Save** button in 'Step 3'. To restore the setup of a previous session, click the **Browse…** button in 'Step 3'.

Programming with PSDpro

Connect the PSDpro device programmer to your PC parallel port per the installation instructions. Click on the 'Conventional Programmers' box in the design flow window. You will see this:

Figure 26. Parallel Programming Window
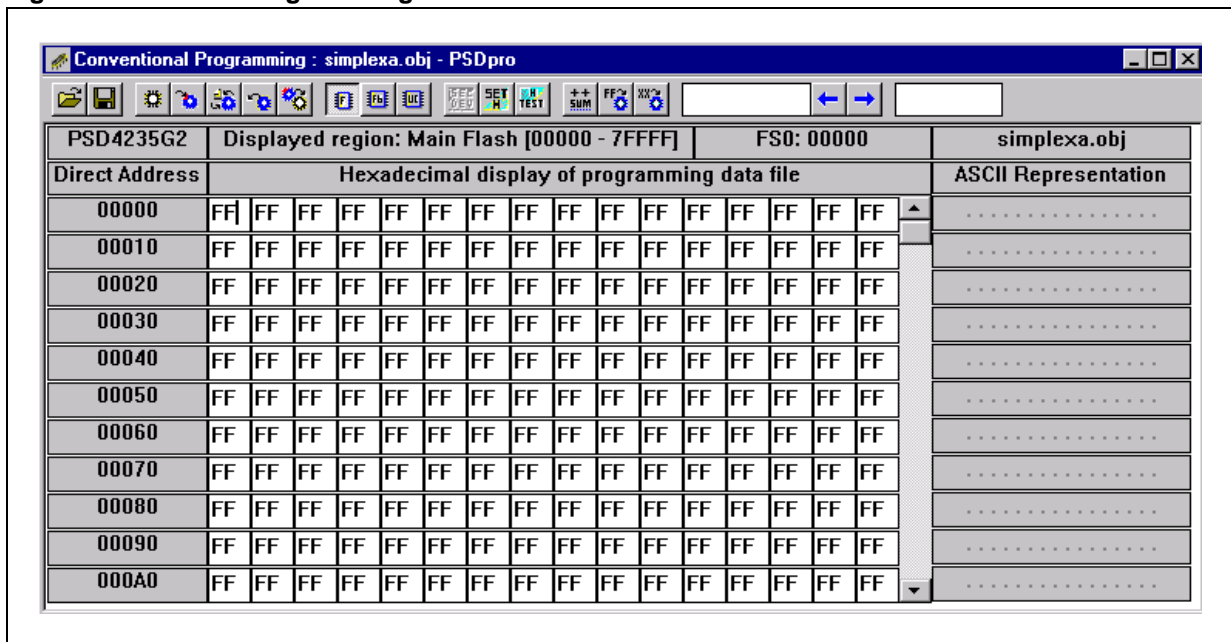
If this is the first use of the PSDpro, you'll need to designate the PSDpro as the device connected to your parallel port. To do this, click the **SET H** icon button at the top of the "Conventional Programming" screen and choose the PSDpro. Then click on the **H TEST** icon to perform a test of the PSDpro and the PC parallel port. After testing, place a PSD4235G2 into the socket of the PSDpro and click on the **Program** icon. (The simpleXA.obj file is automatically loaded when this process is invoked.) The messaging of PSDsoft will inform you when programming is complete.

This window is also helpful even if you do not have a PSDpro device programmer. Use this window to see where the 'Merge MCU Firmware' utility has placed MCU firmware within physical memory of the PSD. For this design example, click on the secondary PSD Flash memory icon "Fb" in the tool bar. You can see the AAh pattern in csboot0 from the file boot_32K.hex. Scroll down to the beginning of csboot1 (address 82000) and see the BBh pattern, and so on. This is useful when you want to examine your own firmware. To see how all of the MCU absolute addresses translated into direct physical PSD memory addresses, view the report that PSDsoft generates under "Reports" from the main toolbar, then select "Address Translation Report." Within the report, the Start and Stop addresses are the absolute MCU system addresses that you have specified. The addresses shown in square brackets in the reportare the direct physical addresses used by a device programmer to access the memory elements of the PSD in a linear fashion (a special device programming mode that the MCU cannot access).

SECOND DESIGN EXAMPLE – ISP, FULL IAP & CPLD LOGIC ELEMENTS

This second design example builds upon the first by adding true IAP capability. You will see how to execute from secondary PSD Flash memory in program space while programming the main PSD Flash memory in data space, then move main PSD Flash memory to program space for execution. We will also create some complex logic in the CPLD requiring use of the Extended Design Assistant.

Memory Map

Figure 27 and Figure 28 represent the system memory maps for this design.

Figure 27 represents the system memory map at power-up and after reset. This map is also valid during IAP. Notice that all of the main PSD Flash memory is initially in Data space so that the P51XA can write to it during IAP. Also notice that all of the secondary PSD Flash memory is initially in Program space so the P51XA can execute code from it during IAP. The choice for this initial placement of memory in Program or Data space was made within PSDsoft Express ('Define MCU and PSD' in flow diagram).

Figure 28 represents the system memory map after IAP is complete. All of main PSD Flash memory has moved to Program space. The PSD has a control register (named the VM register) that allows the P51XA to change the definition of Program space and Data space at run-time for IAP purposes. This VM register is accessed at an address offset from the base address, "csiop".

Sequence of events for IAP:

■ Figure 27 - at power on or after reset, the P51XA boots from secondary PSD Flash memory

■ Figure 27 – P51XA runs a checksum of the main PSD Flash memory in Data space

■ Figure 27 - If needed, P51XA programs and verifies main PSD Flash memory in Data space via the UART

■ Figure 27 – P51XA writes 06h to the VM register to place main PSD Flash memory into Program space

■ Figure 28 – main Flash memory has moved to program space as a result of writing 06h to VM register

■ Figure 28 – P51XA can now execute application code from either main or secondary PSD Flash memory

To accomplish this IAP function, no chip-select equations have to change from the first simple design example. Only the VM register must be accessed at run time as described above.

For MCUs/MPUs without Harvard architecture (Harvard: separate program and data address spaces) the VM register is not needed since there is only one address space for both code and data. IAP is much simpler for these MCUs/MPUs.

Figure 27. Memory Map at Boot-Up or Reset and During IAP

P51XA
Program Space

P51XA
Data Space

8FFFFh	fs0 64K bytes PSD Main Flash	8FFFFh — 80000h ⎫ 64K bytes
	fs7 64K bytes PSD Main Flash	7FFFFh — 70000h ⎫ 64K bytes
	fs6 64K bytes PSD Main Flash	6FFFFh — 60000h ⎫ 64K bytes
	fs5 64K bytes PSD Main Flash	5FFFFh — 50000h ⎫ 64K bytes
	fs4 64K bytes PSD Main Flash	4FFFFh — 40000h ⎫ 64K bytes
nothing mapped	fs3 64K bytes PSD Main Flash	3FFFFh — 30000h ⎫ 64K bytes
	fs2 64K bytes PSD Main Flash	2FFFFh — 20000h ⎫ 64K bytes
	fs1 64K bytes PSD Main Flash	1FFFFh — 10000h ⎫ 64K bytes
	nothing mapped	0FFFFh — 0A000h
08000h	rs0, 8k bytes PSD SRAM	08000h - 09FFFh
06000h - 07FFFh	csboot3, 8Kb PSD 2nd Flash	csiop, PSD cnti regs	07000h - 070FFh	64K bytes
04000h - 05FFFh	csboot2, 8Kb PSD 2nd Flash	lcd_e, ext LCD chip sel	06000h - 06001h
(Note 2) 02000h - 03FFFh	csboot1, 8Kb PSD 2nd Flash	nothing mapped	00400h - 05FFFh
00000h - 01FFFh	csboot0, 8Kb PSD 2nd Flash	P51XA-G3 Regs/SRAM	00000h - 003FFh

AI04081

Note: 1. PSD VM register initially 12h, Main PSD Flash memory in Data space.
2. IAP loader code gets programmed here by JTAG-ISP or conventional programmer tool.

Figure 28. Memory Map just after P51XA writes 06h to PSD VM register
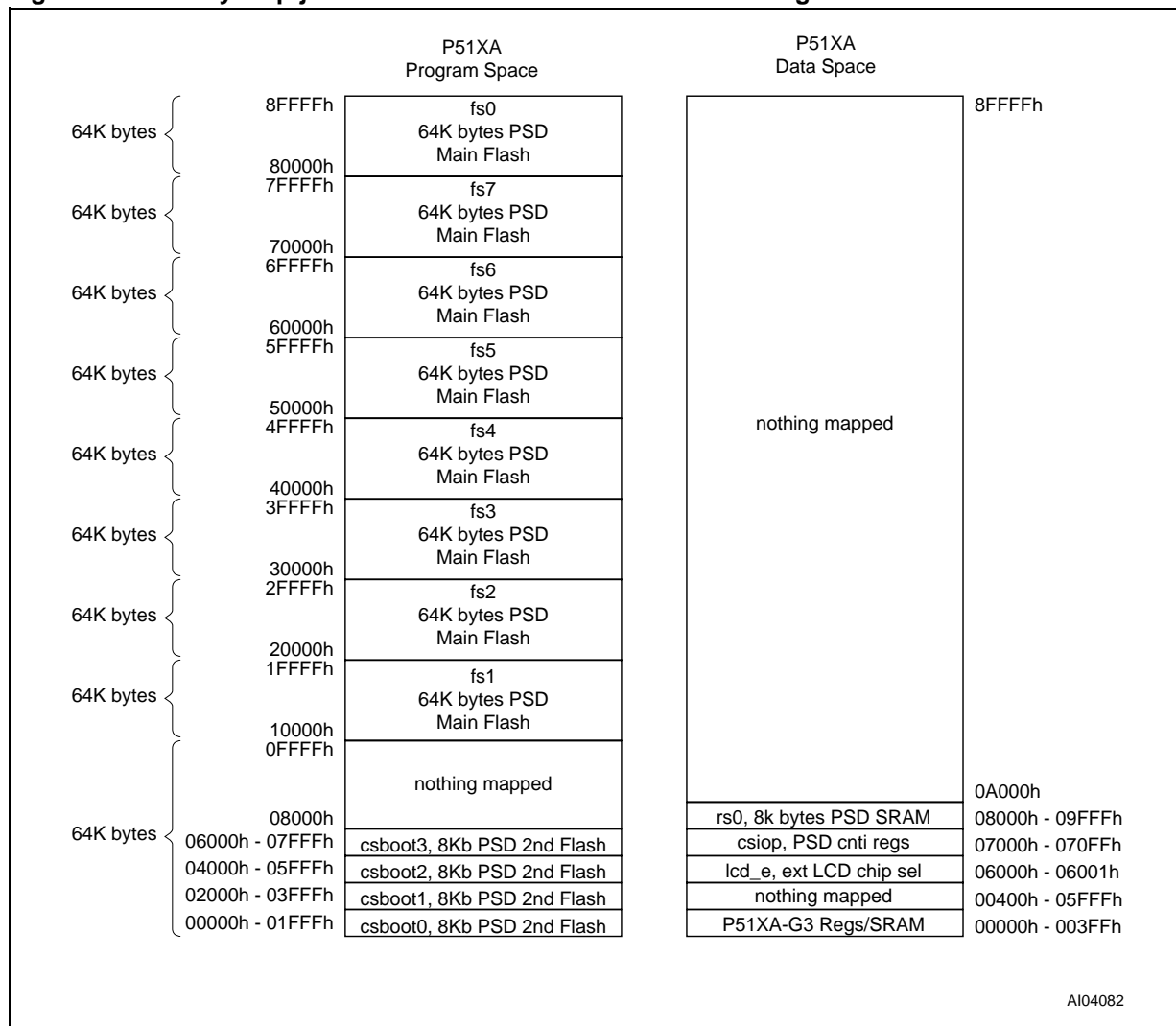
Note: 1. IAP complete, main PSD Flash memory moves to Program space

Your system design may require that you operate application code completely from main PSD Flash memory after IAP is complete. This means swapping the secondary PSD Flash memory (containing IAP loader code) out of Program space, and replacing it with main PSD Flash memory (containing application code). This is explained in the third design example.

PSDsoft Express Design Entry

We are finished with IAP issues, now let's get started on the advanced CPLD logic design. Invoke PSDsoft *Express*, open the project "simpleXA" from the first design example (if not already open). Pull down 'Project' from the menu at the top of the screen, and select 'Save As'. For this second design example, save the first project under the new name "logicXA".

For this second design, "logicXA" we want to use the Extended Design Assistant environment so go to the 'Project' menu pull down at the top of the screen and select 'Preference'. Then enable ABEL editing by clicking the box as shown, then **OK**.

Figure 29. Design Assistant Selection
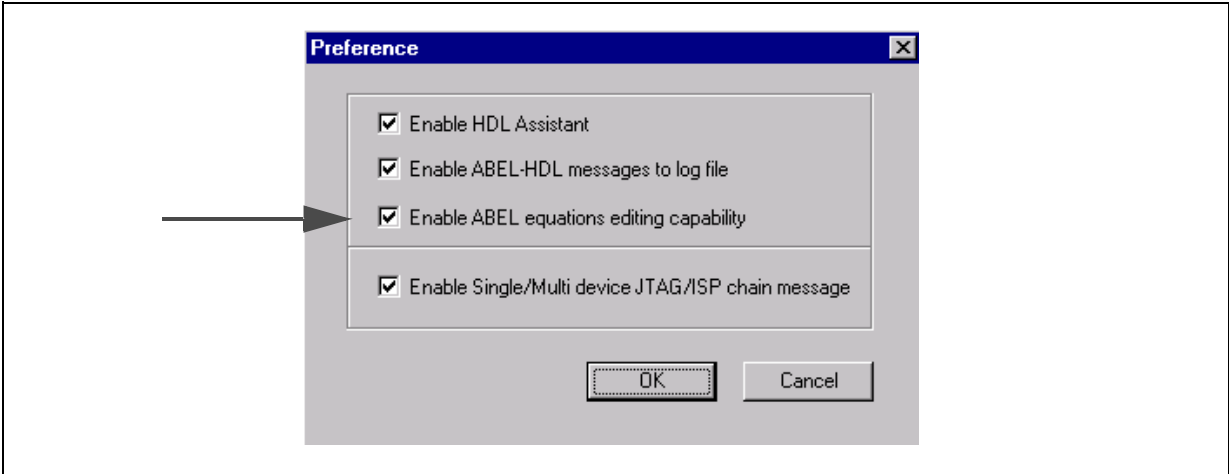
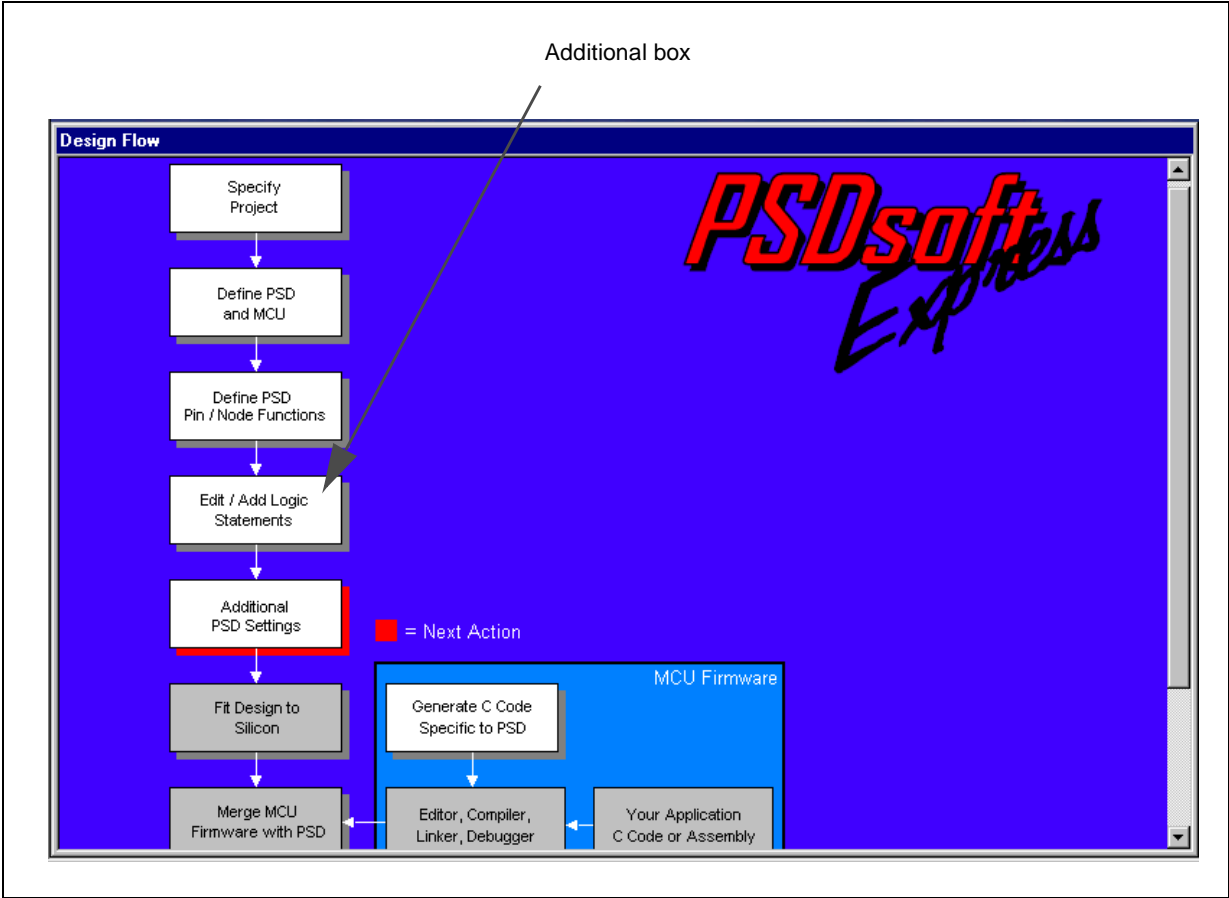You should see the full flow diagram as shown below. This flow also appears if 'Extended Design Assistant' is chosen for a new design. Note that the ability to edit the ABEL file is not available for PSD9XXF or PSD4135G devices, both of which have simple PLDs (no registers).

Figure 30. Design Flow

For this second design example, we'll implement the following logic elements to illustrate PSD functionality:

■ 4-state state machine with comparator feature.

■ Eight debounced inputs used for state machine input.

■ 4-bit reloadable down-counter with initial value set by the MCU.

■ Simple clock divider circuit.

■ 20 general purpose I/O pins controlled by MCU firmware.

■ PSD page register.

■ Miscellaneous combinatorial logic.

The general tactic is to use the Graphic User Interface (GUI) of the Designs Assistant as much as possible to create these logic functions before we have to manually edit the generated ABEL HDL file. You will see that the GUI creates all of the necessary pin and signal declaration statements as well as some of the simple logic equations. After this point, we will open the ABEL file and add more ABEL statements to implement the state machine and down-counter.
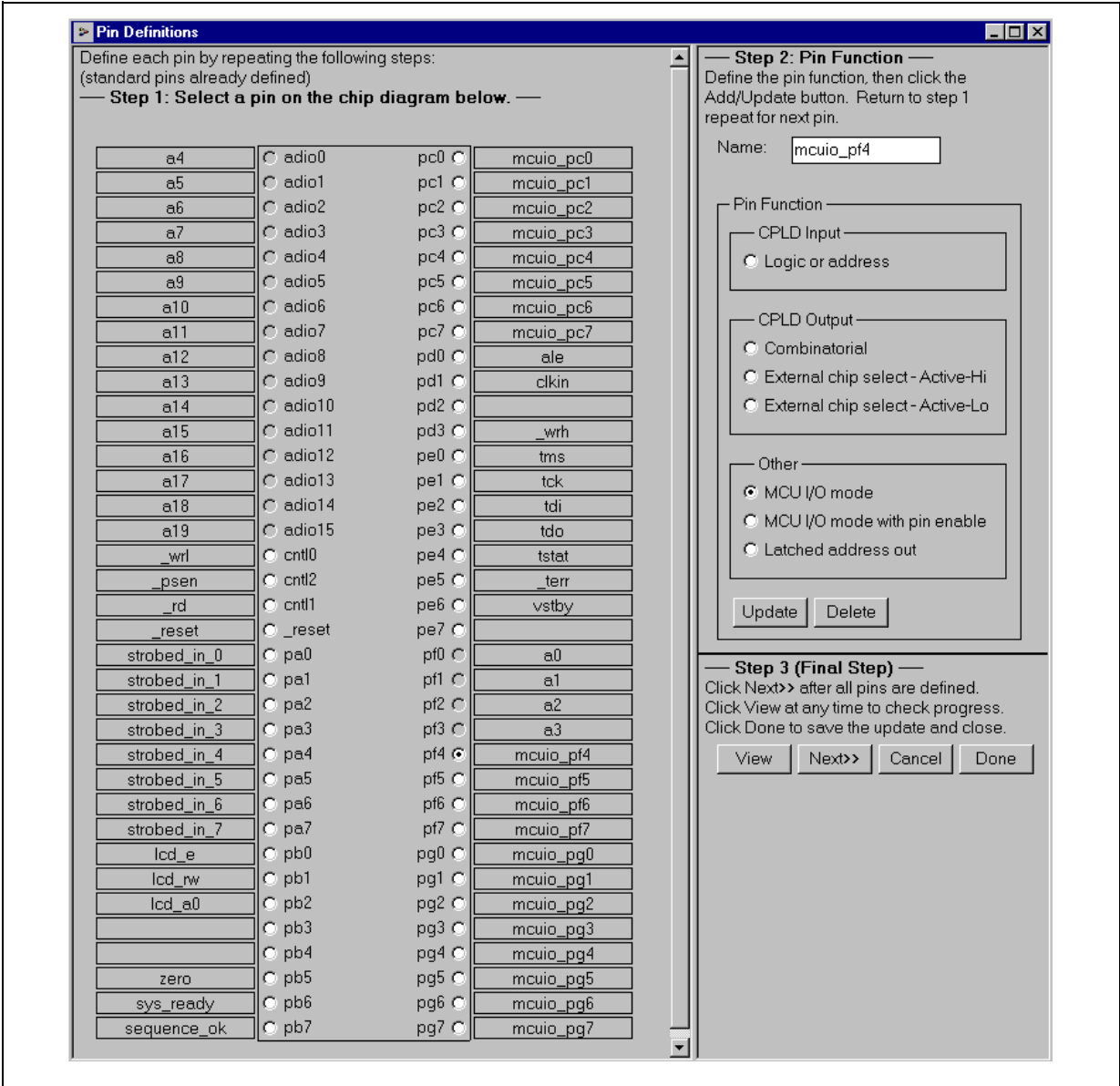
Pin Definitions

To achieve this, let's go back and define the remaining pin functions from the schematic of Figure 4. Click on the 'Define PSD Pin/Node Function' box and add the following signals:

■ Define eight inputs on Port A that are clocked (sampled) as they enter the PSD. Choose **Product Term (PT) clocked register** from the CPLD Input section, and name them "strobed_in_0" through "strobed_in_7". In silicon, these are IMCs.

■ Define a combinatorial CPLD output on Port B pin pb5. Choose **Combinatorial** from the CPLD Output section and name it "zero". Click **Add**.

■ Define a logic input to the CPLD on Port B pin pb6. Choose **Logic or address** from the CPLD Input section name it "sys_ready". Click **Add**.

■ Define a combinatorial CPLD output on Port B pin pb7. Choose **Combinatorial** from the CPLD Output section and name it "sequence_OK". Click **Add**.

■ Define eight MCU general purpose I/O signals on Port C. The MCU can set these pins to logic high or low as outputs, or read the pins as inputs all through firmware at runtime. To set this up, choose **MCUI/ O Mode** from the Other section and name them "mcuio_pc0" through "mcuio_pc7".

■ Define four MCU general purpose I/O signals on Port F. Choose **MCUI/O Mode** from the Other section and name them "mcuio_pf4" through "mcuio_pf7".

■ Define eight MCU general purpose I/O signals on Port G. Choose **MCUI/O Mode** from the Other section and name them "mcuio_pg0" through "mcuio_pg7".

■ Define a common PSD clock signal input on Port D pin pd1. Choose **Common clock input, CLKIN** in the Other section.

Your screen should look like this:

Figure 31. Pin Definition Screen

Click **Next>>**.

Page Register

This brings you to the PSD page register definition screen. Although we will not need to page memory since this MCU has plenty of address lines, we will use one of the page register bits for general logic. In this case, we define one page register bit as logic and name it "begin". This will be used in our state machine to allow it to start cycling. Using page register bits saves the use of OMCs. All page register bits are available as CPLD inputs. Note that the page register bits are cleared upon power-up and subsequent resets.

Define the "begin" bit as follows, then Click **Next >>**.

Figure 32. Page Register Definition
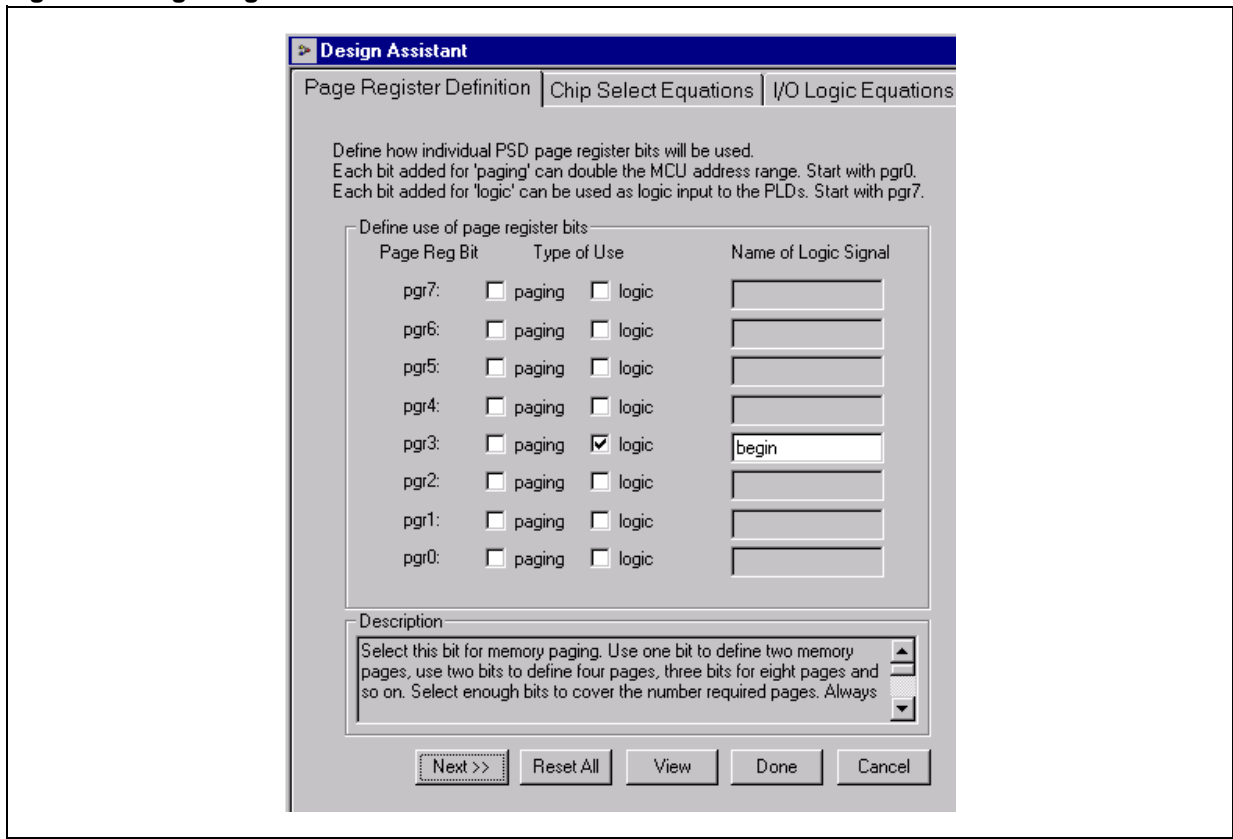
I/O Logic Equations

There are no changes needed to the memory map (chip-selects) from the first design as all IAP enhancements can be accomplished by using the VM register in this case. Click **Next >>** to skip the 'Chip Select Equations' screen. You should see the 'I/O Logic Equations' screen as follows:
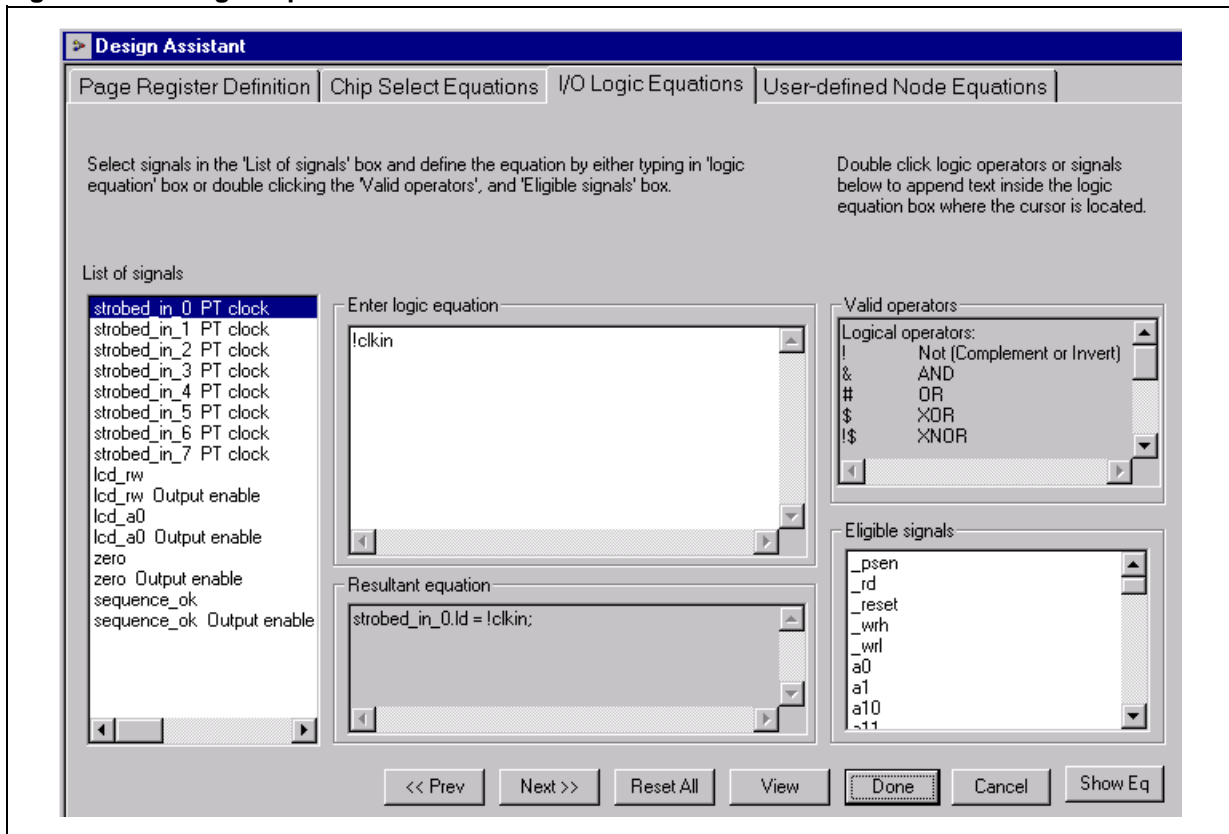
Figure 33. I/O Logic Equations

Notice the eight strobed inputs. These are Input Micro-Cells (IMCs), which offer a flip-flop on each of these input pins. For this example, we will define the clock to strobe these IMCs on the opposite edge of the state-machine clock. This will guarantee that a stable value is presented to the state-machine. Do this by parking the cursor in the 'Enter logic equation' box, double-click the "!" symbol in the 'Valid operators' box, then double-click "clkin". It should look like the figure above. Repeat this for all eight inputs.

The signals "lcd_rw" and "lcd_a0" should already be defined from the first design. The logic for the signals "zero" and " and "sequence_OK" will be defined when we type in logic while editing the generated ABEL HDL file, so leave them blank. However, define their output enable signals so they are always on by assigning "V_{CC}". That's all we need to do so click **Next >>**.

User-defined Node Equations

In this final screen of the Design Assistant, we define internal logic nodes, both combinatorial and registered. Your screen should look like this:

Figure 34. User-defined Node Equations
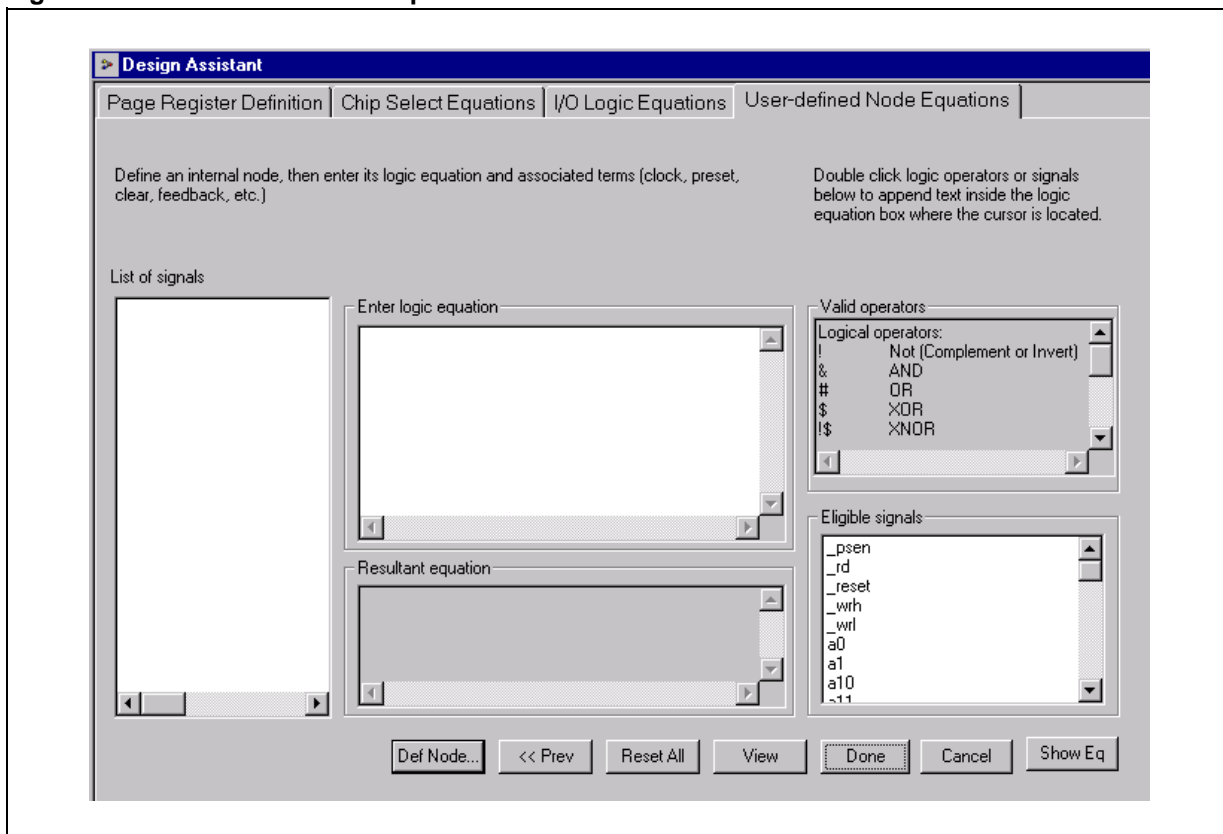
Let's establish all of the internal nodes used for this second design. They are:

■ Two register nodes for a 4-state state-machine — state_bit_0 and state_bit_1

■ One register node for a simple clock divider — half_clkin

■ Four register nodes for a down-counter — down_count0 to down_count3

■ Four register nodes to pre-load the down-counter — init_count0 to init_count3

■ One combinatorial node used as an intermediate node for down-counter — term_count

Each internal node, either combinatorial or registered, will consume one Output MicroCell (OMC).

First we'll define two registers to implement a 4-state state machine. Click on **Def Node…** Enter the name "state_bit_0" and designate it as a register as shown here:

Figure 35. Define Node Window
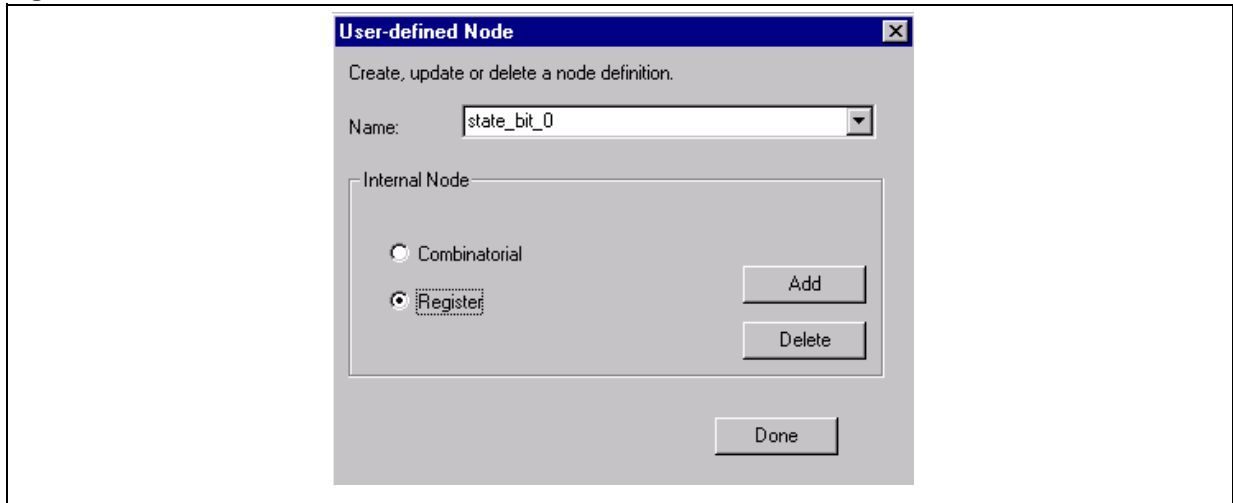
Now click **Add** . Do the same for state_bit_1. You should have the following entries in the 'List of Signals' box.

Figure 36. List of Signals
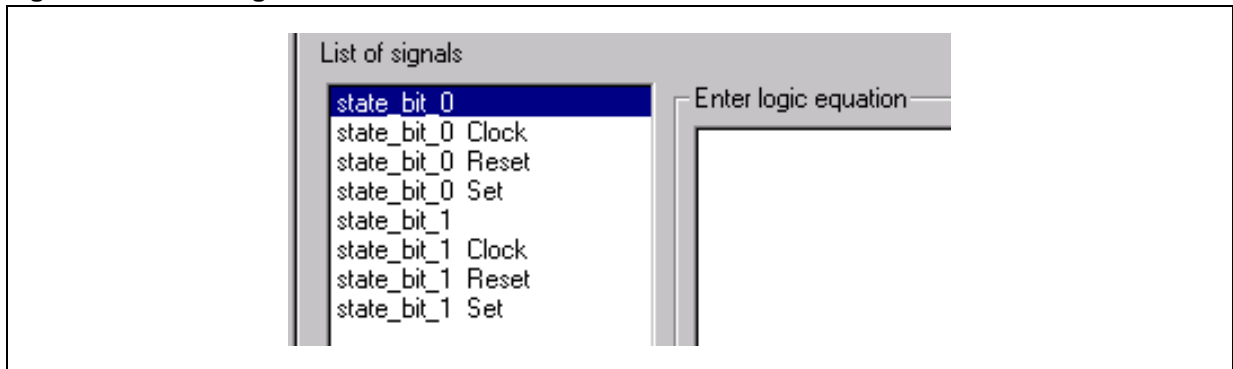
You can see that for each register node that is included, its input, clock, reset, and preset values are automatically added to the list. Equations can be specified for all of these elements. Figure 37 illustrates the relationship between a registered node and its signal names for this example.

Figure 37. Internal Register Node and Associated Signal Names
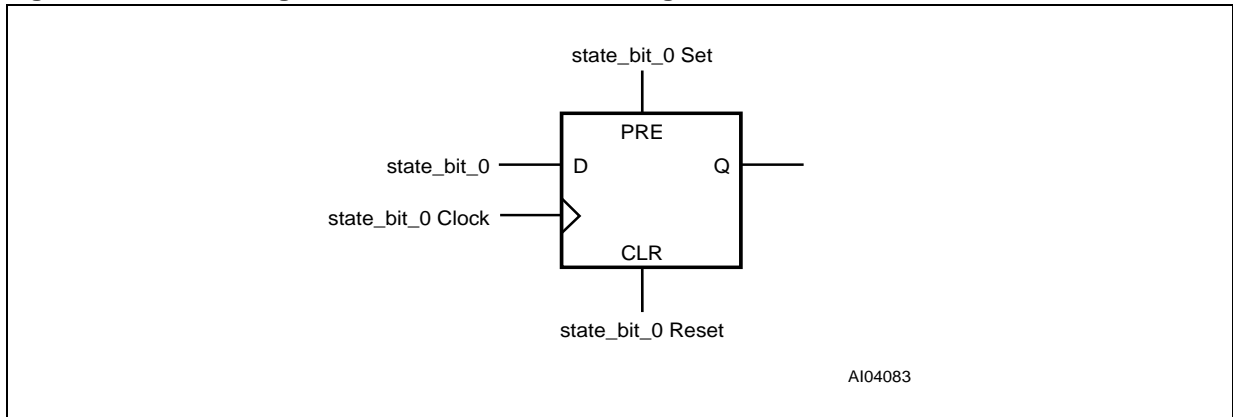
Continue to add nodes. Enter the following node names and node types, clicking **Add** after each:

```
down_count0 … register
down_count1 … register
down_count2 … register
down_count3 … register
init_count0 … register
init_count1 … register
init_count2 … register
inti_count3 … register
term_count … combinatorial
```

Next we will specify equations for as many of these nodes as we can using the Design Assistant GUI. Afterwards, we will manually edit the generated ABEL statements to finish the logic.

Let's start with the state machine nodes. We won't specify logic for the node inputs using the GUI because that will be done when the state machine statements are added to the ABEL file. So leave it blank as shown here (same for state_bit_1):

Figure 38. List of Signals
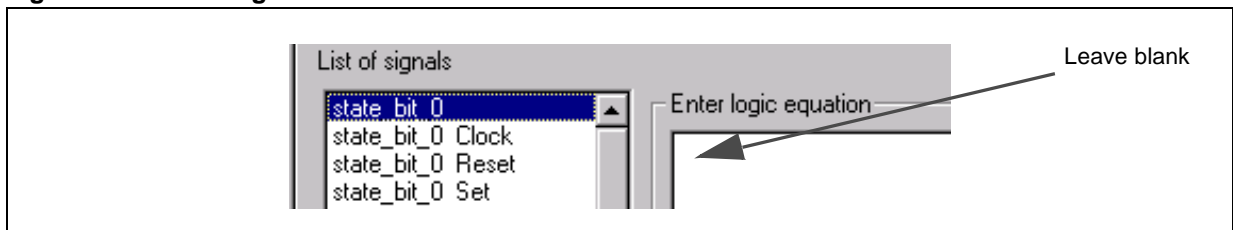
However, let's assign the clock for each state machine node as "clkin":

Figure 39. List of Signals
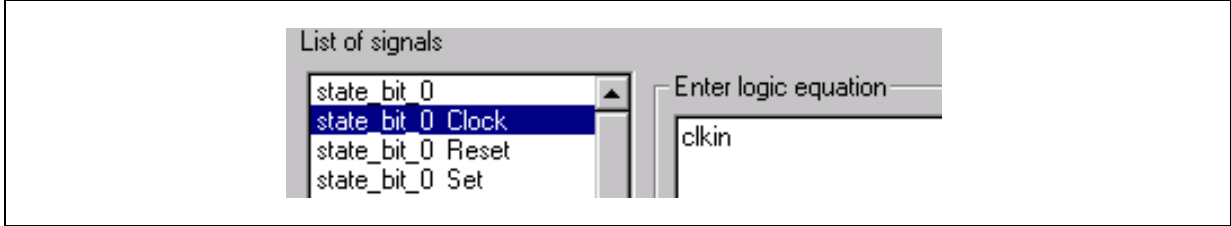
Assign the reset for each state machine node as "!_reset":

Figure 40. List of Signals
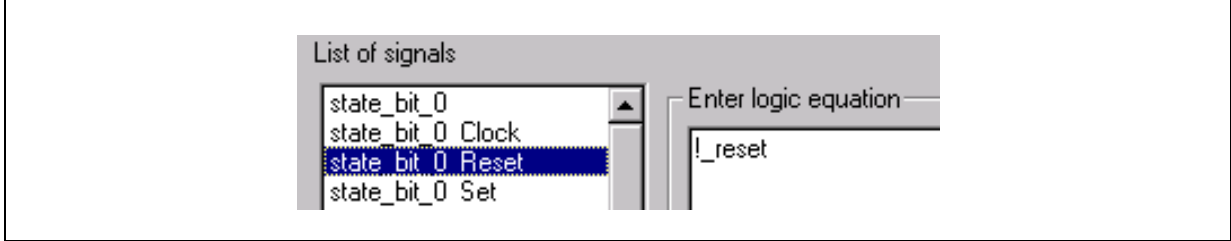
Assign the preset for each state machine node to be "Gnd"

Figure 41. List of Signals
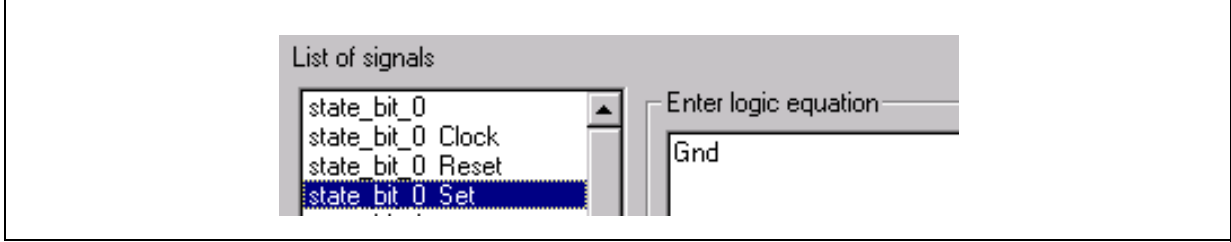
Now lets implement the logic for the simple clock divider circuit shown in Figure 42.

Figure 42. Simple Clock Divider
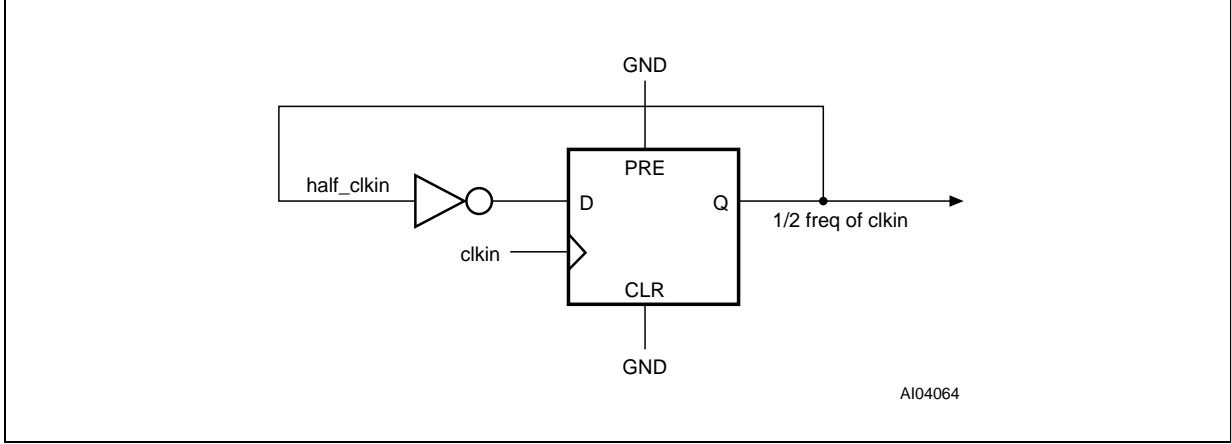
To do this, make these logic assignments for the registered node, "half_clkin":

■ half_clkin!half_clkin

■ half_clkin Clockclkin

■ half_clkin ResetGnd

■ half_clkin SetGnd

Moving to the down-counter, let's implement this simple 4-bit auto-reload down-counter, shown in Figure 43. It will be clocked by "half_clkin" and its output "zero" will indicate when the count has reached zero, at which time the counter will automatically reload the initial value and count down again. The MCU will load an initial 4-bit value just once (during a startup initialization routine) and the counter will automatically reload this value as needed. The MCU loads the count by writing to four OMCs that we have labeled "init_count0" through "init_count3". The MCU just has to write to the appropriate address offset inside PSD control register space, "csiop" to load the OMCs. See the data sheet for details on control register offsets and functions.

Figure 43. 4-bit auto-reloading down-counter
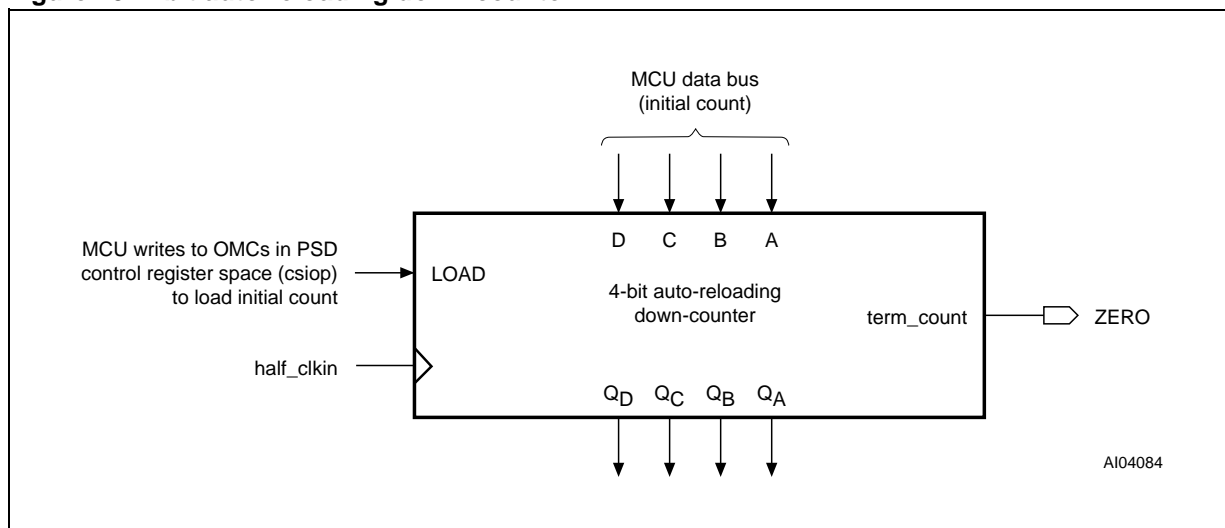
Let's make assignments for the clock, reset, and preset using the Design Assistant GUI, but leave each node input equation blank (assignment will be made later when editing ABEL file). The down-counter will be clocked with the divided clock of Figure 42. So make these assignments for the register nodes: down_count0, down_count1, down_count2, and down_count3.

■ down_countx<blank>

■ down_countx Clockhalf_clkin

■ down_countx Reset !_reset

■ down_countx Set Gnd

Next, make assignments for the four register nodes which hold the initial count of the down-counter. Again, no equation will be assigned to the node inputs (this happens when editing ABEL file). Make the following assignments for the register nodes init_count0, init_count1, init_count2, and init_count3.

■ init_countx <blank>

■ init_countx ClockGnd

■ init_countx Reset !_reset

■ inti_countx Set Gnd

The clock and preset inputs are grounded because we do not want any logic overriding what the MCU has loaded into these registers.

And finally, the combinatorial node, "term_count" needs no equation assigned from Design Assistant GUI because it will be defined by adding statements to the ABEL file later. Leave it blank.

Click **Done** and you will exit the Design Assistant. At this time, an ABEL HDL file is generated and preliminary checks are performed.

Editing the Generated ABEL HDL File

Click the box 'Edit/Add Logic Statements'. You will see two windows pop up.

The HDL Assistant window is there for your easy reference. You may browse through it to find ABEL language examples for various declarations and logic functions. Simply cut and paste the desired statements into the other window that has popped up, the PSDabel Design Entry window. Note that you may turn off the HDL Assistant feature by pulling down 'Preference' from the top menu and un-checking the appropriate box.

The PSDabel Design Entry window is a text editor for the generated ABEL file. Be careful to type only within the areas designated as "preserved" areas. If you type outside of these preserved areas, you will loose those statements next time PSDsoft generates the ABEL file again after a design iteration in the Design Assistant GUI.

There are two "preserved" areas, one for declarations and another for logic equations denoted by the following:

```
// Begin user preserved declarations (not affected by iterations of DA usage)


Type your declaration statements here …


// End user preserved declarations (not affected by iterations of DA usage)


// Begin user preserved equations (not affected by iterations of DA usage)


Type your logic equation statements here …


// End user preserved equations (not affected by iterations of DA usage)
```

For this example design, type in the following declarations:

```
// Begin user preserved declarations (not affected by iterations of DA usage)
===================
WSIPSD PROPERTY 'DataBus_OMC D[7:4]:down_count[3:0]';
        // This PROPERTY statement forces the alignment of
        // down_count bits [3..0] to the MCU data bus bit positions [7..4].
        // If this WSIPSD PROPERTY statement was not present, then PSDsoft
        // would pick random MCU bit positions. The WSIPSD PROPERTY is needed
        // only if the MCU will read or write to MicroCells and only if a
        // particular MCU data bus position is required by the designer.
```

```
WSIPSD PROPERTY 'DataBus_OMC D[3:0]:init_count[3:0]';
        // This statement forces the alignment of
        // init_count bits [3..0] to the MCU data bus bit positions [3..0].


DCOUNT = [down_count3..down_count0];  // 4-bit down counter
INIT = [init_count3..init_count0];// 4-bit initial count from MCU

STINPUTS = [strobed_in_7..strobed_in_0];
        // 8 inputs that are stobed (sampled) on the way into
        // the PSD (debounced). These inputs are clocked on the
        // opposite edge of the state machine clock (!clkin) so
        // they are stable for each state machine transition.


STATE_MACHINE = [state_bit_1..state_bit_0];
        // 2 bits for 4-state state machine. Clocked by
        // common PSD clock input (clkin).


// End user preserved declarations (not affected by iterations of DA usage)
===================


Now type in the following equations for the down-counter and the state-machine:
```

```
// Begin user preserved equations (not affected by iterations of DA usage)
===================


//**** 4-bit down counter. accepts initial value from mcu, auto reloads ****

term_count = (DCOUNT.fb == 0);  // true when count reaches zero
 when (term_count) then DCOUNT := INIT;
                        // automatically reload counter with initial
                        // value after a count of zero is reached


zero = term_count;      // Assign terminal count to PSD output pin


//**** simple state machine, looks for predefined sequence
//                  of values appearing on the 8 strobed inputs *****


state_diagram STATE_MACHINE;

state 0:
    sequence_ok = 0;           // indicate sequence not found yet
    if ((begin == 1) & (sys_ready == 1)) then 1 else 0;
                               // stay in this state if 'begin'
```

```
                                           // signal is zero or system is
                                           // not ready
state 1:
   if (STINPUTS == ^hAB) then 2 else 1;
                            // stay in this state until pattern ABh is found

state 2:
   if (STINPUTS == ^hCD) then 3 else 2;
                            // stay in this state until pattern CDh is found

state 3:
   sequence_ok = 1;
                       // indicate correct sequence was found, then start over
   goto 0;

// End user preserved equations (not affected by iterations of DA usage)
===================
```

Here is the representative state diagram of the state-machine:

Figure 44. Example State Machine
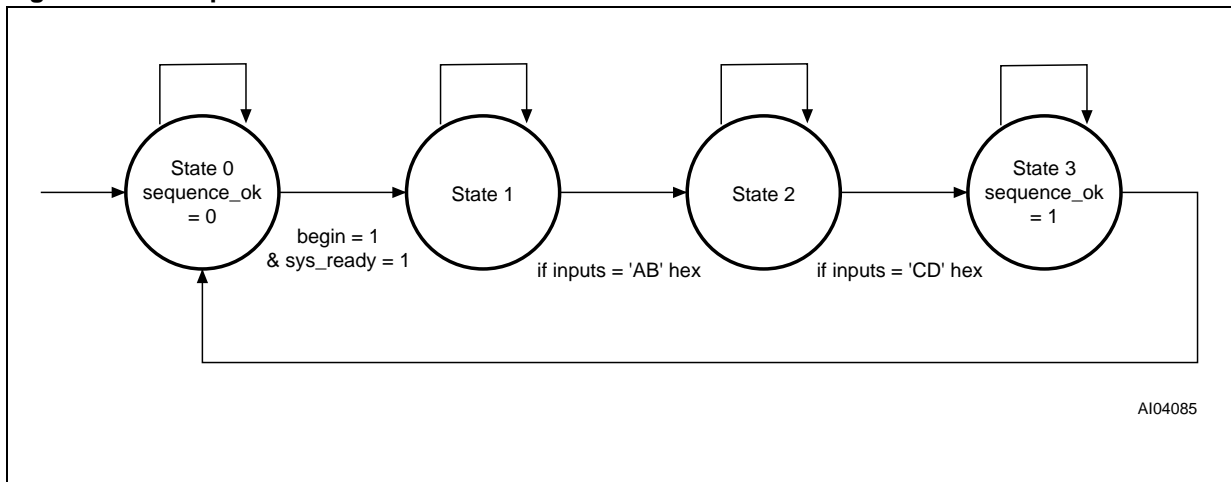
Finishing the design

Click the 'Fit Design to Silicon' box. After a successful fit, click the 'Merge MCU Firmware' box, and follow the same procedure used in the first design. No firmware filename needs to be designated for the main PSD Flash memory segments (fs0 – fs7) since they will be programmed by the P51XA during IAP. Click **OK** in the merging screen to create a composite object file, logicXA.obj, for programming. You are now ready to program the PSD as described in the section entitled "Programming the PSD" on page 21.

THIRD DESIGN EXAMPLE – ISP AND ADVANCED IAP

The third design example adds enhanced IAP features. The physical connections between the MCU and PSD4235G2 do not change, but chip-selects (memory map) and PSD page register definitions do change. We will not change any of the CPLD logic in this design.

This enhanced design derives the most utility out of the PSD architecture by providing a means to replace the secondary PSD Flash memory with a segment of main PSDflash memory (swapping) after IAP is complete. These benefits result:

■ IAP bootloader code in secondary PSD Flash memory can be updated in the field while executing from main PSD Flash memory.

■ The entire application can be executed from main Flash memory after IAP is complete.

■ The system software designer can make use of two sets of MCU interrupt vectors/routines and low-level code: one set during IAP (contained in secondary Flash memory) and a different set after IAP (contained in main Flash memory).

■ The secondary PSD Flash memory can be split in half. One half used for boot loader code during IAP and the other half used as general data storage after IAP.
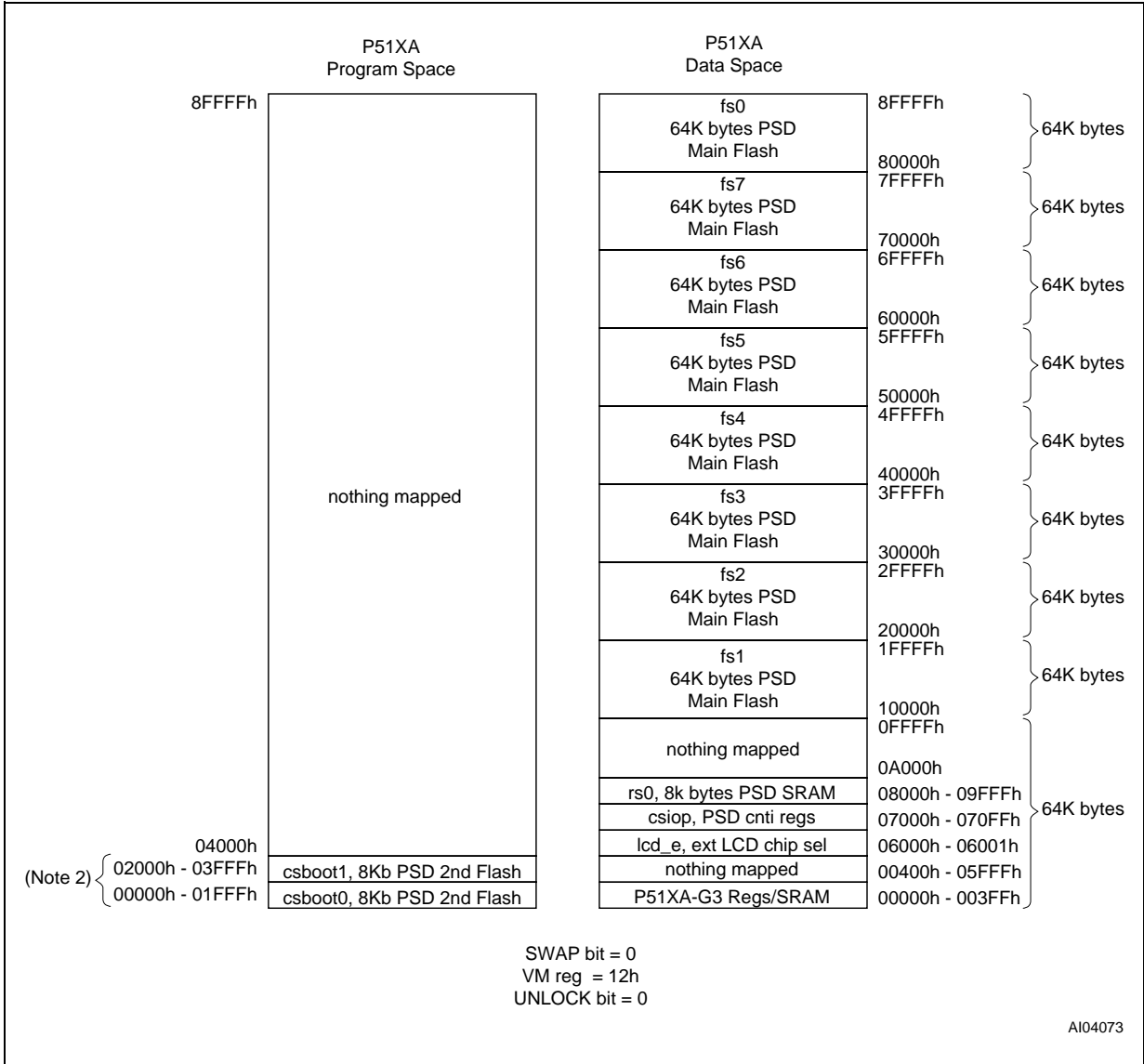
Memory Map

The memory map for this design is a sequence of four steps shown in Figure 45 through to Figure 48. Figure 45 is the memory map at system power-on or system reset. The swap bit and unlock bit are defined as two of the eight PSD page register bits. Here's the sequence after power-up or reset:

■ Figure 45: P51XA boots from secondary Flash memory (csboot0/csboot1) at address 0000, the VM register contains the initial value of 12h from the point-and-click settings in PSDsoft.

■ Figure 45: P51XA performs a checksum of main Flash memory (fs0..fs7) in Data space

■ Figure 45: P51XA downloads to main Flash memory from host computer if needed and validate contents

■ Figure 45: P51XA writes 06h to PSD VM register

■ Figure 46: Main Flash memory has moved to Program space because of 06h in VM register

■ Figure 46: P51XA sets swap bit to logic one (writes to PSD page register)

■ Figure 47: Secondary Flash memory (csboot0/sboot1) has moved out of the MCU address range 0000 to 3FFF and main Flash memory (fs0) has moved into its place because of the swap bit. This swapping action is implemented by qualifying the chip-selects with the swap signal. Also as a result of setting the swap bit, the secondary Flash memory segments csboot2 and csboot3 appear. They cannot be used for data until after the next step.

■ Figure 47: P51XA writes 0Ch to PSD VM register.

■ Figure 48: Secondary Flash memory (csboot0..csboot3) has moved to Data space because of 0Ch in VM register. Now secondary Flash memory segments csboot2 and csboot3 can be used for general data.

Figure 48 shows the final memory map. The P51XA now has a full 512 KBytes of main Flash memory (fs0 .. fs7) in Program space, 16 KBytes secondary Flash memory (csboot2/csboot3) in Data space for general data storage, as well as 8 KBytes of battery backed SRAM (rs0) in Data space. The 16 KBytes of IAP loader code (csboot0/csboot1) is no longer in MCU "executable" position.

If the P51XA needs to update the IAP loader code that resides in secondary Flash memory segments csboot0 and csboot1, it may do so only after setting the unlock bit in the page register. Note that all page register bits are cleared to zero at power-on and at any system reset.

Figure 45. Memory Map at Boot-Up or Reset and During IAP [1]

Note: 1. PSD VM register initially 12h, Main PSD Flash memory in Data space.
2. IAP loader code gets programmed here by JTAG-ISP or conventional programmer tool.

Figure 46. Memory Map just after P51XA writes 06h to PSD VM register
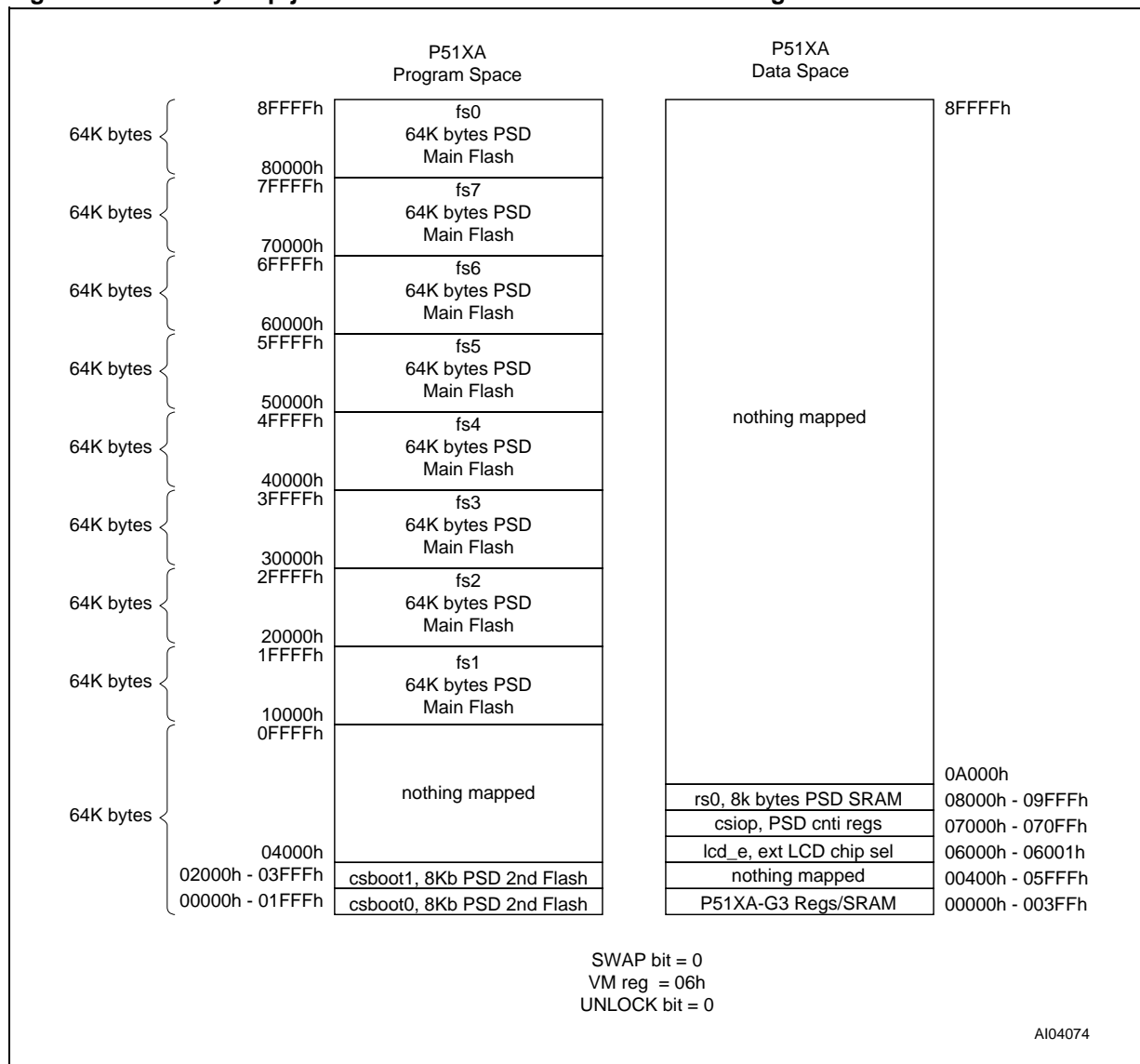
Note: IAP complete, main PSD Flash memory moves to Program space

Figure 47. Memory Map just after P51XA sets swap bit = 1

Note: IAP loader code is swapped away, main PSD Flash memory takes its place.VM reg = 06, unlock = 0

Figure 48. Memory Map just after P51XA writes 0Ch to PSD VM register [1]

```
                    P51XA                           P51XA
                 Program Space                    Data Space

          8FFFFh ┌──────────────┐        8FFFFh ┌──────────────────────────┐
                 │              │               │                          │
64K bytes        │              │               │     nothing mapped       │
                 │ nothing mapped│               │                          │
                 │              │               ├──────────────────────────┤
                 │              │               │ csboot1, 8Kb PSD 2nd Flash│ 86000h - 87FFFh ⎫
          80000h ├──────────────┤               │ csboot0, 8Kb PSD 2nd Flash│ 84000h - 85FFFh ⎬ Note (2)
          7FFFFh │     fs7      │               │ csboot1, 8Kb PSD 2nd Flash│ 82000h - 83FFFh ⎬
64K bytes        │ 64K bytes PSD│               │ csboot0, 8Kb PSD 2nd Flash│ 80000h - 81FFFh ⎭
                 │  Main Flash  │               │                          │
          70000h ├──────────────┤               │                          │
          6FFFFh │     fs6      │               │                          │
64K bytes        │ 64K bytes PSD│               │                          │
                 │  Main Flash  │               │                          │
          60000h ├──────────────┤               │                          │
          5FFFFh │     fs5      │               │                          │
64K bytes        │ 64K bytes PSD│               │     nothing mapped       │
                 │  Main Flash  │               │                          │
          50000h ├──────────────┤               │                          │
          4FFFFh │     fs4      │               │                          │
64K bytes        │ 64K bytes PSD│               │                          │
                 │  Main Flash  │               │                          │
          40000h ├──────────────┤               │                          │
          3FFFFh │     fs3      │               │                          │
64K bytes        │ 64K bytes PSD│               │                          │
                 │  Main Flash  │               │                          │
          30000h ├──────────────┤               │                          │
          2FFFFh │     fs2      │               │                          │
64K bytes        │ 64K bytes PSD│               │                          │
                 │  Main Flash  │               │                          │
          20000h ├──────────────┤        0A000h ├──────────────────────────┤
          1FFFFh │     fs1      │               │ rs0, 8k bytes PSD SRAM    │ 08000h - 09FFFh
64K bytes        │ 64K bytes PSD│               │ csiop, PSD cnti regs      │ 07000h - 070FFh
                 │  Main Flash  │               │ lcd_e, ext LCD chip sel   │ 06000h - 06001h
          10000h ├──────────────┤               │ nothing mapped            │ 00400h - 05FFFh
          0FFFFh │     fs0      │               │ P51XA-G3 Regs/SRAM        │ 00000h - 003FFh
                 │ 64K bytes PSD│               └──────────────────────────┘
          00000h │  Main Flash  │
                 └──────────────┘

                         SWAP bit = 0
                         VM reg  = 0Ch
                         UNLOCK bit = 0

                                                               AI04076
```
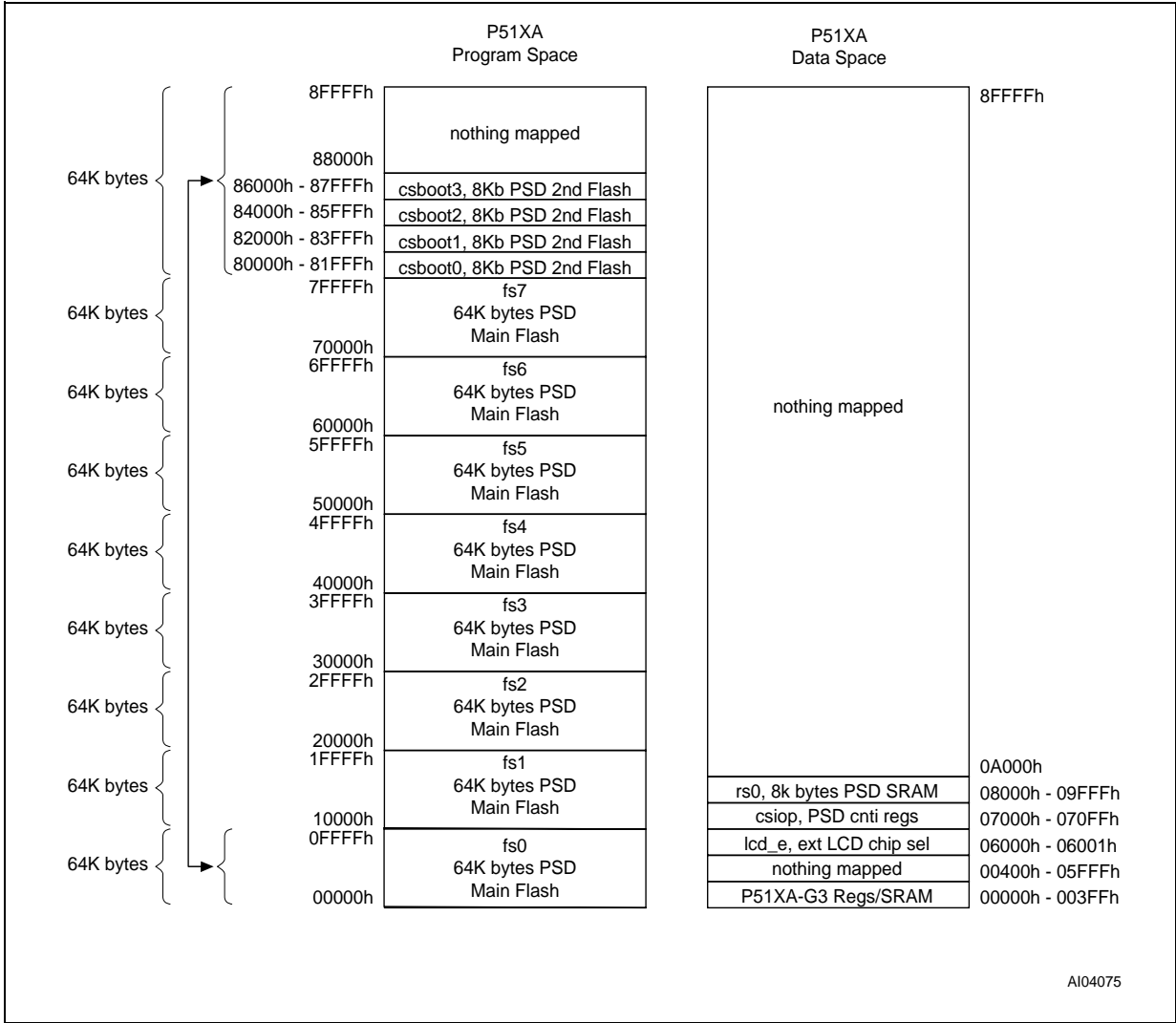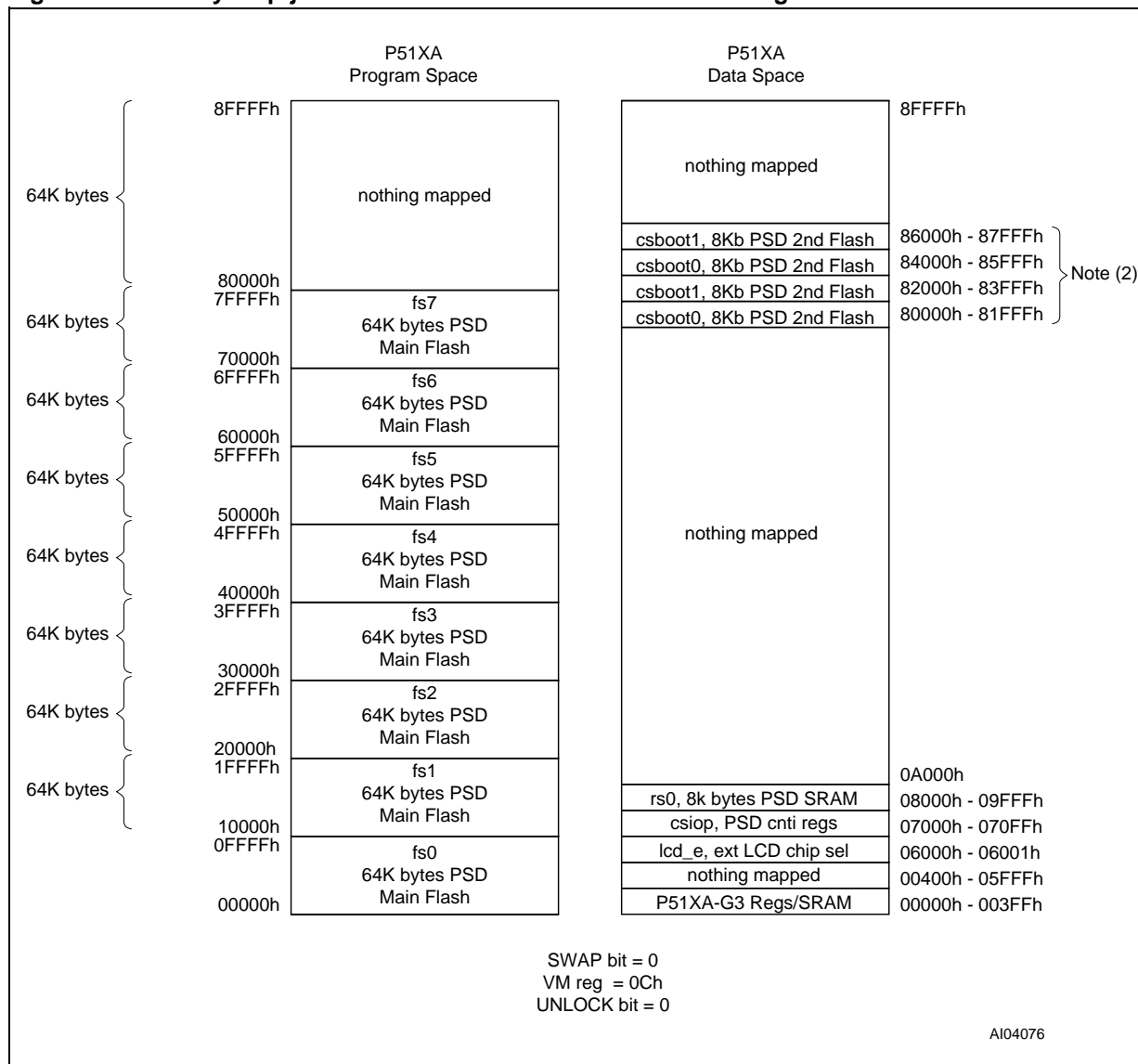
Note: 1. Secondary PSD Flash memory moves to Data space. swap = 1, unlock = 0
      2. IAP loader code only accessible if UNLOCK bit = 1.

Each time this P51XA system gets reset or goes through a power-on cycle, the PSD presents the memory map of Figure 45 to the MCU, and the boot sequence is repeated.

Note: When the P51XA is executing code from the secondary PSD Flash memory (csboot0 and csboot1), and then it sets the swap bit, it is very important that the P51XA firmware linker has set up "synchronized" code in the segment of main PSD Flash memory that replaces the secondary PSD Flash memory. This is necessary to create seamless MCU operation during the actual swap of memory since the P51XA is completely unaware that there is a swap going on. It just continues to fetch opcodes and operands during the memory swap. This requires that the operands and opcodes in main PSD Flash memory that follow the MCU instructions that actually set the swap bit in the secondary PSD Flash memory, are continuous. This means that the remainder of the instructions to complete setting the swap bit is present in main PSD Flash memory so there is continuous operation throughout the memory swapping process.

PSDsoft Express Design Entry

To implement the advanced memory maps of Figure 45 through to Figure 48, invoke PSDsoft *Express*, open the project "logicXA" from the second design example (if not already open). Now pull down the menu 'Project' from the top of the screen, and select 'Save As'. For this third design example, save the second project under the new name "advancXA". Now click on the 'Define PSD Pin/Node Functions' box in the design flow diagram. Click **Next >>** to get to the 'Page Register Definition' screen since no pin assignments need to be changed for this third design.

Page Register Definition

You will need to define two additional PSD page register bits to be used for logic as shown below labeling one bit "swap" and the other bit "unlock".

Figure 49. Page Register Definition
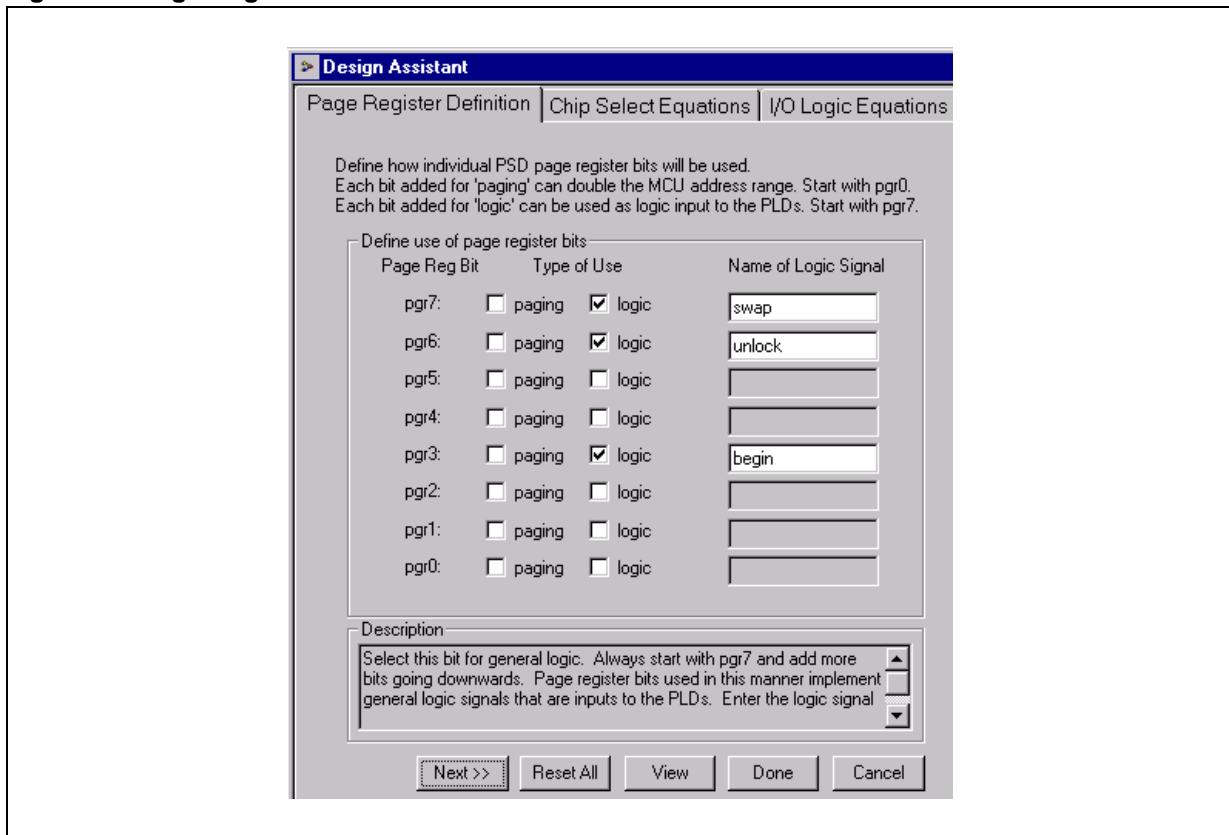
Click **Next >>**.

Chip-Select Equations

The chip-select equations for PSD SRAM (rs0), PSD control registers (csiop), and the external LCD module (lcd_e), and most of the internal PSD memory segments do not change from the second design example. Only chip-selects for main PSD Flash memory segment fs0, and the secondary PSD Flash memory segments csboot0 – csboot3 need to change for this third design because they are affected by memory swapping.

These internal memory chip-selects must be qualified with the page register bit "swap" as shown below. The secondary PSD memory segments, csboot0 and csboot1, must be additionally qualified by "unlock"

to prevent the MCU from inadvertently writing to IAP boot and loader code after IAP is complete. The following illustrates how the chip-selects will look when you enter their definition based the memory maps of on Figure 45 through to Figure 48.

Figure 50. FS0 Address Range

Figure 51. CSBOOT0 Address Range

Figure 52. CSBOOT1 Address Range

Figure 53. CSBOOT2 Address Range

Figure 54. CSBOOT3 Address Range

Notice that these PSD physical memory segments can appear in more that one MCU address space depending on the "swap" and "unlock" qualifiers. Now the memory maps of Figure 45 through to Figure 48 have been implemented. Click **Done** and you should see the main flow diagram.

Finishing the design

There's no need to edit the the ABEL HDL statements since we have not touched the CPLD. Click the 'Fit Design to Silicon' box. After a successful fit, click the 'Merge MCU Firmware' box. You will see an informational dialog box pop up that indicates non-natural address signals were used in PSD chip-select equations. This is because of the "swap" and "unlock bits". PSDsoft displays this message to remind you that your MCU compiler/linker should account for any non-natrurual MCU address signals. Click **OK**, since this does not apply to our example.

Now specify the file name \PSDsoft\Examples\boot_16K.hex for segments csboot0 and csboot1. There is no P51XA firmware in this file, it is used only for illustration. You will find the pattern AAh in csboot0, and the pattern BBh in csboot1. No firmware filename needs to be designated for the main PSD Flash memory segments (fs0 – fs7) since they will be programmed by the P51XA during IAP. No firmware file needs to

be specified for secondary PSD Flash memory segments csboot2 and csboot3 because these will be used for general purpose data written by the P51XA. Click **OK** in the merging screen to create a composite object file for programming. You are now ready to program the PSD as described in the section entitled "Programming the PSD" on page 21.

CONCLUSION

These examples are just three of an endless number of ways to configure the *Easy*FLASH PSD for your system. Concurrent memories with a built-in programmable decoder at the segment level offer excellent flexibility. The ability to expand your system does not require any physical connection changes, as everything is configured internal to the PSD. And finally, the JTAG channel can be used for ISP anytime, and anywhere, with no participation from the MCU. All of these features are crosschecked under the PSDsoft *Express*™ development environment to minimize your effort to design a Flash memory-based system capable of ISP and IAP.

REFERENCES

1. PSD4235G2 Data Sheet

2. *Application Note AN1153— JTAG Information—PSD8XXF* for detailed use of the JTAG channel

For current information on PSD products, please consult our pages on the world wide web:

www.st.com/psd

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

apps.psd@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.