

4

SYSTEM DESIGN

OVERVIEW

The S3C44B0X, SNASUMG's 16/32-bit RISC microcontroller is cost-effective and high performance microcontroller solution for hand-held device and general application. The integrated on-chip functions of S3C44B0X are

- 2.5V Static ARM7TDMI CPU Core with 8KB cache (SAMBAs bus architecture up to 75MHz)
- External memory controller (FP/EDO/SDRAM Control, Chip Select logic)
- LCD Controller (up to 256 color DSTN) with 1-ch LCD-dedicated DMA
- 2-ch general DMAs / 2-ch peripheral DMAs with external request pins
- 2-ch UART / 1-ch SIO (IRDA1.0, 16-byte FIFO)
- 1-ch multi-master IIC-BUS controller & 1-ch IIS-BUS controller
- 5-ch PWM Timers & 1-ch internal timer
- Watch Dog Timer
- 71-bit general purpose I/O ports / 8-ch External Interrupt Source
- Power control : Normal, Slow, Idle and Stop mode
- 8-ch 10-bit ADC
- RTC with calendar function
- On-chip clock generator with PLL

Therefore, you can use S3C44B0X as amount types of system.

APPLICABLE SYTEM WITH S3C44B0X

If your product need to be networked, the S3C44B0X, SNASUMG's 16/32-bit RISC microcontroller can be reduce your system cost. There are sample system, it can be designed with S3C44B0X.

- GPS phone
- PDA (Personal Data Assistance)
- Fish Finder
- Portable Game Machine
- Fingerprint Identification System
- TWM (Two Way Messaging) Terminal
- Car Navigation System
- MP3 Player etc.

MEORY INTERFACE DESIGN

BOOT ROM DESIGN

When system reset, a S3C44B0X access 0x00000000 address. And S3C44B0X should be configure some system variable after reset. Therefore this special code (BOOT ROM image) should be located on address 0x00000000. A boot ROM can have a various width of data bus, and it is controlled by OM[1:0] pins.

Table 4-1. Data Bus Width for ROM Bank 0

OM[1:0]	Data Bus Width
00	8-bit (byte)
01	16-bit (half-word)
10	32-bit (word)
11	Test Mode

ONE BYTE BOOT ROM DESIGN

A design with one byte boot ROM is shown in Figure 4-1.

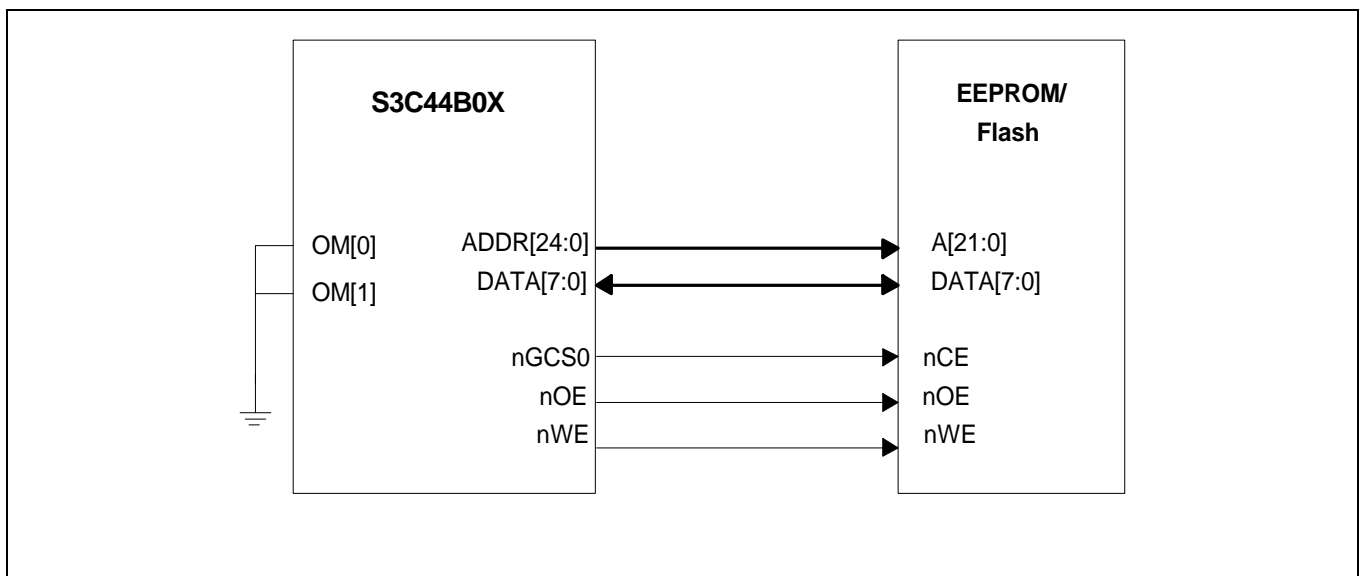


Figure 4-1. One Byte Boot ROM Design

MAKE AND FUSING ONE BYTE ROM IMAGE

When make one byte ROM image, you can use the binary file that made from compile and link.

HALF-WORD BOOT ROM DESIGN WITH BYTE EEPROM/FLASH

A design with half-word boot ROM with byte EEPROM/Flash is shown in Figure 4-2.

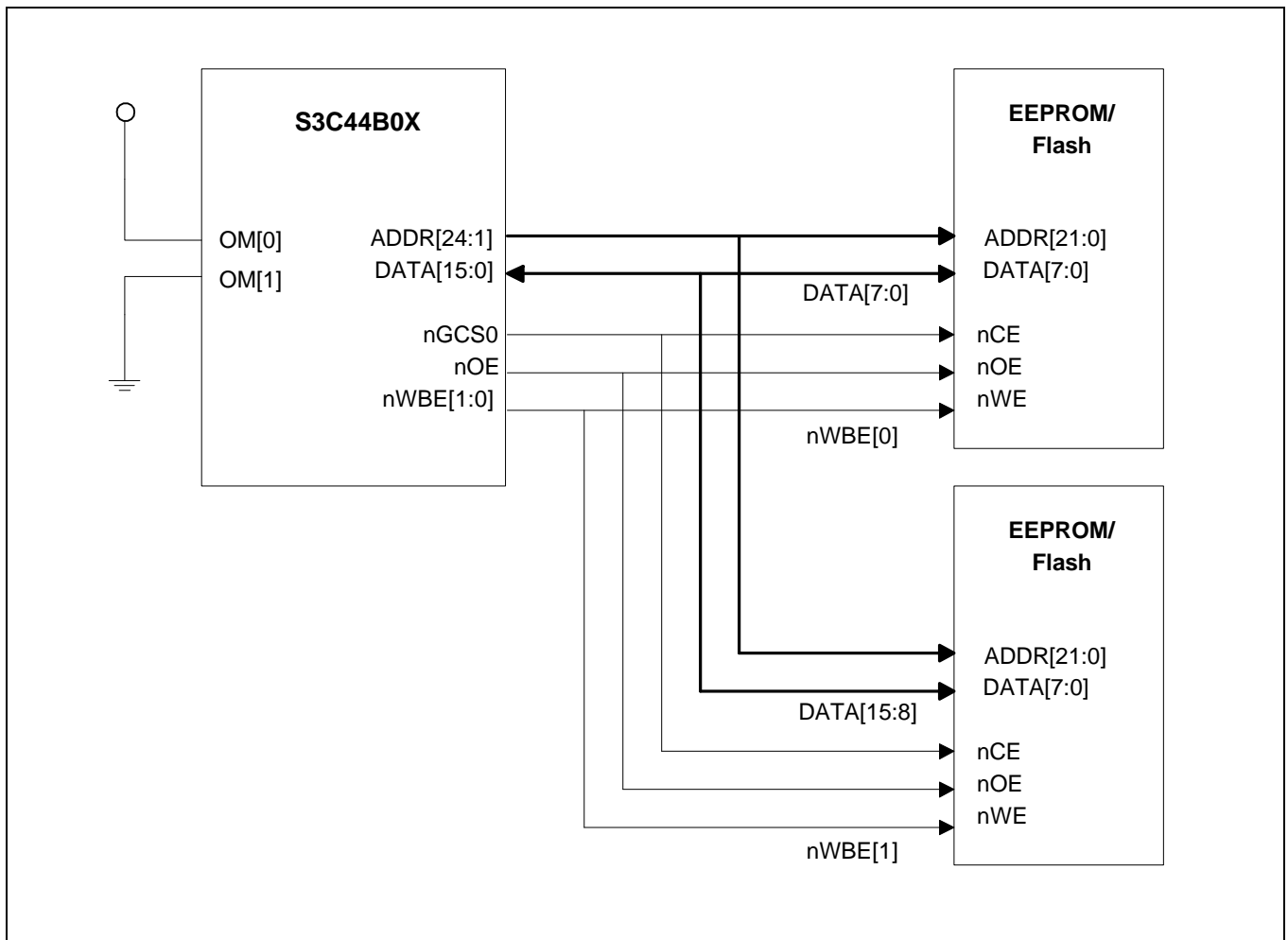


Figure 4-2. The Half-Word Boot ROM Design with Byte EEPROM/Flash

MAKE AND FUSING HALF-WORD ROM IMAGE WITH BYTE EEPROM/FLASH

When make half-word ROM image, you can split two image files, EVEN and ODD.

Table 4-2 Relationship ROM Image and Endian

	Big Endian	Little Endian
DATA[7:0]	Odd	Even
DATA[15:8]	Even	Odd

HALF-WORD BOOT ROM DESIGN WITH HALF-WORD EEPROM/FLASH

A design with half-word boot ROM with byte EEPROM/Flash is shown in Figure 4-3.

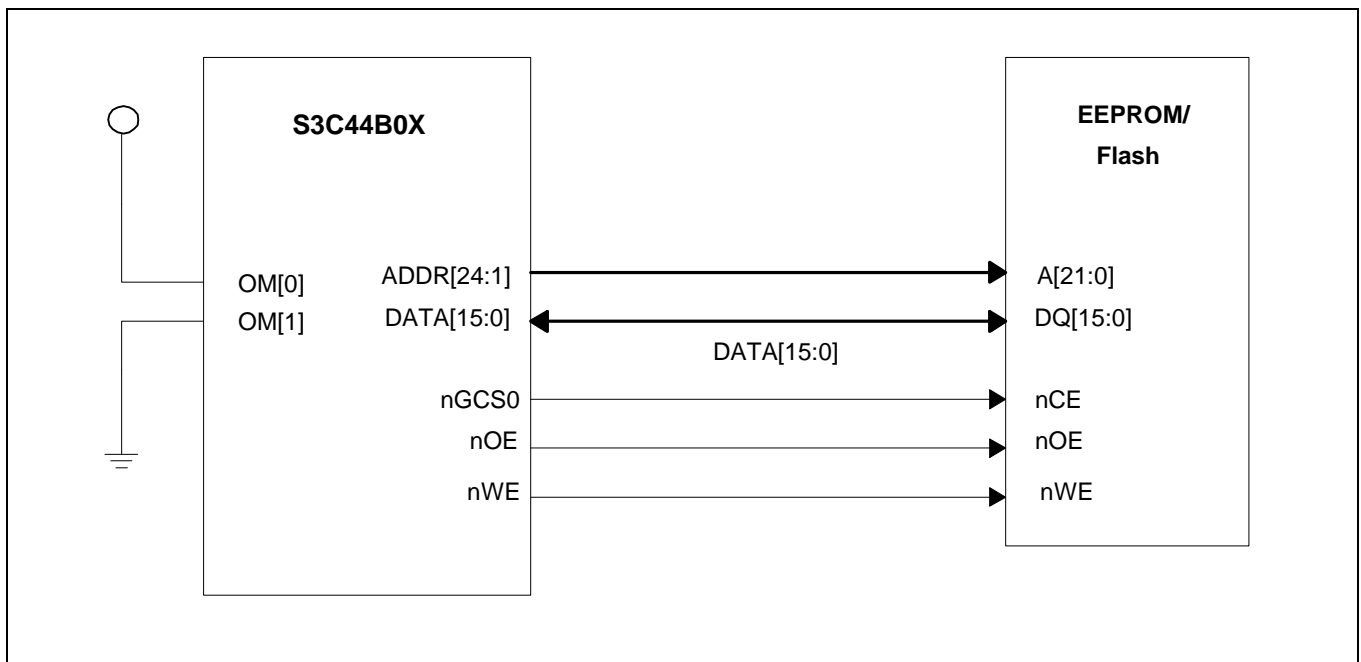


Figure 4-3. The Half-Word Boot ROM Design with Half-Word EEPROM/Flash

WORD BOOT ROM DESIGN WITH HALF-WORD EEPROM/FLASH

A design with word boot ROM with byte EEPROM/Flash is shown in Figure 4-4.

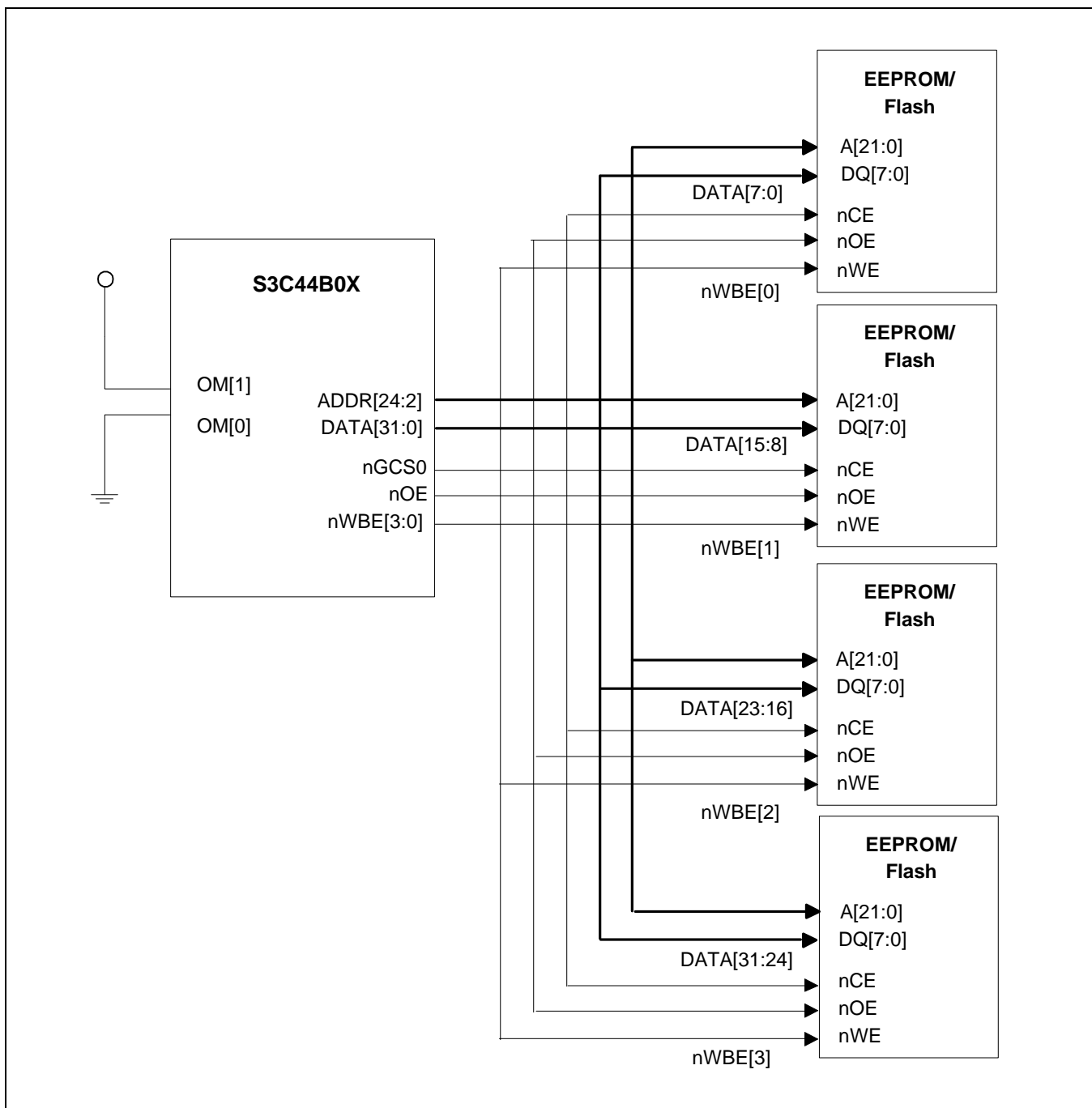


Figure 4-4. The Word Boot ROM Design with Byte EEPROM/Flash

MAKE AND FUSING WORD ROM IMAGE WITH BYTE EEPROM/FLASH

When you make word ROM image, you can split four image file.

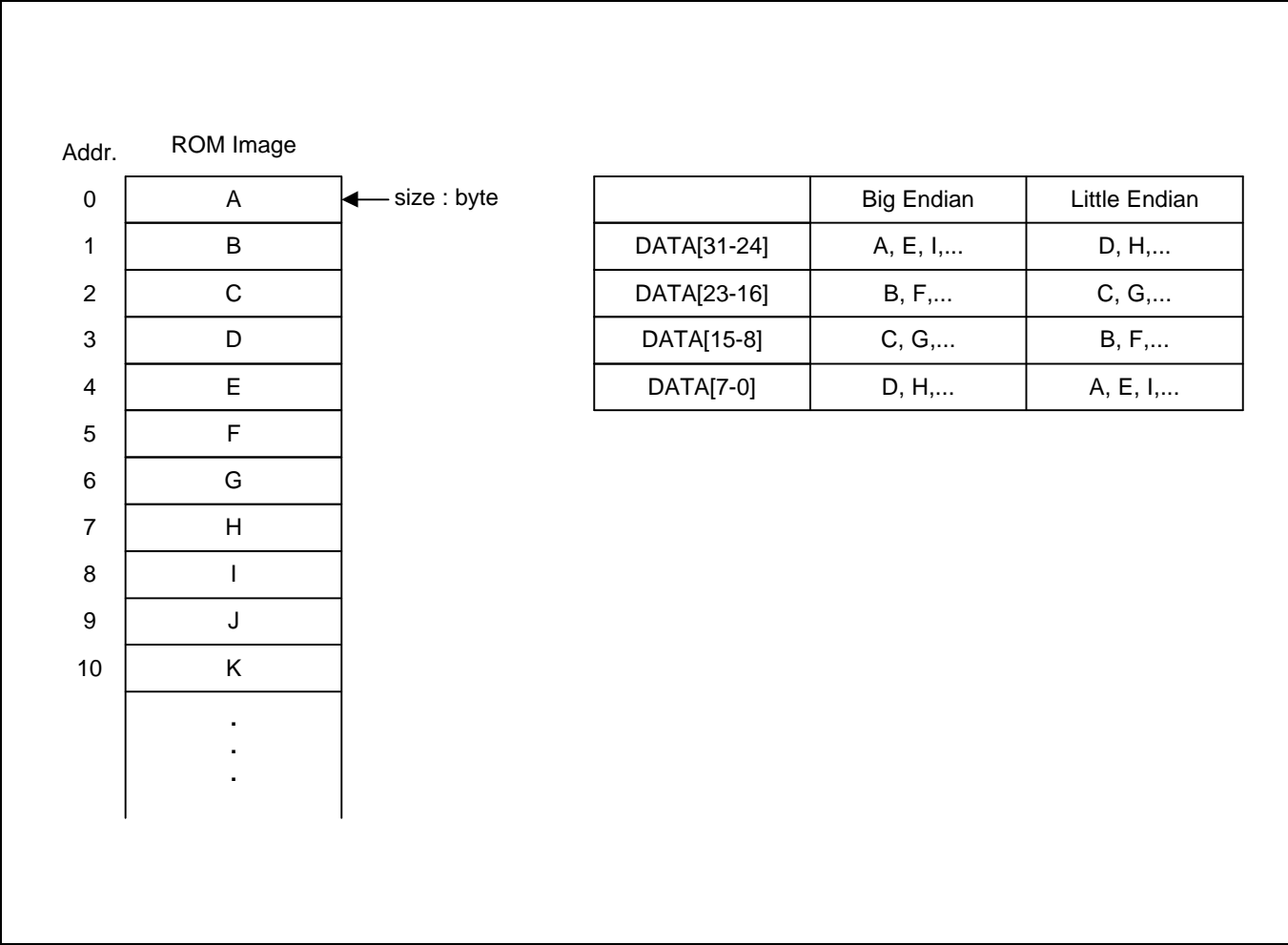


Figure 4-5 Relationship ROM Image and Endian

MEMORY BANKS DESIGN AND CONTROL

The S3C44B0X has 6 ROM/SRAM banks (ROM0 bank for boot ROM) and 2 ROM/SRAM/FP/EDO/SDRAM banks. The system manager on S3C44B0X can control access time, data bus width for each banks by S/W. The access time of ROM/SRAM banks and FP/EDO/SDRAM banks is controlled by BANKCON0~7 and BANKCON6~7 control register on system manager. The type memory of bank6 & 7 has to be same. (example ROM & ROM, SDRAM & SDRAM) The data bus width for each ROM/SRAM/DRAM banks is controlled by BWSCON control register.

The ROM bank 0 is used for boot ROM bank, therefore bank 0 is controlled by H/W, OM[1:0] is used for this purpose.

The control of BWSCON, BANKCON0-7, REFRESH, BANKSIZE, MRSRB6/7 is performed when system reset, by special command, LDMIA and STMIA. Sample code for special register configuration is described below.

Sample code for special register configuration

```

LDR      r0, =SMRDATA
LDMIA    r0, {r1-r13}
LDR      r0, =0x01c80000    ;BWSCON Address
STMIA    r0, {r1-r13}

. . . . .

SMRDATA
DCD 0x22221210    ;BWSCON
DCD 0x00000600    ;GCS0
DCD 0x00000700    ;GCS1
DCD 0x00000700    ;GCS2
DCD 0x00000700    ;GCS3
DCD 0x00000700    ;GCS4
DCD 0x00000700    ;GCS5
;DCD 0x0001002a    ;GCS6 EDO DRAM(Trcd=3,Tcas=2,Tcp=1,CAN=10)
;DCD 0x0001002a    ;GCS7 EDO DRAM(Trcd=3,Tcas=2,Tcp=1,CAN=10)
DCD 0x00018000    ;GCS6 SDRAM(Trcd=2,SCAN=8)
DCD 0x00018000    ;GCS7 SDRAM(Trcd=2,SCAN=8)
DCD 0x00a60000+953;Refresh(REFEN=1,TREFMD=0,Trp=3.5(D)or 4(SD),
;          Trc=5(S), Tchr=3(D),Ref CNT)
DCD 0x0           ;Bank size, 32MB/32MB
DCD 0x20          ;MRSR 6(CL=2)
DCD 0x20          ;MRSR 7(CL=2)

```

ROM/SRAM BANKS DESIGN

The ROM/SRAM banks 1-7, can have a various width of data bus, and the bus width is controlled by S/W. A sample design for ROM/SRAM bank 1-7 is shown in Figure 4-6, Figure 4-7, Figure 4-8 and Figure 4-9.

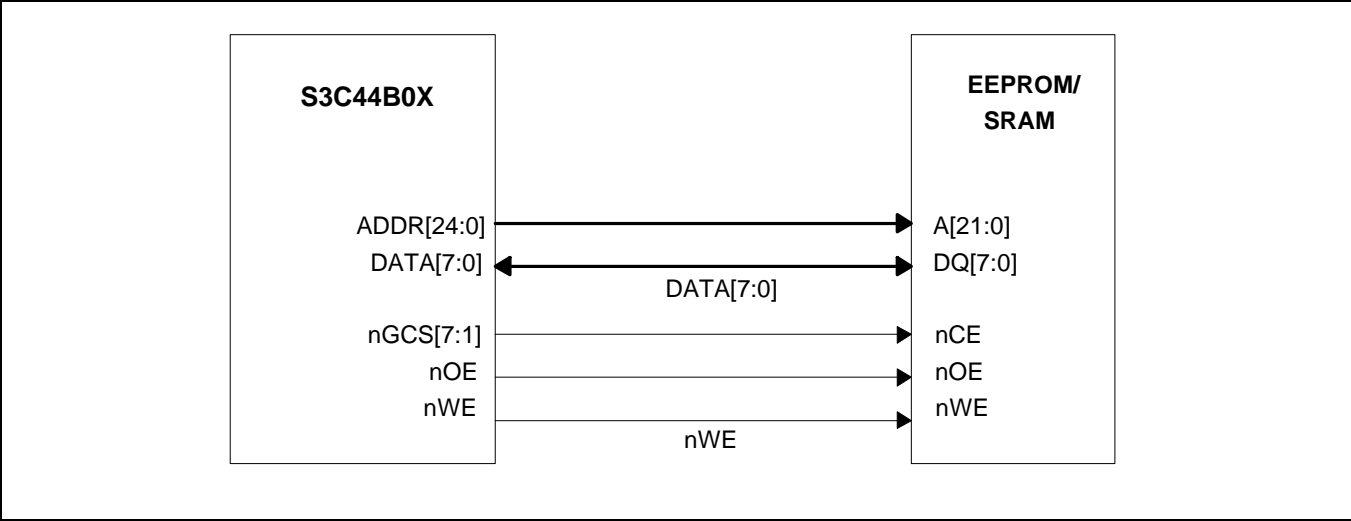


Figure 4-6 One-byte EEPROM/SRAM Banks Design

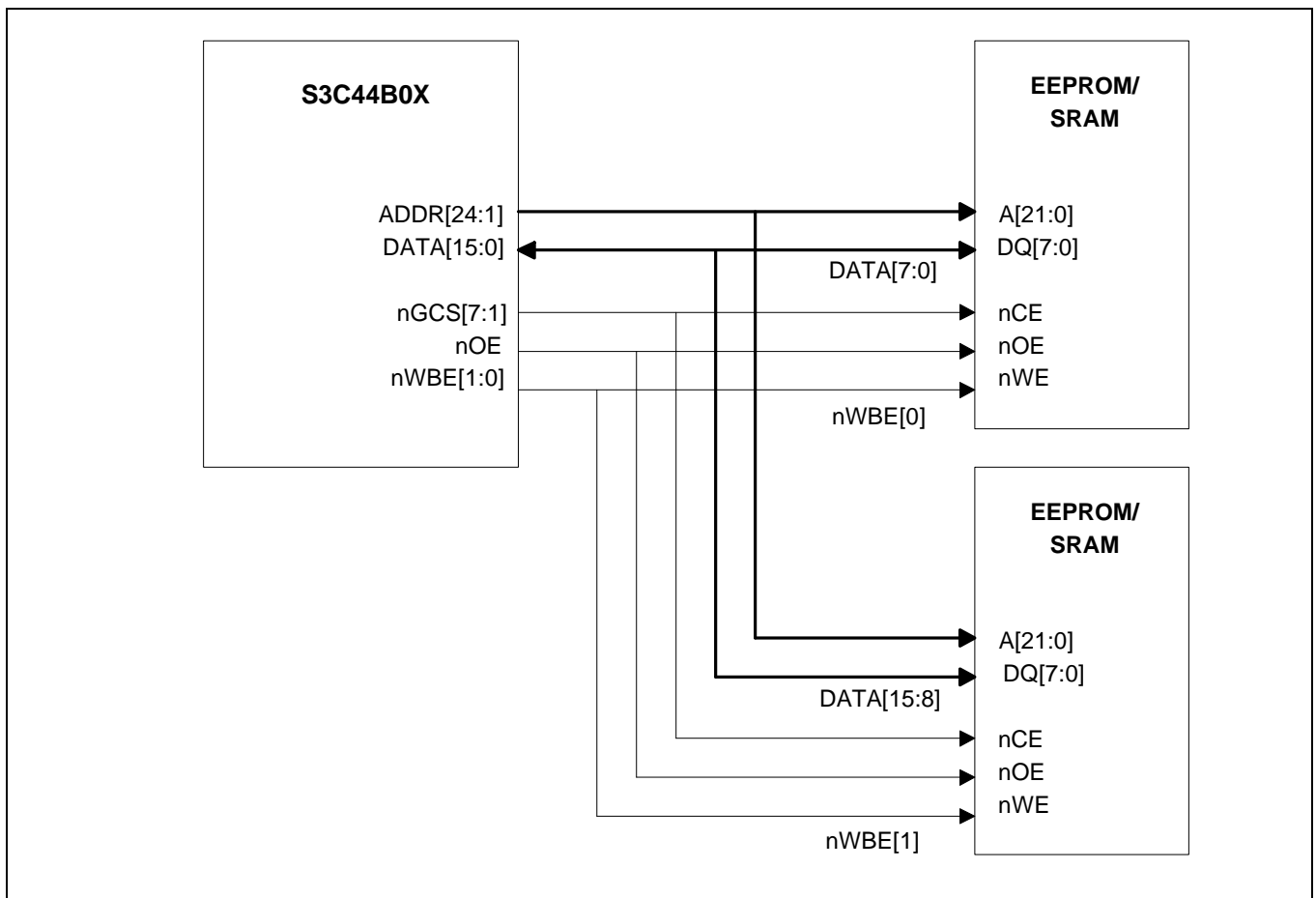


Figure 4-7 Half-word EEPROM/SRAM Banks Design

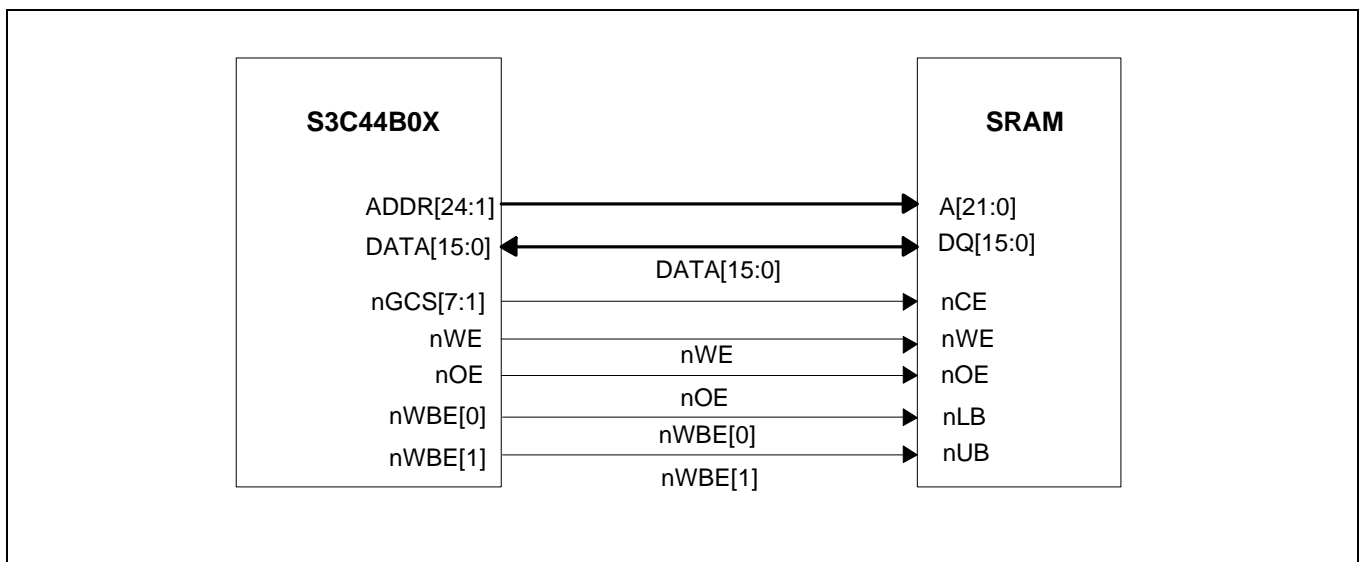


Figure 4-8 Half-word SRAM Banks Design with Half-word SRAM

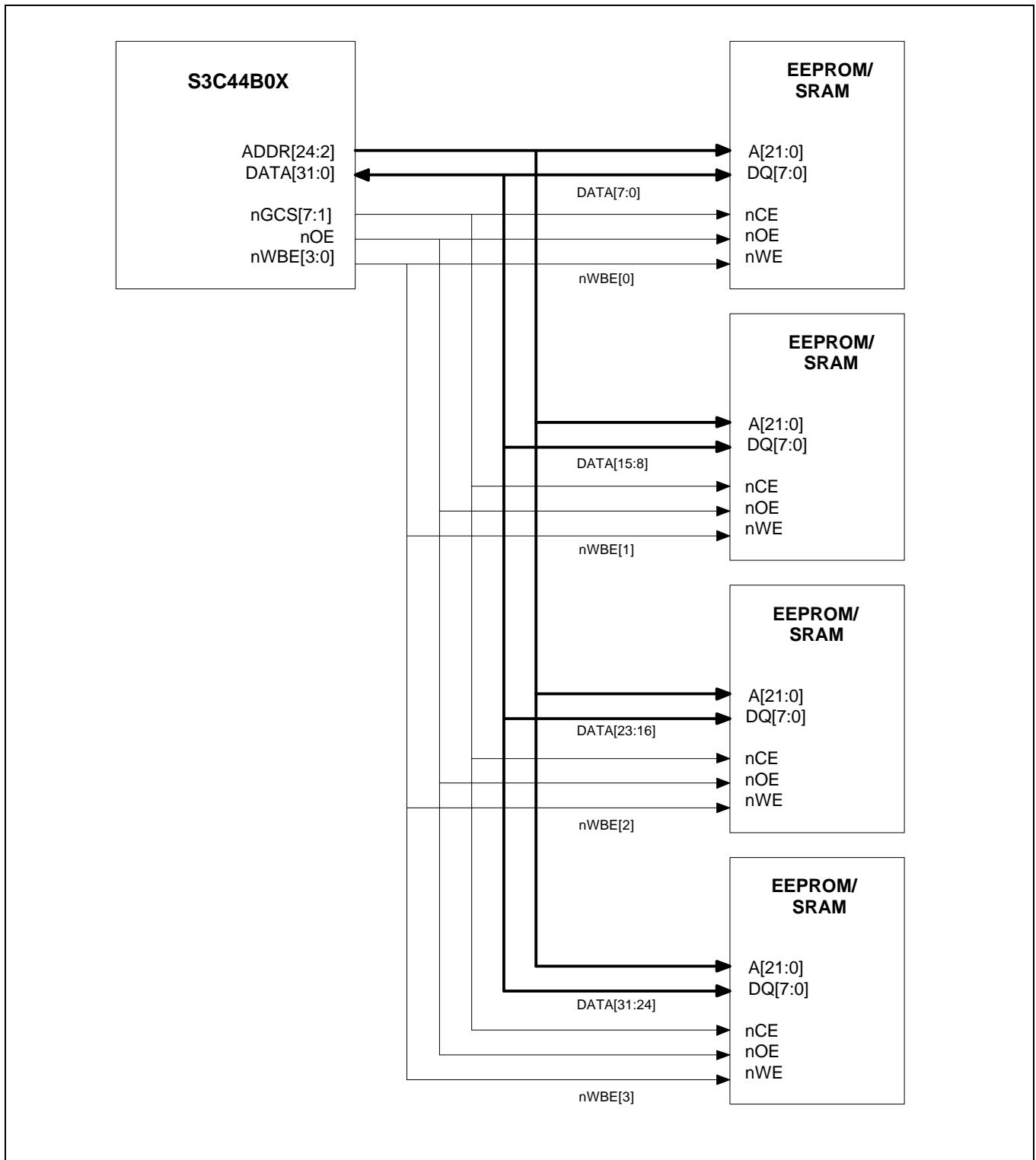


Figure 4-9 Word EEPROM/SRAM banks design

EDO DRAM BANKS DESIGN FOR S3C44B0X

The DRAM banks 6-7, can have a various width of data bus, and the bus width is controlled by S/W, A BWSCON special register set. A sample design for DRAM bank 6-7 is shown in Figure 4-10 and Figure 4-11.

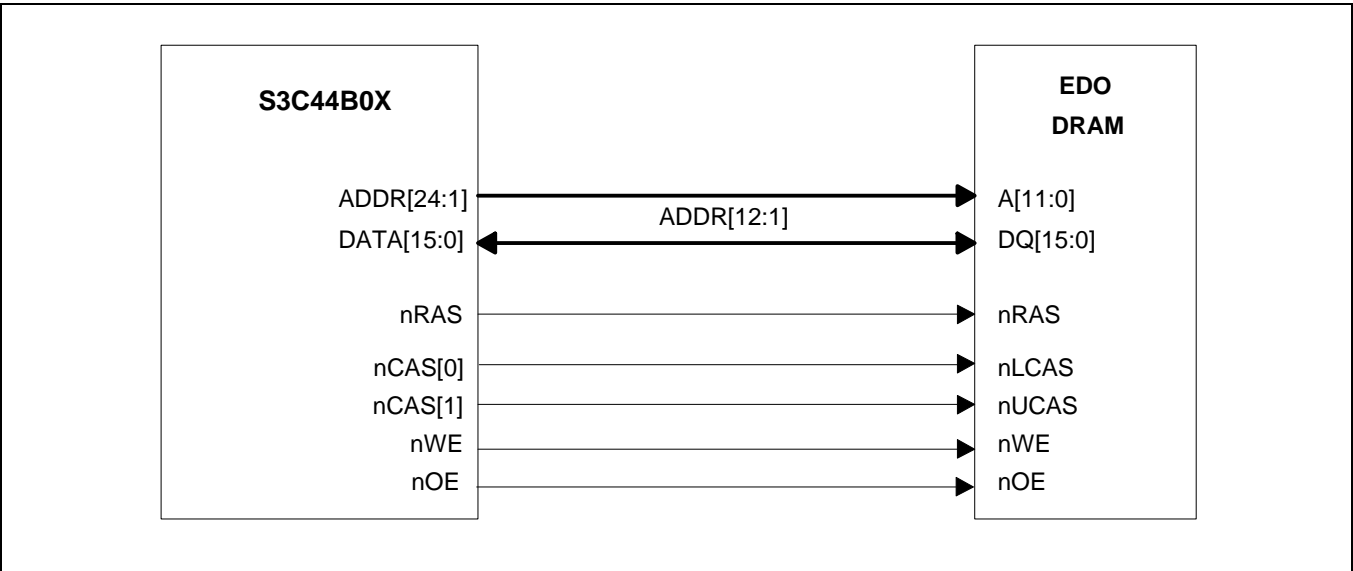


Figure 4-10 Half-Word EDO/Normal DRAM Banks Design

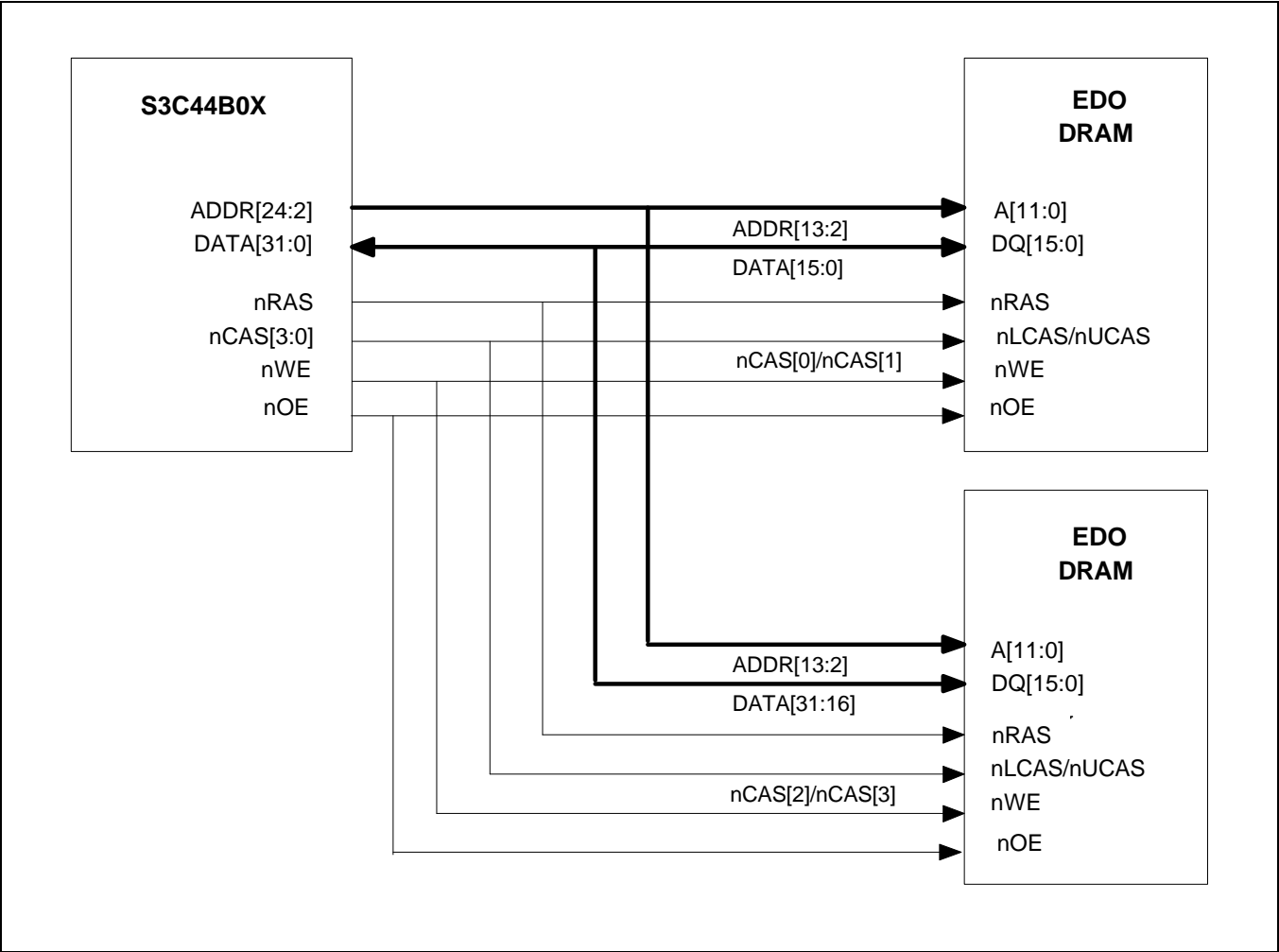


FIGURE 4-11 WORD EDO/NORMAL DRAM BANK

SDRAM BANKS DESIGN FOR S3C44B0X

The S3C44B0X Synchronous DRAM interface features are as follows :

- Maximum column address of SDRAM: 10 bit
- CAS latency: 2/3 cycle

Table 4-3 SDRAM Bank Address configuration

Bank Size	Bus Width	Base Component	Memory Configuration	Bank Address
2MByte	x8	16Mbit	(1M x 8 x 2Bank) x 1	A20
	x16		(512K x 16 x 2B) x 1	
4MB	x8	16Mb	(2M x 4 x 2B) x 2	A21
	x16		(1M x 8 x 2B) x 2	
	x32		(512K x 16 x 2B) x 2	
8MB	x16	16Mb	(2M x 4 x 2B) x 4	A22
	x32		(1M x 8x 2B) x 4	
	x8	64Mb	(4M x 8 x 2B) x 1	A[22:21]
	x8		(2M x 8 x 4B) x 1	
	x16		(2M x 16 x 2B) x 1	A22
	x16		(1M x 16 x 4B) x 1	A[22:21]
	x32		(512K x 32 x 4B) x 1	
16MB	x32	16Mb	(2M x 4 x 2B) x 8	A23
	x8	64Mb	(8M x 4 x 2B) x 2	
	x8		(4M x 4 x 4B) x 2	A[23:22]
	x16		(4M x 8 x 2B) x 2	A23
	x16		(2M x 8 x 4B) x 2	A[23:22]
	x32		(2M x 16 x 2B) x 2	A23
	x32		(1M x 16 x 4B) x 2	A[23:22]
	x8	128Mb	(4M x 8 x 4B) x 1	
	x16		(2M x 16 x 4B) x 1	
32MB	x16	64Mb	(8M x 4 x 2B) x 4	A24
	x16		(4M x 4 x 4B) x 4	A[24:23]
	x32		(4M x 8 x 2B) x 4	A24
	x32		(2M x 8 x 4B) x 4	A[24:23]
	x16	128Mb	(4M x 8 x 4B) x 2	
	x32		(2M x 16 x 4B) x 2	
	x8	256Mb	(8M x 8 x 4B) x 1	
	x16		(4M x 16 x 4B) x 1	

The required SDRAM interface pin is CKE, SCLK, nSCS[1:0], nSCAS, nSRAS, DQM[3:0], ADDR[12]/AP. The sample design with SDRAM is shown in Figure 4-12, and Figure 4-13.

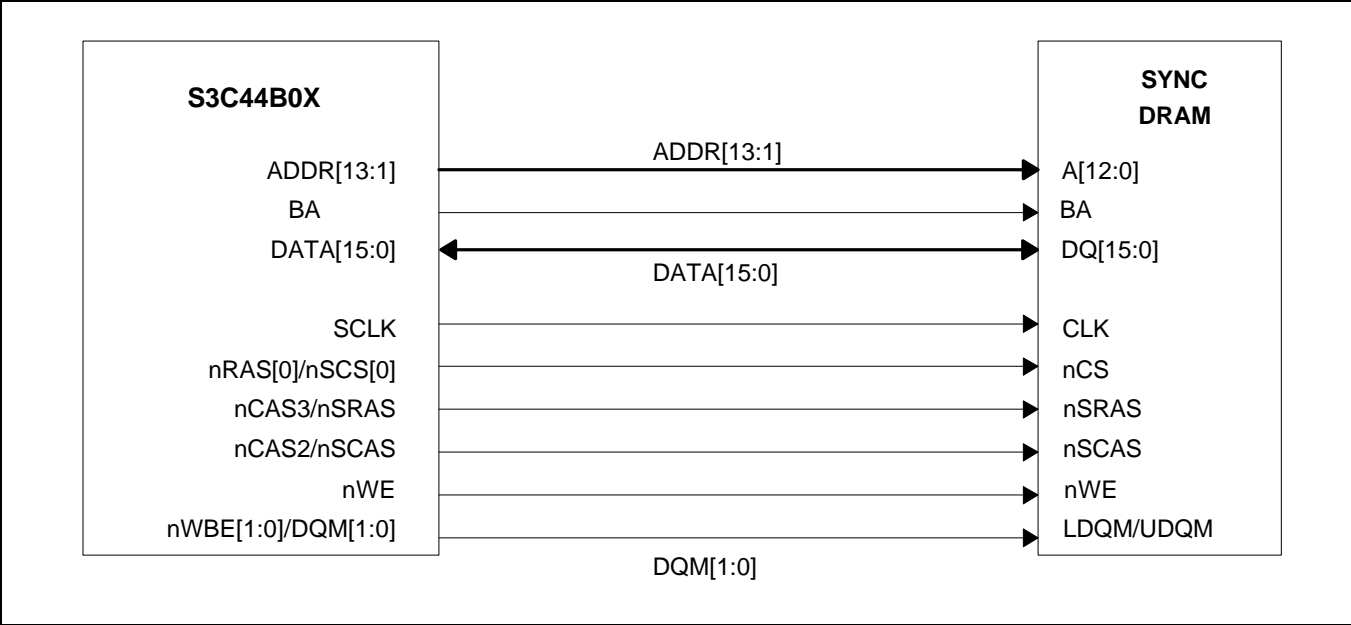


Figure 4-12 Half-word SDRAM Design with Half-word Component

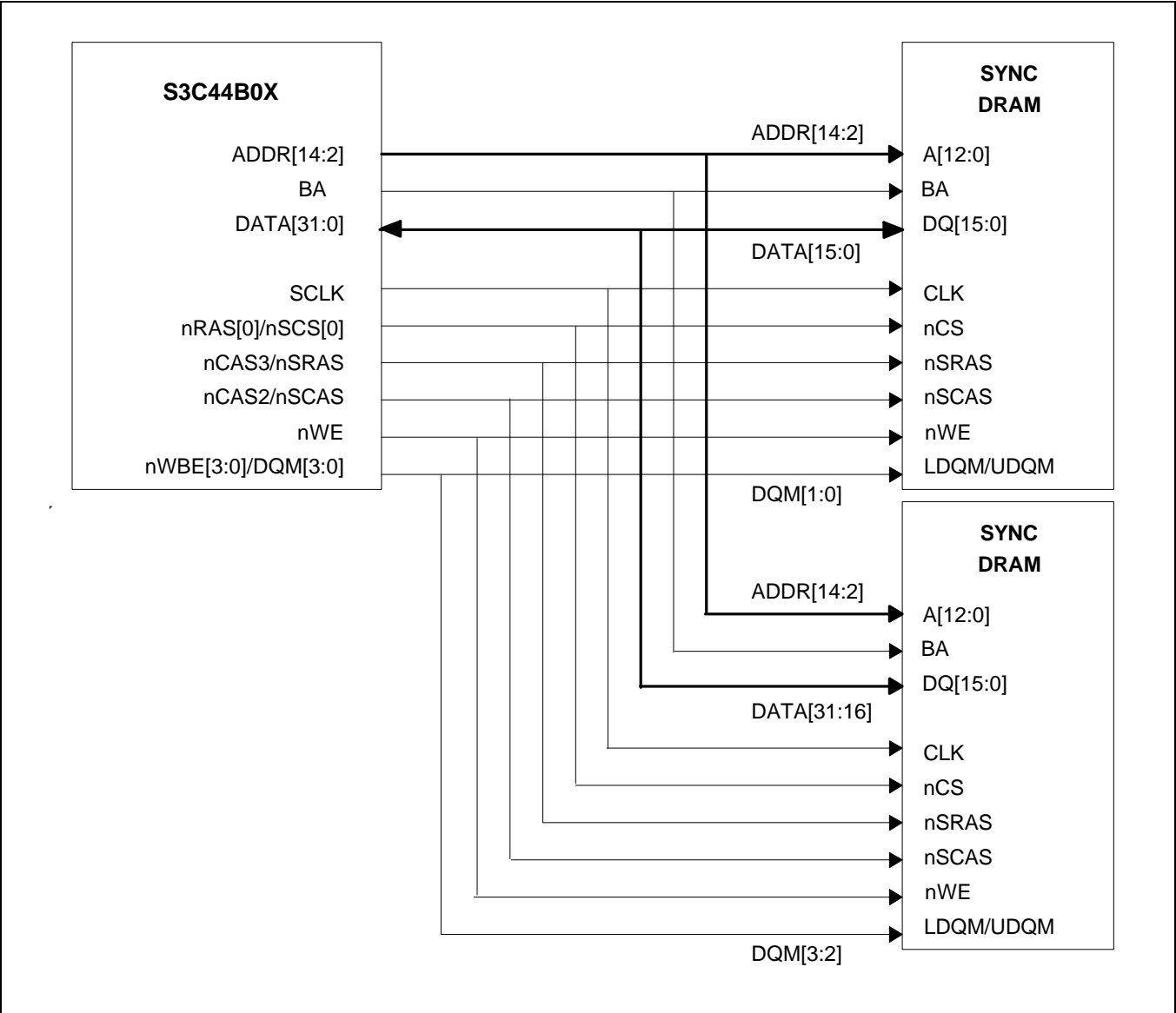


Figure 4-13 Word SDRAM Design with Half-word Component

I/O PORT CONFIGURATION

S3C44B0X has multiplexed input/output/function port pins. The SMDK41100 demo board uses only some functions, then some pins are float or some pins have selectable function port with 0 ohm resistor, therefore users must define the pin's configuration and attach the 0 ohm resistor on the proper place before running the main program.

For reducing the power consumption in SMDK41100, the port state and usage of internal pull-up resistor are decided very carefully. The followings are the sample port configuration for SMDK41100.

Port A : Memory address pins

Port	H/W connection	@Normal	@Stop	@Idle
PA0	OPEN	ADDR0		
PA1	ADDR16	ADDR16		
PA2	ADDR17	ADDR17		
PA3	ADDR18	ADDR18		
PA4	ADDR19	ADDR19		
PA5	ADDR20	ADDR20		
PA6	BA0	ADDR21		
PA7	BA1	ADDR22		
PA8	OPEN	ADDR23		
PA9	OPEN	ADDR24		
PDATA	-	-		
PCONA	-	0x3ff		

Port B : Memory control and output(LED) pins

Port	H/W Connection	@Normal	@Stop(data)	@Idle
PB0	SCKE	SCKE		
PB1	SCLK	SCLK		
PB2	nSCAS	nSCAS		
PB3	nSRAS	nSRAS		
PB4	DQM0	nWBE2/nBE2/DQM2		
PB5	DQM1	nWBE3/nBE3/DQM3		
PB6	OPEN	nGCS1		
PB7	OPEN	nGCS2		
PB8	OPEN	nGCS3		
PB9	LED	Output	Output(High)	Output
PB10	LED	Output	Output(High)	Output
PDATB	-	-	0x600	-
PCONB	-	0x1ff		

Port C : IIS, LCD data and UART control pins.

Not used function pins are defined input port with pull-up resistor enabled to reduce the power consumption. If the UART function is not used the pins should be disabled pull-up resistor, output signal(Tx, nRTS) is defined output port and input signal(Rx, nCTS) is defined input port avoid conflicting the signals of MAX3232.

Port	H/W Connection	@ Normal(data, pull-up)	@ Stop(data, pull-up)	@ Idle(data, pull-up)
PC0	IISLRCK	Input(pull-up enable)		
PC1	IISDO	Input(pull-up enable)		
PC2	IISDI	Input(pull-up enable)		
PC3	IISCLK	Input(pull-up enable)		
PC4	VD7	Input(pull-up enable)		
PC5	VD6	Input(pull-up enable)		
PC6	VD5	Input(pull-up enable)		
PC7	VD4	Input(pull-up enable)		
PC8	OPEN	Input(pull-up enable)		
PC9	OPEN	Input(pull-up enable)		
PC10	nRTS1	Output(High, pull-up disable)		
PC11	nCTS1	Input(pull-up disable)		
PC12	TxD1	Output(High, pull-up disable)		
PC13	RxD1	Input(pull-up disable)		
PC14	nRTS0	Output(High, pull-up disable)		
PC15	nCTS0	Input(pull-up disable)		
PDATC	-	0x5400		
PUPC	-	0xfc00		
PCONC	-	0x11100000		

Port D : LCD data and LCD control pins.

Port	H/W Connection	@ Normal(pull-up)	@ Stop(data, pull-up)	@ Idle(pull-up)
PD0	VD0	VD0(pull-up disable)	Output(High,pull-up disable)	VD0(pull-up disable)
PD1	VD1	VD1(pull-up disable)	Output(High,pull-up disable)	VD1(pull-up disable)
PD2	VD2	VD2(pull-up disable)	Output(High,pull-up disable)	VD2(pull-up disable)
PD3	VD3	VD3(pull-up disable)	Output(High,pull-up disable)	VD3(pull-up disable)
PD4	VCLK	VCLK(pull-up disable)	Output(High,pull-up disable)	VCLK(pull-up disable)
PD5	VLINE	VLINE(pull-up disable)	Output(High,pull-up disable)	VLINE(pull-up disable)
PD6	VM	VM(pull-up disable)	Output(High,pull-up disable)	VM(pull-up disable)
PD7	VFRAME	VFRAME(pull-up disable)	Output(High,pull-up disable)	VFRAME(pull-up disable)
PDATD	-	-	0xff	-
PUPD	-	0xff	0xff	0xff
PCOND	-	0xaaaa	0x5555	0xaaaa

Port E : UART control pins and ENDIAN pin.

PE8 port should be defined input port with disabled pull-up resistor, because the PE8 pin is connected to GND for little endian mode in SMDK41100 demo board.

Port	H/W Connection	@Normal(pull-up)	@Stop(pull-up)	@Idle(pull-up)
PE0	OPEN	Input(pull-up enable)		
PE1	TxD0	TxD0(pull-up disable)		
PE2	RxD0	RxD0(pull-up disable)		
PE3	OPEN	Input(pull-up enable)		
PE4	OPEN	Input(pull-up enable)		
PE5	OPEN	Input(pull-up enable)		
PE6	OPEN	Input(pull-up enable)		
PE7	OPEN	Input(pull-up enable)		
PE8	ENDIAN	ENDIAN(pull-up disable)		
PDATE	-	-		
PUPE	-	0x106		
PCONE	-	0x28		

Port F : IIS and SIO control pins.

PF0 and PF1 pins should be defined disabled pull-up resistor input port, because these pins are connected pull-up resistors by hardware.

Port	H/W Connection	@Normal(pull-up)	@Stop(pull-up)	@Idle(pull-up)
PF0	IIC_SCL	Input(pull-up disable)		
PF1	IIC_SDA	Input(pull-up disable)		
PF2	OPEN	Input(pull-up enable)		
PF3	OPEN	Input(pull-up enable)		
PF4	OPEN	Input(pull-up enable)		
PF5	SIOTxD	Input(pull-up enable)		
PF6	SIORDY	Input(pull-up enable)		
PF7	SIORxD	Input(pull-up enable)		
PF8	SIOCLK	Input(pull-up enable)		
PDATEF	-	-		
PUPF	-	0x3		
PCONF	-	0x0		

Port G : External interrupt pins.

The SMDK41100 demo board is using PG4 and PG5 ports as external interrupt ports with button.

Port	H/W Connection	@Normal(pull-up)	@Stop(pull-up)	@Idle(pull-up)
PG0	EINT0	Input(pull-up enable)		
PG1	EINT1	Input(pull-up enable)		
PG2	EINT2	Input(pull-up enable)		
PG3	EINT3	Input(pull-up enable)		
PG4	EINT4	Input(pull-up disable)		
PG5	EINT5	Input(pull-up disable)		
PG6	OPEN	Input(pull-up enable)		
PG7	OPEN	Input(pull-up enable)		
PDATG	-	-		
PUPG	-	0x30		
PCONG	-	0x0		

Special pull-up resistor control register

In normal operation, the pull-up resistor should be disabled but in stop mode it is enabled for power consumption.

Connection	@Normal	@Stop	@Idle
SPUCR0	Pull-up disable	Pull-up enable	Pull-up disable
SPUCR1	Pull-up disable	Pull-up enable	Pull-up disable
Hz@STOP	Previous state	High-impedance	Previous state
SPUCR	0x7	0x0	0x7

NOTE : To reduce power consumption users shall consider the state of pins and refer to the table below.

Usage of Pins	Pull-up resistor + Data
Unused input port pins	Pull-up enable
Normal output port pins	Pull-up disable + Data High
Function(address, data, control) pins	Pull-up disable

LCD CONNECTION WITH S3C44B0X

The S3C44B0X LCD interface example circuit is as follows :

- UG-32F04(320x240 mono STN LCD) from SAMSUNG DISPLAY DEVICES CO.,LTD (refer to Figure 4-14)
 . TL497CAN can be used to make VEE(-25V).
- UG-24U03A(320x240 mono STN LCD) from SAMSUNG DISPLAY DEVICES CO.,LTD (refer to Figure 4-15)
 . VEE is generated by the circuit on LCD module.
 . VL is 2.4V typically.
 . DISPON H : display on, L : display off
 . nEL_ON H : EL off L : EL on
- KHS038AA1AA-G24 (256 color STN LCD) from KYOCERA Co. (refer to Figure 4-16)
 . DISP signal can be made using I/O port, or power control circuit, or nRESET circuit.
 . V1-V5 can be made using the power circuit recommended by the LCD specification.

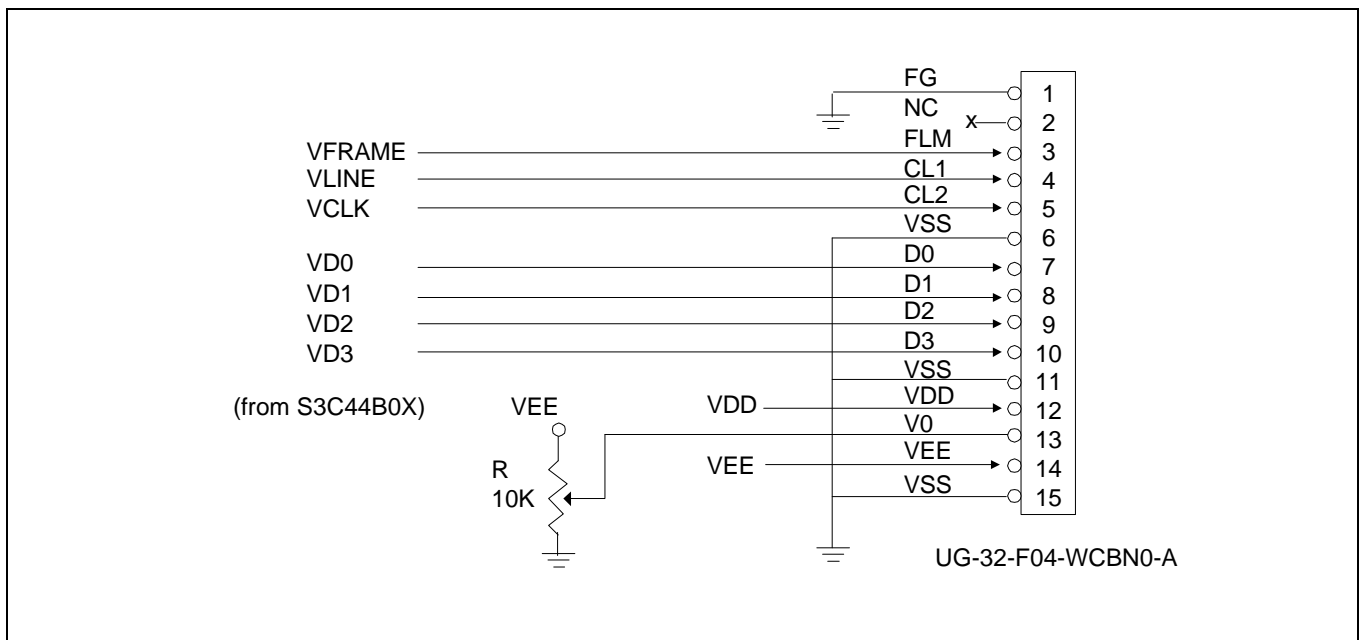


Figure 4-14 UG-32F04 connection with S3C44B0X(320x240 mono STN LCD)

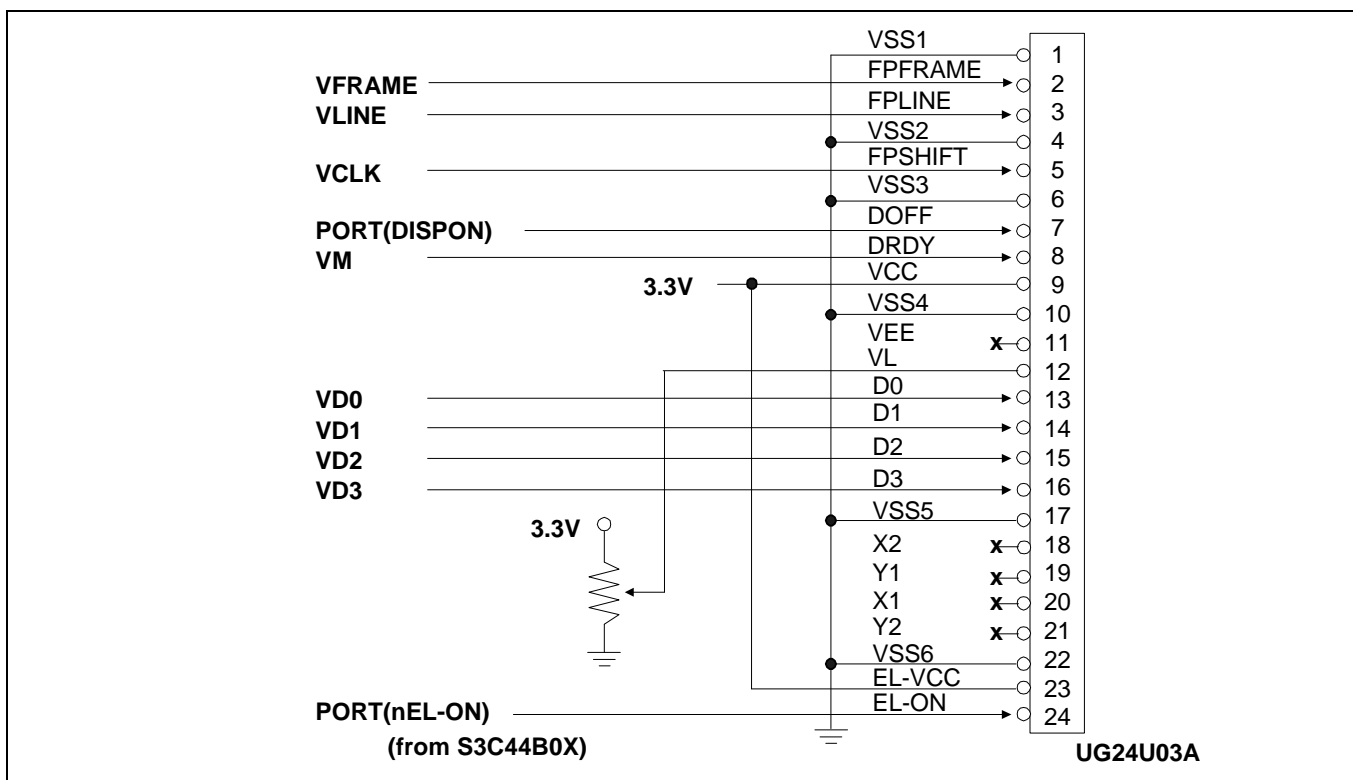


Figure 4-15 UG24U03A connection with S3C44B0X (320x240 mono STN LCD)

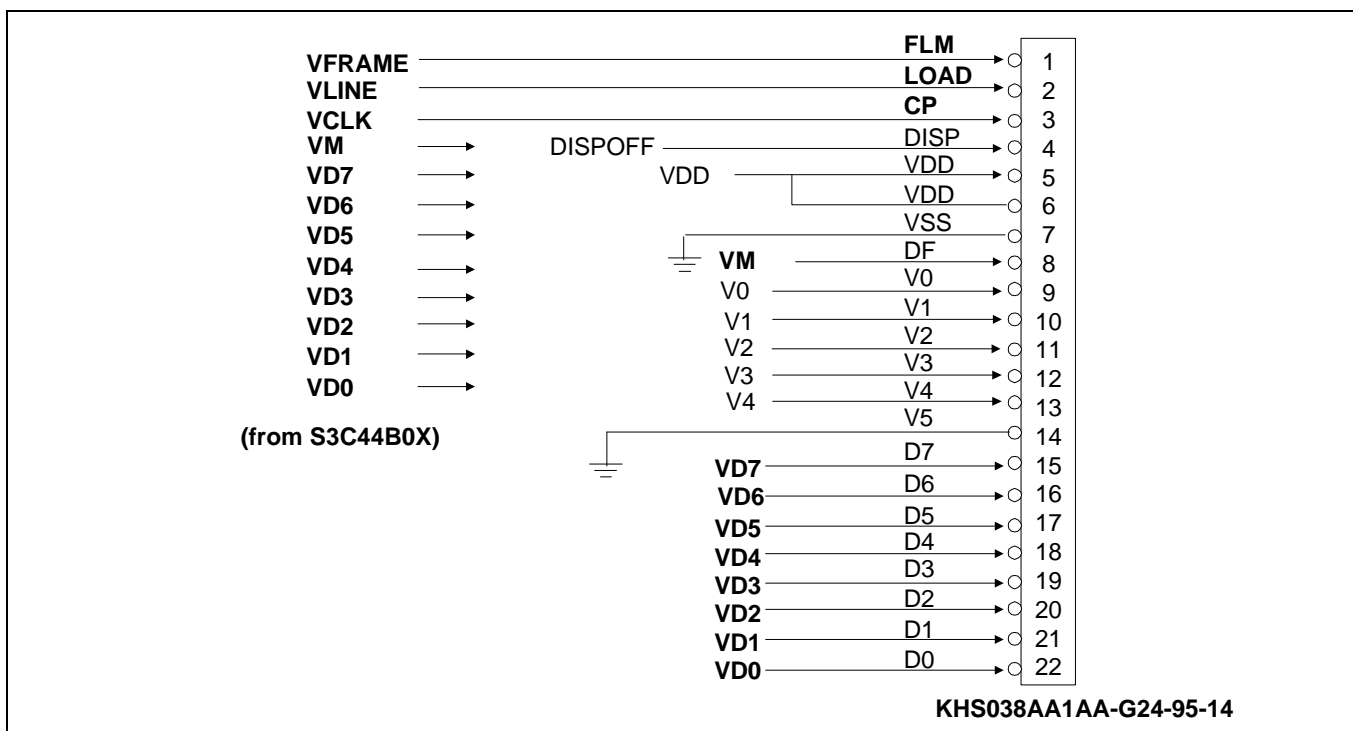


Figure 4-16 KHS038AA1AA-G24 connection with S3C44B0X (256 color STN LCD)

SYSTEM DESIGN WITH DEBUGGER SUPPORT

EmbeddedICE Macrocell and EmbeddedICE Interface

The S3C44B0X has an EmbeddedICE macrocell that provides debug support fro ARM cores. The EmbeddedICE macrocell is programmed in serial using the TAP(Test Access Port) controller on the S3C44B0X. The EmbeddedICE interface is a JTAG protocol conversion unit. It translates a debug protocol message generated by the debugger into a JTAG signal which is sent to the built-in serial and parallel ports.

JTAG port for EmbeddedICE Interface

When you build a system with the S3C44B0X EmbeddedICE interface, you should design a JTAG port for EmbeddedICE interface. Usually, the interface connector is a 14-way box header, and this plug is connected to the EmbeddedICE interface module using 14-way IDC cable.

The JTAG port signals, nTRST,TDI,TMS,TCK have to be connected pulled-up register(10K ohm) externally.

When you operate normal mode without EmdeddedICE, nRESET signal on S3C44B0X is connected nTRST via JP1 (jumper1). In debugger mode, nRESET signal on S3C44B0X is surely seperated nTRST via JP1 (jumper1).

The pin configuration and a sample design are described in Figure 4-17, 4-18, respectively.

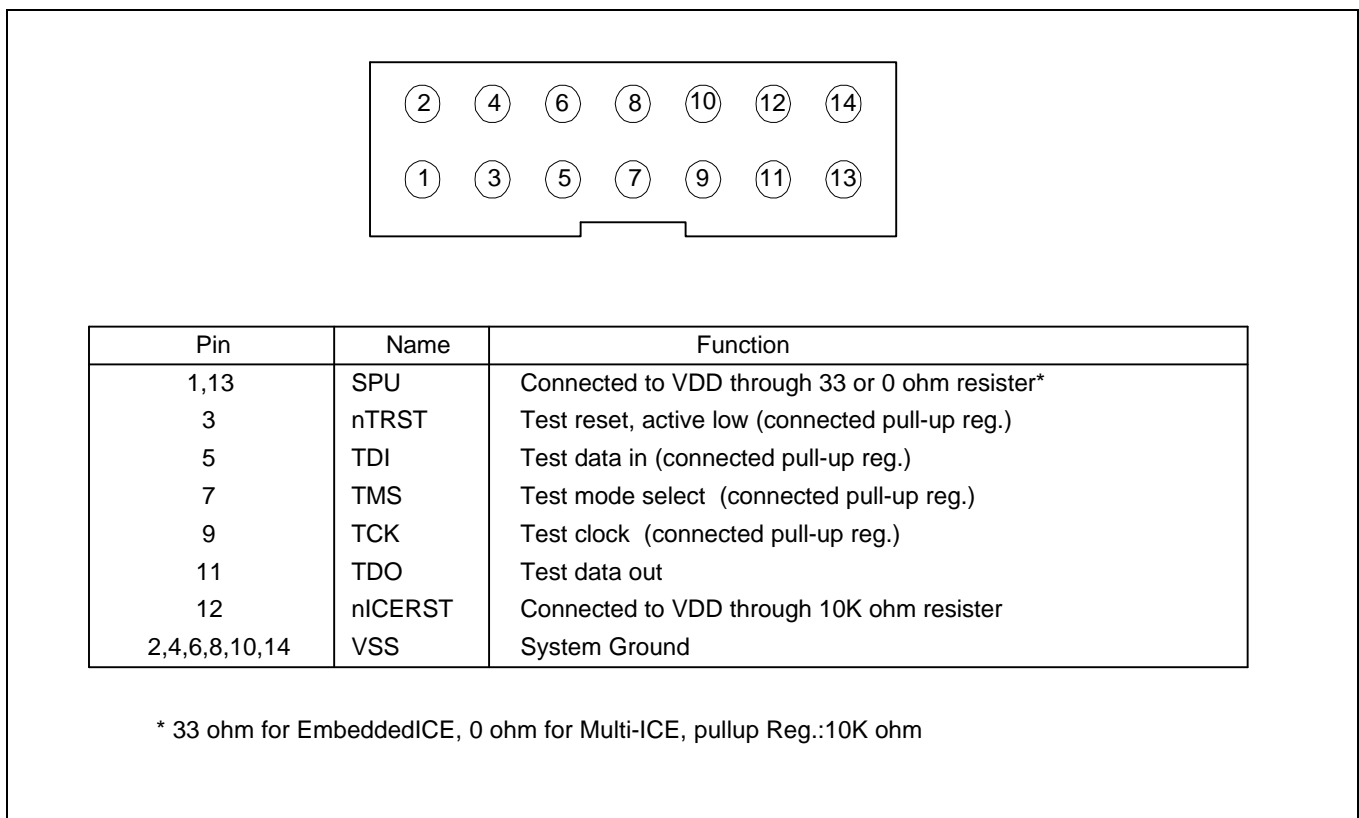


Figure 4-17 EmbeddedICE Interface JTAG Connector

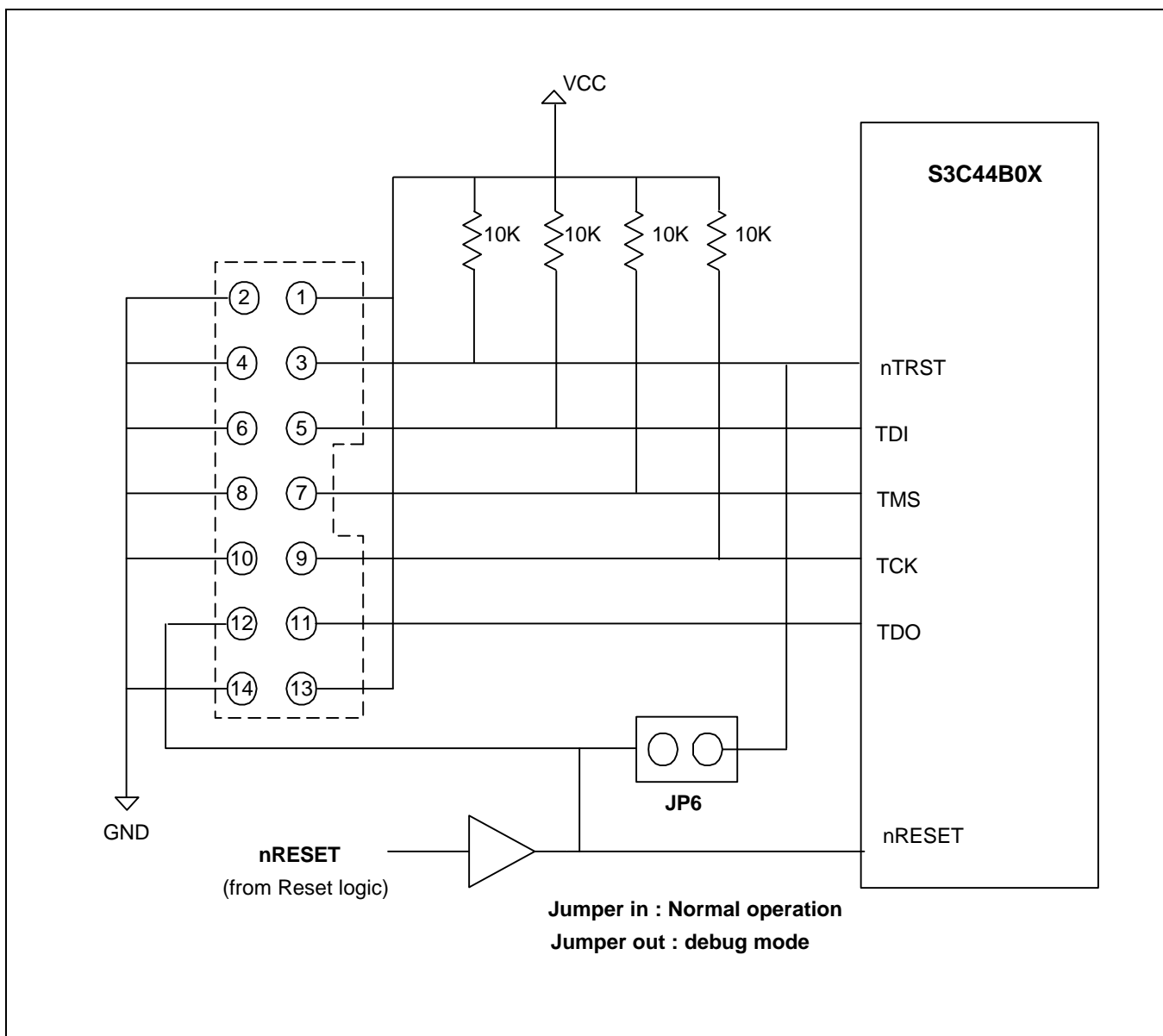


Figure 4-18 EmbeddedICE Interface Design Example

CHECK LIST FOR SYSTEM DESIGN WITH S3C44B0X

When you design a system with the S3C44B0X, you should check a number of items to build a good system. The check list is described below.

- The OM[3:0] and ENDIAN pin have to be configured.
- To run the CPU without using JTAG(ICE), connect nTRST and nRESET pin.
- If EXTCLK pin is used for MCLK, XTAL0 has to be connected to VDD. If XTAL0 pin is used for MCLK, EXTCLK has to be connected to VDD.
- If an input pin is unused, connect the pin to VDD or GND. If the pin is float, S3C44B0X may not operate.
- nGCS6,7 do not support DRAM & SDRAM combination,

Please configure memory type to below combination in bank6 & 7 :

DRAM & DRAM, SDRAM & SDRAM , SRAM & SRAM, SRAM & DRAM, SRAM & SDRAM.

NOTES

5

PSOSYSTEM BOARD SUPPORT PACKAGE

OVERVIEW

The pSOSystem operating system is a modular, high-performance real-time operating system designed specifically for embedded microprocessors. It provides a very complete, multitasking environment based on open system standards.

The pSOSystem operating system is designed to meet three overriding objectives:

- Performance
- Reliability
- Ease-of-Use

The result is a fast, deterministic, yet accessible, system software solution.

The pSOSystem is designed as a set of software components that comprise an operating system for an embedded application. To use pSOSystem, you write an application for the target and link the pSOSystem software components to the application at building time. When the application is downloaded to the target, pSOSystem executes as the operating system.

The pSOSystem is used in a cross-development environment, where you develop the application on a *host* system and then download and run it on a *target* system. The host system is linked to the target system by a serial line or an Ethernet connection depending on the corresponding hardware support.

This chapter provides a brief guide to develop pSOSystem application, and explains how to assemble, build, download, and run pSOSystem application on the S3C44B0X Evaluation Board. Especially, It introduces the pSOSystem BSP (Board Support Package) for S3C44B0X Evaluation Board, which make your pSOSystem application to be able to run on our hardware platform.

As you read this chapter, you may also need to refer to the other manuals in standard documentation set, provided by ISI, for more detailed information.

- pSOSystem System Concepts
- pSOSystem Getting Started
- pSOSystem Advanced Topics
- pSOSystem Programmer's Reference
- pSOSystem System Calls
- pROBE+ User's Guide

SYSTEM ARCHITECTURE

A pSOSystem application consists of the following main elements:

- Application codes, which you write for a particular application.
- The pSOSystem operating system components, which include, but are not limited to, the following system libraries:
 - The pSOS+ Kernel
 - The pROBE+ Target-Level Debugger
 - The pREPC+ Run-Time C Library
 - The pHILE+ File System Manager
 - The pNA+ TCP/IP and UDP Network Manager
- A board-support package (BSP), which is an additional software module to provide the interface between the OS/Application and target hardware platform.

The system architecture is shown below:

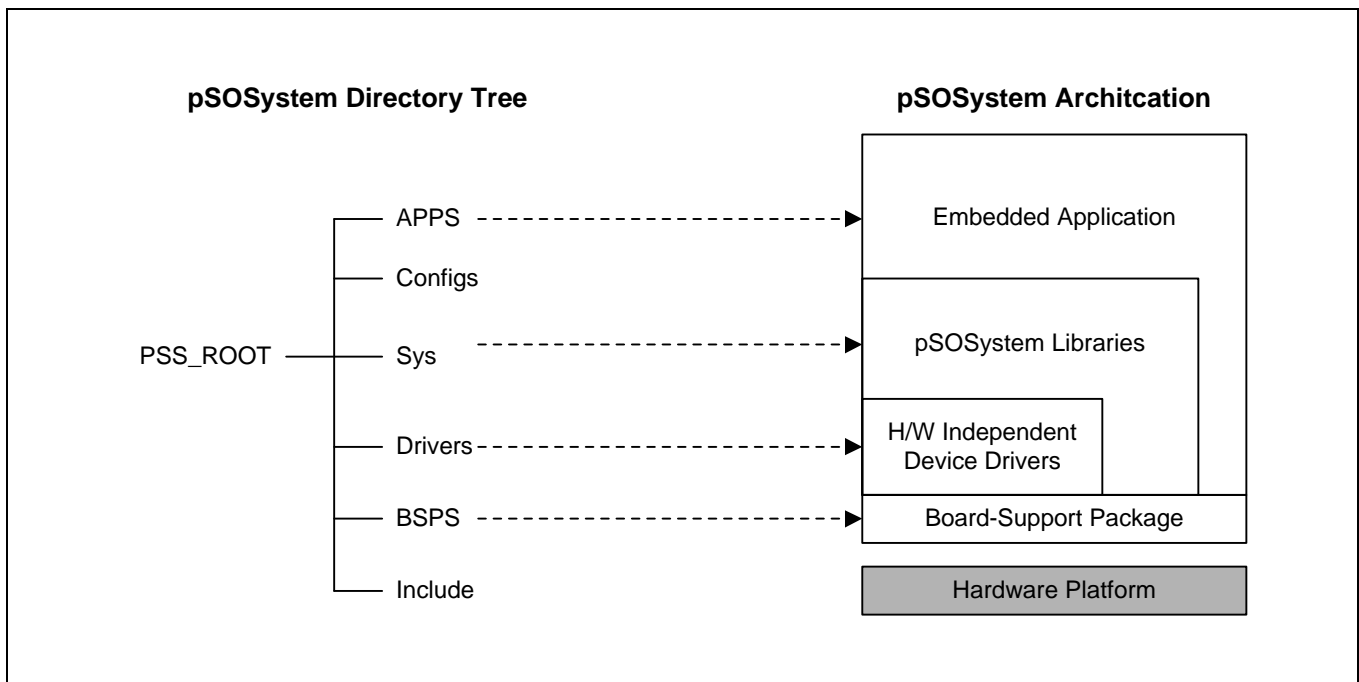


Figure 5-1. System Architecture

DEVELOPMENT ENVIRONMENT DIRECTORY

The pSOSystem development environment locates under a root directory, called as PSS_ROOT, in your host after installation. The root directory consists of several sub-directories, each of which contains the source files, include files or object libraries relative to one layer in pSOSystem architecture, as shown above.

The following table describes the sub-directories under the pSOSystem root directory.

Root	Sub-directory	Description
PSS_ROOT	apps	It has a number of application working directories where you build the pSOSystem application executable image. Each application working directory contains the application source codes, a pSOSystem configuration header file (<i>sys_conf.h</i>) and a driver configuration file (<i>drv_conf.c</i>)
	config	A sub-directory contains the pSOSystem application startup codes and the configuration files for various pSOSystem components.
	sys	A sub-directory contains the operating system components which can be compiled into libraries (<i>libsys.xxx</i> and <i>libsysxx.xxx</i>) and then linked into any application.
	driver	A sub-directory contains some hardware-independent device drivers.
	bsps	It contains a collection of sub-directories, each of which contains a board support package (BSP) that corresponds to a specific hardware platform.
	include	A sub-directory contains the include files that serve as the interface to many parts of pSOSystem such as device drivers and components.

STEP IN pSOSystem APPLICATION DEVELOPMENT

When you develop a pSOSystem application, you should take the following steps:

1. Install the pSOSystem in your host system and set up the host environment.
2. Build the system libraries, *libsys.xxx* and *libsysxx.xxx* (necessary only after installation or version updates).
3. Develop the BSP for the specific hardware platform on which your application will run, and build the BSP library, *libbsp.xxx*, (necessary only after system updated or BSP modified).
4. Write the *application codes* in a working directory, and optionally edit the following files:
 - *sys_conf.h*
 - *drv_conf.c*
 - the makefile (If you use ARM Project Manager to build application, you may not use this file)
5. Link with the system library and BSP library to build the application executable image.
6. Download the executable image to the target platform and run it.

SETTING UP THE HOST ENVIRONMENT

To configure the host system, you need to set up some environment variables. You can edit a batch file with name, such as **envarm.bat**, to set the these environment variables and execute this batch file prior to starting your pSOS-related work; or you can do it by modifying the **envarm.ksh** file if you has installed the pRISM+. A sample batch file is shown below.

```
SET HOST=win32

rem *****
rem * Set up the root path of pSOSystem and corresponding utilities *
rem *****
SET PSS_ROOT=C:\isiarm\pssarm.223
SET PATH=%PSS_ROOT%\bin\win32; %PSS_ROOT%\mksnt;%PATH%

rem *****
rem * Replace the next line with the path to your target BSP.      *
rem *****
SET PSS_BSP=%PSS_ROOT%\bsps\41100

rem *****
rem * Replace the next line with the token for your BSP_TYPE      *
rem * Valid tokens are 32l, 32b, 16l, or 16b                      *
rem *****
SET BSP_TYPE=32B

rem *****
rem * Set up the ARM SDT path                                     *
rem *****
SET ARMINC=C:\isiarm\arm211a\include
SET ARMLIB=C:\isiarm\arm211a\lib\embedded
SET PATH=C:\isiarm\arm211a\bin;%PATH%
```

In above settings, the PSOS environment related variables are described in the table below:

Variable Name	Description
HOST	Defines the name of sub-directory, which containing make utilities, under %PSS_ROOT%\bin
PSS_ROOT	Points to the top of the pSOSystem directory tree on your system.
PSS_BSP	Path to the pSOSystem Board Support Package (BSP).
BSP_TYPE	Execution model of your particular ARM processor. This defines ARM (32) or Thumb (16) mode and big (b) or little (l) endian.
PATH	This variable points to the pSOSystem directory of executable files specific to building pSOSytem based application.

BUILDING THE PSOS SYSTEM LIBRARIES

After you install the pSOSystem in your host system and set up the host environment, the next step is to build the system libraries. These libraries contain all pSOSystem system components, such as the pSOS+ kernel, the pNA+ TCP/IP manager, the pROBE+ debugger, and so on.

You must build the pSOSystem libraries when you first install the pSOSystem software, and you must rebuild the libraries whenever you install new distribution files.

For the ARM architecture, there are several different system libraries built. Each one corresponds to one of ARM execution modes. Separate libraries are built for C and C/C++ applications as well. The table below outlines all of the library names and the corresponding language/execution model. When building application, the correct library is automatically picked from the **BSP_TYPE** environment variable you set above.

Library Name	Description
libsys.32l	32-bit ARM mode, little endian, C only
libsys.32b	32-bit ARM mode, big endian, C only
libsys.16l	16-bit Thumb mode, little endian, C only
libsys.16b	16-bit Thumb mode, big endian, C only
libsysxx.32l	32-bit ARM mode, little endian, C/C++
libsysxx.32b	32-bit ARM mode, big endian, C/C++
libsysxx.16l	16-bit Thumb mode, little endian, C/C++
libsysxx.16b	16-bit Thumb mode, big endian, C/C++

To build the system libraries, enter the following commands in MS-DOS window:

- cd %PSS_ROOT%\sys\os
- psosmake clean
- psosmake

The **psosmake** command automatically picks up the **makefile** in current directory to complete the required system library building operations.

S3C44B0X EVALUATION BOARD BSP

INTRODUCTION

The board-support package (BSP) is a software layer which provides the interface between pSOS+ kernel/application and hardware platform so as to make the pSOSystem application to be able to run on a specific hardware platform. The BSP contains a collection of hardware-specific functions, which include:

1. Target system hardware initialization during system booting, such as
 - System memory configuration
 - Stack setup for each CPU operating mode, and install exception handlers
 - Peripherals initialization, and so on.
2. Exception handlers
3. Lower-level device drivers.
 - Tick timer driver, used by pSOS+ for task scheduling
 - Serial port driver, to provide polled interface (used by pROBE+) and interrupt-driven interface (used by pSOS+ and pREPC+) for serial port communications.

This document describes the pSOSystem Board Support Package for the Samsung S3C44B0X Evaluation Board. The board uses the S3C44B0X microcontroller which contains an ARM7TDMI CPU core as well as a large number of integrated peripherals such as serial port controllers, timers, interrupt related hardware, memory interface hardware, LCD controller, and others.

The pSOSystem Board Support package for S3C44B0X Evaluation Board provides the low level startup code and drivers necessary to adapt pSOSystem target software to the S3C44B0X Evaluation Board. It can also be used as a basis for a custom board support package for other hardware platforms based on S3C44B0X microcontroller.

The S3C44B0X Evaluation Board BSP supports a tick timer and two serial channels. The BSP and all of the drivers associated with it support big-endian ARM mode execution. Application written for the S3C44B0X Evaluation Board can be debugged using the ARM Ltd. Embedded ICE along with the ARM Debugger for Windows (ADW) on PC. The pROBE+ debugger is also available for application debugging.

INSTALLATION

The S3C44B0X Evaluation Board BSP is delivered as an **ZIP** file which can be installed in the pRISM+ top directory. To install the S3C44B0X Evaluation Board BSP, first extract the '41100_save.bat' file from the BSP zip file and run it to back up some original files that are going to be affected. After that, unwind the ZIP file into proper directories in pRISM+ installation.

A list of the files installed by this package is found in follows.

FILE LIST

The following files are provided as main part of the S3C44B0X Evaluation Board BSP:

File	Usage
PSS_ROOT\bsps\41100\bsp.h	BSP definition file
PSS_ROOT\bsps\41100*.mk	S3C44B0X Evaluation Board BSP specific make files for application
PSS_ROOT\bsps\41100\libbsp.32b	S3C44B0X Evaluation Board BSP library to be linked with applications
PSS_ROOT\bsps\41100\src\board.a	Assembler header file for S3C44B0X Evaluation Board hardware
PSS_ROOT\bsps\41100\src\board.h	C header file for S3C44B0X Evaluation Board hardware
PSS_ROOT\bsps\41100\src\board.c	Provides the functions for board specific hardware initialization used by <i>init.s</i> , and hardware specific information used in pSOSystem initialization process.
PSS_ROOT\bsps\41100\src\bpdialog.c	Provides the dialog functions used in pSOSystem setup.
PSS_ROOT\bsps\41100\src\bspcfg.c	Delivers configuration information from application to BSP.
PSS_ROOT\bsps\41100\src\except.s	Provides the low level IRQ and FIQ handlers specific for S3C44B0X Evaluation Board
PSS_ROOT\bsps\41100\src\init.s	Provides the first entry to S3C44B0X Evaluation Board BSP_control flow, and performs the hardware system initialization, including memory mapping, peripherals initialization, stack setup, exception handlers setup, and so on, and finally enters into pSOSystem initialization function without return.
PSS_ROOT\bsps\41100\src\intrhndl.c	Provides the functions for interrupt handler table initialization, interrupt vector installation and a common interrupt handler for both IRQ and FIQ to dispatch control to a particular ISR.
PSS_ROOT\bsps\41100\src\nvram.c	Non-volatile RAM access routines
PSS_ROOT\bsps\41100\src\pic.h	S3C44B0X interrupt controller specific definitions
PSS_ROOT\bsps\41100\src\serial.h	Definitions for both DISI and non-DISI compliant serial drivers
PSS_ROOT\bsps\41100\src\serial.c	Non-DISI compliant serial drivers for S3C44B0X Evaluation Board
PSS_ROOT\bsps\41100\src\disi41100.c	DISI drivers for S3C44B0X Evaluation Board
PSS_ROOT\bsps\41100\src\timer.h	Timer drivers definitions
PSS_ROOT\bsps\41100\src\timer.c	Timer drivers
PSS_ROOT\bsps\41100\src\makefile	S3C44B0X Evaluation Board BSP library makefile
PSS_ROOT\bsps\devices\arm\vector.s	Provides the Samsung ARM device specific exception handlers except for IRQ and FIQ, and exception handler installation function.

MEMORY LAYOUT

The memory map for the S3C44B0X Evaluation Board BSP is shown in the following table:

Memory Type	Address Range	Usage
ROM	0x0000000 0x00ffffff	Bootting ROM area
DRAM	0xc000000 0xc02ffff	Data area for pROBE+ boot ROM.
DRAM	0xc030000 0xc7fefff	Available for downloaded image of pSOSystem and application when using pROBE+ boot ROM.
DRAM	*0xc000000 *0xc7fefff	<i>Available for downloaded image of pSOSystem and application if using the Samsung standard on-board boot ROM rather than the pROBE+ boot ROM.</i>
DRAM	0xc7ff000 0xc7ffeff	Simulated NVRAM area
DRAM	0xc7fff00 0xc7fffff	Exception vector table

When using the Samsung standard on-board boot ROM to download the pSOSystem application into DRAM, application should be built with start execution address 0xc000000 since this boot ROM is going to jump to there after program downloaded. If using pROBE+ boot ROM, you can modify the file **bsps\41100\bsp.mk** to reclaim the memory option so that a part of DRAM area can be used by pROBE+ boot ROM.

DETAILS IN S3C44B0X BSP

1. Hardware Initialization

The hardware initialization is the first step in the pSOSystem startup sequence, and the related code is contained in the files *init.s* and *board.c*. After hardware initialization is complete, the *init.s* will pass control to *sysinit.c*. While the files *init.s* and *board.c* initialize the hardware, the *sysinit.c* file initializes the system software, such as pSOSystem components, drivers and some miscellaneous software functions. The pSOSystem startup sequence is shown below.

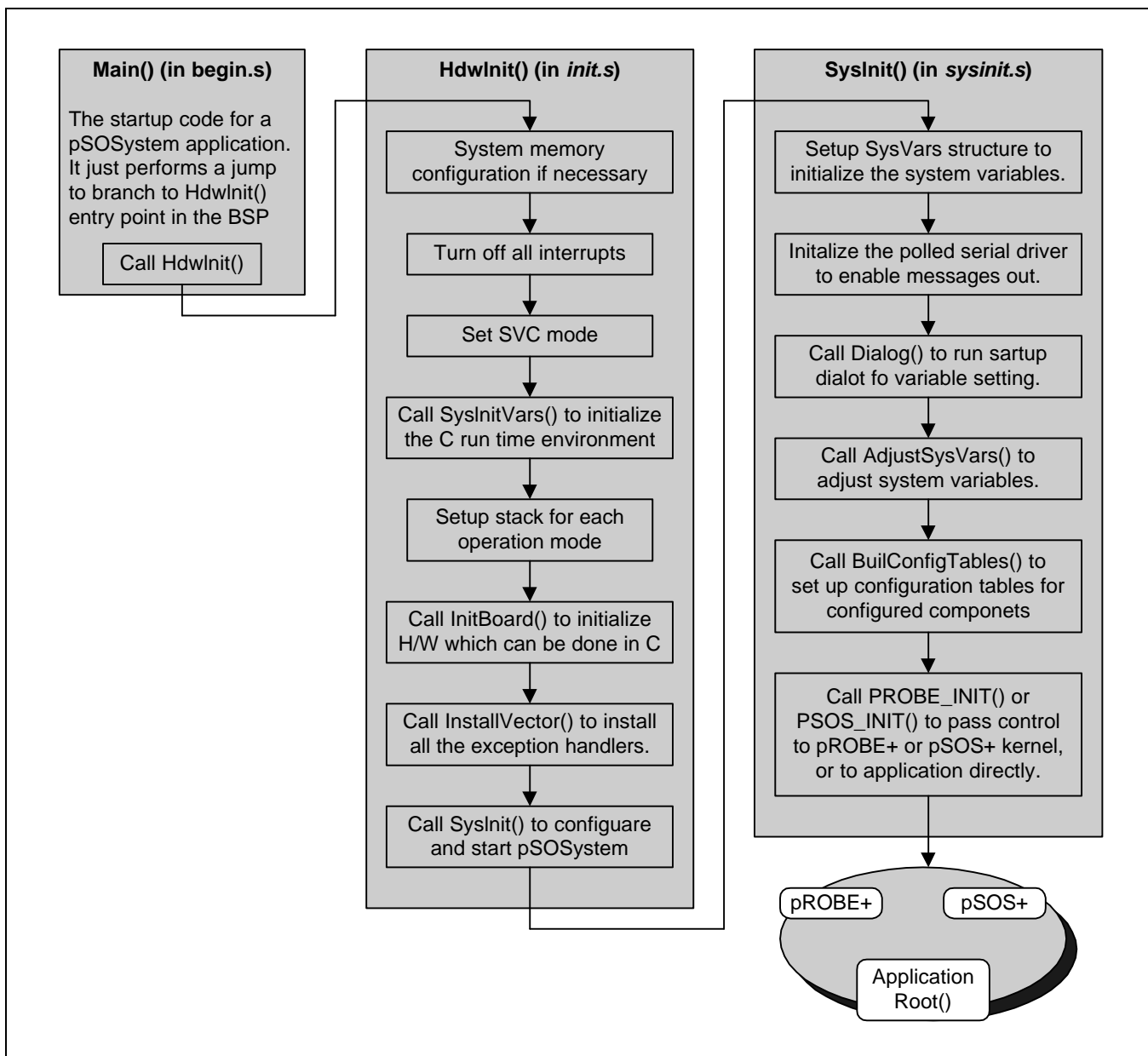


Figure 5-2. pSOSystem Startup Sequence

The main files and functions related to hardware initialization include:

- **init.s**, this file contains routines to initialize the CPU and hardware system.

RomHdwInit	Performs all necessary hardware initialization needed by the board, such as system memory configuration, stack setup for various CPU operating modes, exception handlers installation, and various peripherals initialization. It is used for creating pSOSystem boot ROM.
-------------------	--

HdwInit	Performs necessary hardware initialization needed for creating RAM image.
----------------	---

- **board.c**, this file contains some routines used to initialize the board specific hardware and provide hardware specific information.

InitBoard	Initializes the board specific hardware. It is called from the <i>init.s</i> to perform board initialization that can be done in C.
------------------	---

RamSize	Returns the size of the on-board RAM in bytes. This function is used by any function that needs to know how much contiguous RAM supported on board. It is call by system software configuration files such as <i>sysinit.c</i> , <i>psoscfg.c</i> and so on.
----------------	--

BspRamBase	Returns the base address of a contiguous block of RAM which can be used by pSOSystem.
-------------------	---

BspCpuType	Returns the type of ARM CPU in use on the board.
-------------------	--

SysInitFail	Reports a system initialization failure.
--------------------	--

- **nvarm.c**, this file contains routines to access the simulated NVRAM area on S3C44B0X Evaluation Board. It is used for system configuration parameter storing.

StorageRead	Read data from the simulated NVRAM area.
--------------------	--

StorageWrite	Write data to the simulated NVRAM area.
---------------------	---

2. Exception Handlers and Interrupt Process Mechanism

For the eight exceptions of ARM CPU, the corresponding exception handler routines are included in two source files. One is the file **except.s** which is used to handle the interrupt exceptions and contains the IRQ and FIQ handler routines, and another is the **vectors.s** which locates in the directory **bsp\devices\arm** and contains the functions to set up and maintain exception vector table, the routines to allow dynamic assignment of exception handler vectors, and the handler routines for other six kinds of exceptions.

The interrupt process mechanism implemented in S3C44B0X Evaluation Board BSP is illustrated below.

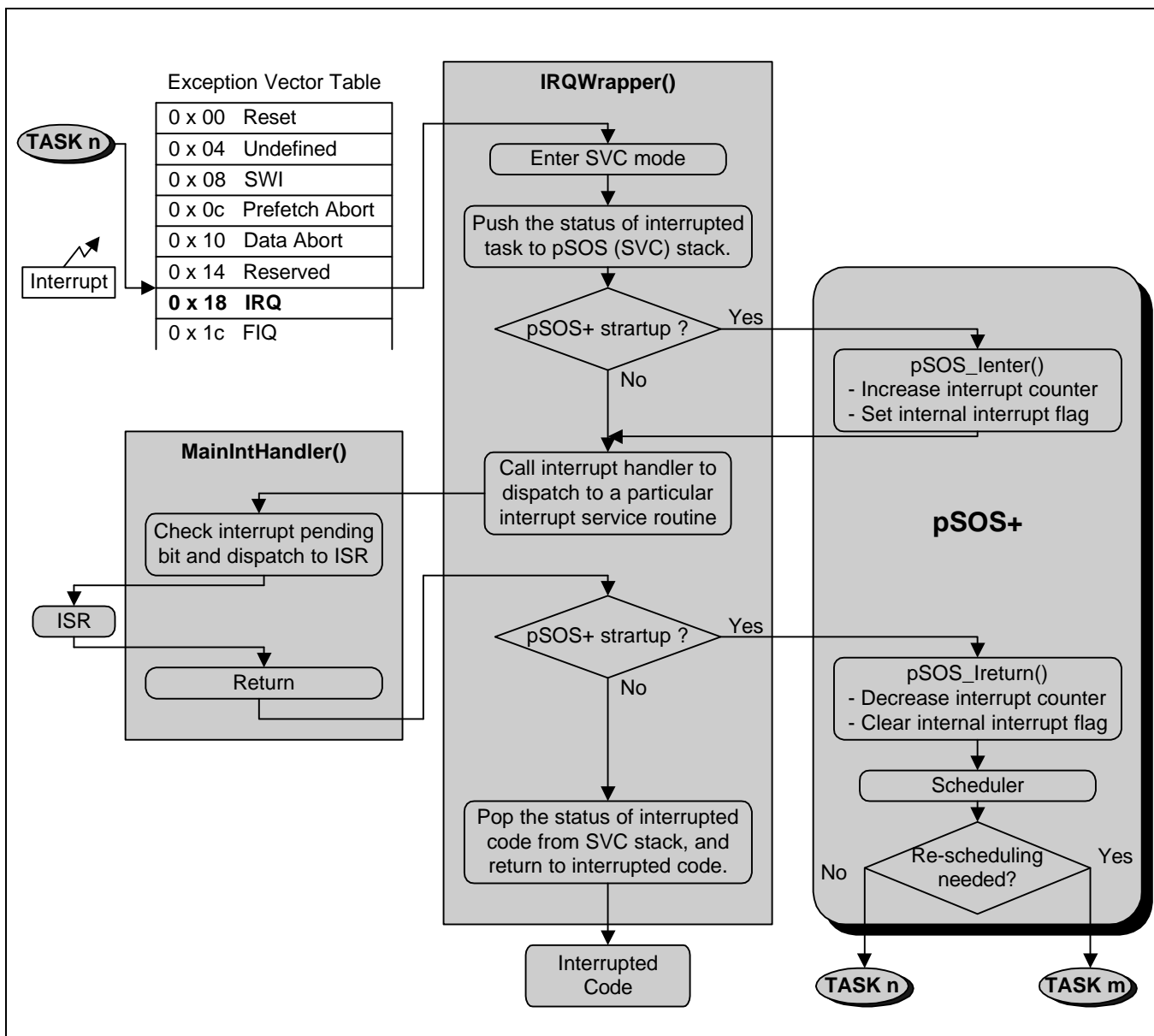


Figure 5-3. S3C44B0X Evaluation Board BSP

The main files and functions related to exception process include:

- **except.s**, this file contains code for IRQ and FIQ handling.

IRQWrapper Handles dispatching of IRQ requests.

FIQWrapper Handles dispatching of FIQ requests.

- **vectors.s**, this file contains codes for initializing and handling six ARM exceptions except for IRQ and FIQ. This includes setting up and maintaining a vector table, wrapper code for exception handlers, and routines to allow dynamic assignment of vectors.

EXCEPT_COMMON Common exception wrapper code.

RESERVEDWrapper RESV exception wrapper code.

ABORTWrapper Data access abort wrapper code.

PREFETCHWrapper Instruction pre-fetch abort wrapper code.

UNDEFWrapper Undefined instruction wrapper code.

SWIWrapper SWI wrapper code.

ROM_FIQ Passes control to FIQ handler installed in soft vector table when FIQ exception occurs. It is only used by boot ROM.

ROM_IRQ Passes control to IRQ handler installed in soft vector table when IRQ exception occurs. It is only used by boot ROM.

ROM_RESERVED Passes control to RESV handler installed in soft vector table when RESV exception occurs. It is only used by boot ROM.

ROM_DATAABORT Passes control to data abort handler installed in soft vector table when data access abort occurs. It is only used by boot ROM.

ROM_PREFETCH Passes control to pre-fetch handler installed in soft vector table when instruction pre-fetch abort occurs. It is only used by boot ROM.

ROM_UNDEFINED Passes control to undefined handler installed in soft vector table when undefined instruction appears. It is only used by boot ROM.

ROM_SWI Passes control to SWI handler installed in soft vector table when SWI exception occurs. It is only used by boot ROM.

InstallVector Installs a default exception handler into soft vector table.

- **intrhndl.c**, this file contains routines for interrupt handling.

InitHandlerTable Initializes a table of pointers to interrupt handlers (ISRs).

SysSetInterrupt Sets an entry in the interrupt handler table to point to a particular handler.

MainIntHandler Dispatches control to correct interrupt handler for a particular interrupt.

3. Hardware-Specific Device Drivers

Two lower-level device drivers are implemented in S3C44B0X BSP, i.e. the tick timer driver and serial port driver.

The pSOSystem tick timer is derived by Timer 3 in S3C44B0X chip. The code to program the timer is in **bsps\41100\src\timer.c**. You may use other timers in S3C44B0X for your own use.

pSOSystem can use both serial ports on the S3C44B0X Evaluation Board. The serial port 1 is mapped to pSOSystem channel 1 and the serial port 2 to pSOSystem channel 2. Two kinds of serial drivers are provided in S3C44B0X Evaluation Board BSP. The source code for DISI driver is found in **bsps\41100\src\disi41100.c**, and the non-DISI compliant serial driver in **bsps\41100\src\serial.c**.

Tick Timer

Tick timer is used by pSOS+ kernel. It generates clock tick and announces passage of time to the pSOS+ kernel for task schedule and management. On S3C44B0X Evaluation Board, we use the S3C44B0X on-chip timer (Timer 3) as the system tick timer. The timer control functions are contained in the file **timer.c**.

In the file **timer.c**, two main functions are provided:

Rtclnit	Initialize the tick timer controller to provide periodic interrupts.
Delay100ms	Delay execution for approximately 100ms. This function is called only during system startup time, and used by startup dialog code to give user a chance to do serial input (for example, to change some system configuration parameters). Here, we use a simple delay routine to simulate this function.

BSP Serial Driver

The BSP serial driver provides two lower-level serial interfaces for upper-level serial driver:

1. led serial interface:

It is used by upper-level hardware-independent serial driver (**pollio.c**) to print output and prompt for input during system booting, or used by pROBE+ debugger console driver for debugging message display or command input. The **pollio.c** module contains two functions, **Prompt** and **Print**, which are used in place of two standard ANSI functions, **scanf** and **printf**, during system startup, that is, at the time before pREPC+ is initialized.

2. Inrrupt-driven serial interface:

It is used by upper-level hardware-independent serial driver (**console.c** or **diti.c**) for use with pSOS+ and pREPC+. The interrupt-driven interface provides a high-efficiency serial channel access for application program.

Two kinds of serial drivers are implemented in S3C44B0X Evaluation Board BSP, that is, the non-DISI compliant serial driver and DISI driver, in which DISI is a new protocol used by pSOSystem Terminal, SLIP, PPP and pROBE+ upper level drivers to interface with the chip dependent lower level driver. You can use either of them for the pSOSystem application, but not in simultaneous.

To determine which serial driver to be included in the BSP, you should define `BSP_NEW_SERIAL` parameter in ***bsps\41100\bsp.h*** file, and select related driver objects in ***bsps\41100\src\makefile*** file.

For example, if you intend to use the DISI driver for serial channels, you should define `BSP_NEW_SERIAL` as YES in *bsp.h* file, and include DISI protocol related objects in *makefile* file, as below:

```
<< bsp\41100\bsp.h >>
```

```
¼ ¼ ¼
```

```
/*=====*/
/*      SERIAL CHANNELS      */
/*=====*/
```

```
#define BSP_SERIAL          2
#define BSP_NEW_SERIAL      YES
#define BSP_SERIAL_MINBAUD  300
#define BSP_SERIAL_MAXBAUD 115200
```

```
¼ ¼ ¼
```

```
<< bsp\41100\src\makefile >>
```

```
¼ ¼ ¼
```

```
##-----*
## Modules compiled into this BSP
##-----*
```

```
SRC_OBJ1 = $(OBJ_DIR)init.o      $(OBJ_DIR)board.o  $(OBJ_DIR)intrhdl.o
SRC_OBJ2 = $(OBJ_DIR)except.o    $(OBJ_DIR)vectors.o $(OBJ_DIR)cpu.o
SRC_OBJ3 = $(OBJ_DIR)nvram.o     $(OBJ_DIR)bspcfg.o
SRC_OBJ4 = $(OBJ_DIR)drv_cutl.o  $(OBJ_DIR)timer.o
#SRC_OBJ5 = $(OBJ_DIR)console.o  $(OBJ_DIR)serial.o
SRC_OBJ5 = $(OBJ_DIR)disi41100.o $(OBJ_DIR)dipi.o $(OBJ_DIR)diti.o $(OBJ_DIR)gsblk.o
```

```
¼ ¼ ¼
```

Otherwise, you should define `BSP_NEW_SERIAL` as NO in *bsp.h* file, and include objects *console.o* and *serial.o* in *makefile* file.

The non-DISI compliant serial driver interface is shown below.

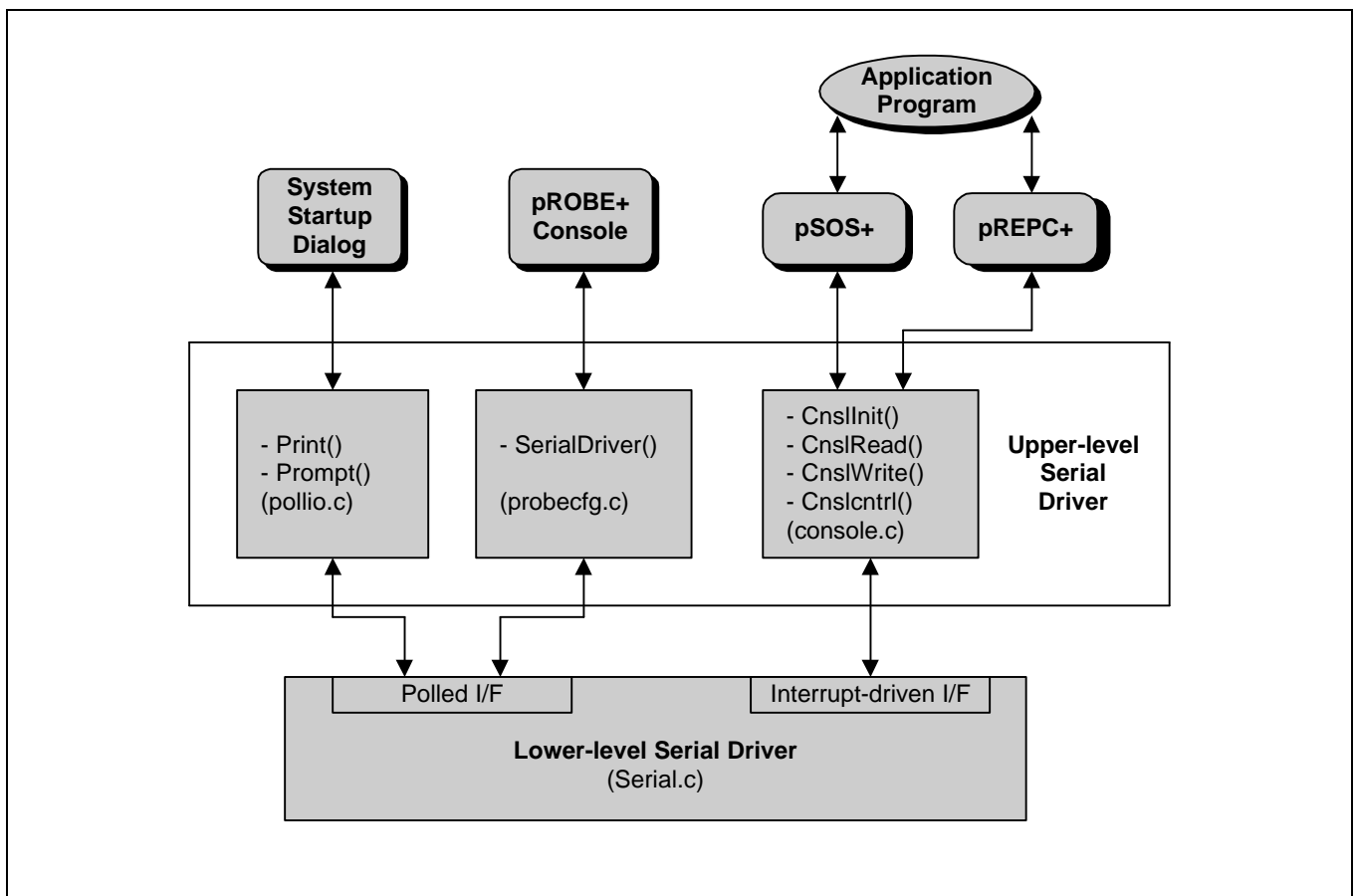


Figure 5-4. Non-DISI Compliant Serial Driver Interface

All lower-level serial routines for non-DISI compliant serial driver interface are contained in the file **serial.c**, which include:

— Startup initialization routine for all serial channels :

SerialSetup Called during startup before any other serial driver calls.

— Polled serial interface routines for pROBE+ debugger console and system startup dialog:

SerialPollInit Initialize the polled serial channels.

SerialPollConsts Check the status of the pROBE+ console channel.

SerialPollConin Get a character from the pROBE+ console.

SerialPollConout Send a character to the pROBE+ console.

— Polled serial interface routines for pROBE+ debugger communication with host:

SerialPollHststs Check the status of the pROBE+ host channel.

SerialPollHstin Get a character from the pROBE+ host channel.

SerialPollHstout Send a character to the pROBE+ host channel.

— Serial channel operation-mode switching routines for the case of application program and pROBE+ sharing one serial channel:

SerialPollOn Called by pROBE+ to turn off the interrupt enables when taking control from the application.

SerialPollOff Called by pROBE+ to restore the interrupt status when relinquishing control to application.

— Interrupt-driven serial interface routines for **console.c**:

SerialIntBaud Change the baud rate of interrupt-driven serial channel.

SerialIntInit Initialize the interrupt-driven serial channel.

SerialIntRxioff Turn off receive interrupt.

SerialIntRxion Turn on receive interrupt.

SerialIntTxioff Turn off transmit interrupt.

SerialIntTxion Turn on transmit interrupt.

SerialIntRead Read a character from serial channel.

SerialIntWrite Write a character to serial channel.

The DISI interface is shown below.

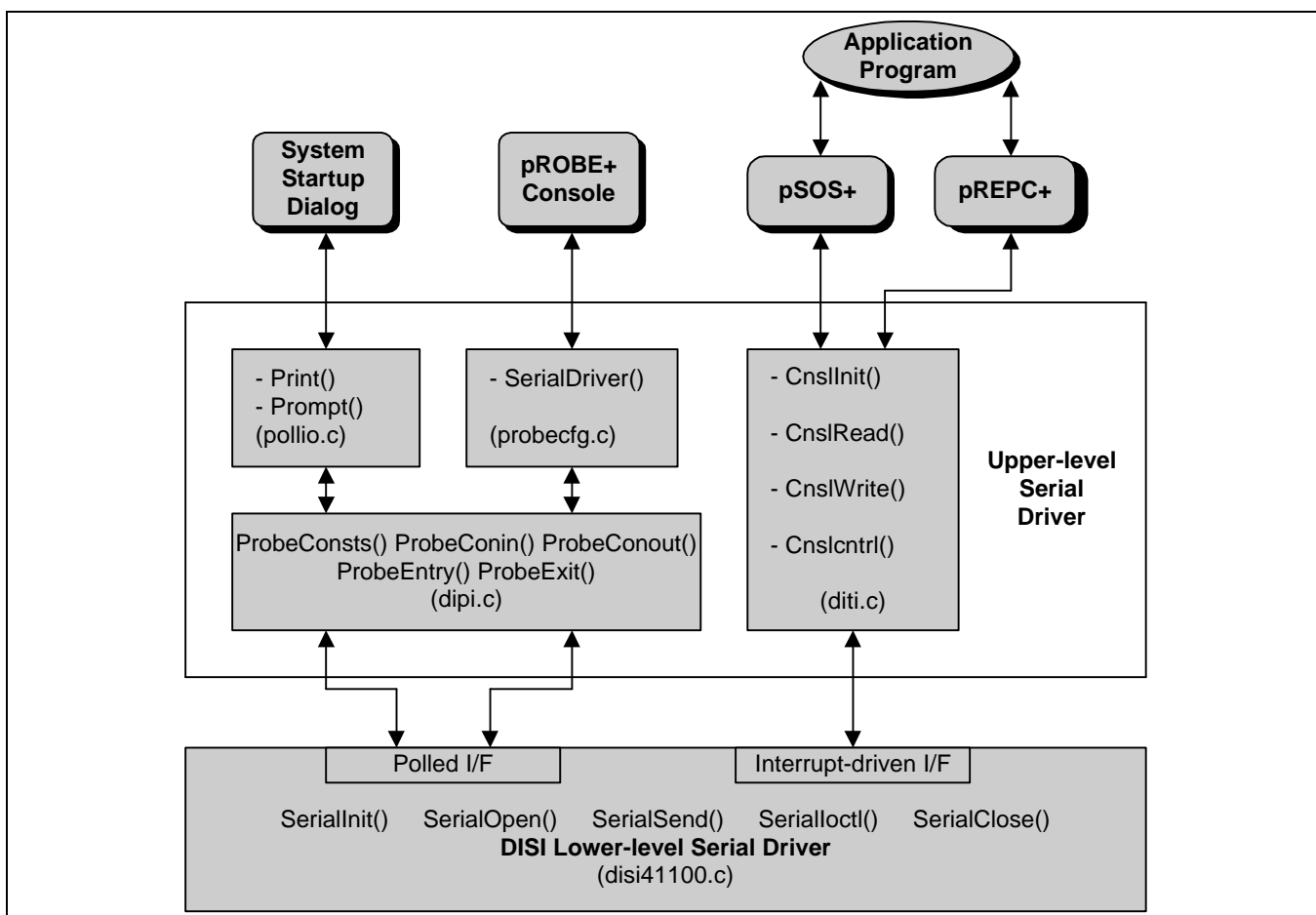


Figure 5-5. DISI Interface

The device-dependent lower-level serial routines for DISI interface are contained in the file **disi41100.c**, which include:

SerialInit	Initialize the driver.
SerialOpen	Open a channel.
SerialSend	Send data on the channel.
Serialloctl	Perform a control operation on the channel.
SerialClose	Close the channel.

HARDWARE JUMPER SETTINGS

Switch and jumper settings on the S3C44B0X Evaluation board are described in additional documentation. There are no extra requirements of pSOSystem other than removing the Jumper (JP1) when using the Embedded ICE for debugging.

BUILD THE BSP LIBRARY

The S3C44B0X Evaluation Board BSP library is supplied with the file **bsps\41100\libbsp.32b**. If necessary to rebuild the BSP library, two methods can be used. One is to create a BSP library generation project by the Project Manager in ARM SDT, which needs to contain all the BSP source files and header files for S3C44B0X Evaluation Board, and build the library in the ARM SDT environment.

Alternatively, you can use the **psosmake** utility to build it. To do it, you should modify the 'envarm.bat' to reflect the BSP as '41100' and 'BSP_TYPE' as '32b' first, and then type the command lines in DOS window as below.

```
— cd %PSS_ROOT%\bsps\41100\src
— psosmake clean
— psosmake
```

The **psosmake** command automatically picks up the **makefile** in current directory to complete the required BSP library building operations.

APPLICATION EXAMPLES

Several application examples are provided along with S3C44B0X BSP to demonstrate the use of pSOSystem and its components. They are contained in the applications directory, **%PSS_ROOT%\apps**. Each application is located in its own sub-directory, and contains a **makefile** to be used to build the application and causes it to be linked with the correct libraries.

The following table summarizes these examples.

Sub-directory	Description
hello	Simple one-task application that displays the message "Hello, world." It is a starting point to get an application up and running on target board.
proberom	This application can be used to build pSOSystem Boot ROM for non-Ethernet systems. It is also an example of how to use a system startup dialog.
demo	A multitasking application in which several tasks compete for various resources.

Each sample application includes a **sys_conf.h** configuration file which has been properly set to match S3C44B0X Evaluation Board. You can also use these settings as a reference for other applications.

HELLO

The Hello sample is a simple program that displays a message. The application consists of a single task named ROOT that prints out a short message to the target's serial port and then suspends itself.

The output for this application is the familiar:

Hello, world

The root.c file contains the macro OUTPUT_TO_DEBUGGER. Edit the root.c file to set this macro to 0 in order to view output on the serial port (pROBE console). Setting it to 1 will cause the program to send the output message through the pROBE+ debugger to the standard I/O window of your pRISM+ source-level debugger.

PROBEROM

The proberom application is used to build pSOSystem Boot ROM for S3C44B0X Evaluation Board. This is only sample application that does not require the pSOS+ kernel. The only component included is the pROBE+ debugger. When used to boot the system, this code initializes the hardware and executes the pROBE+ debugger, or allows a remote connection via the console port to a remote debugger.

A system startup dialog is also enabled in this application, which gives out some system configuration information during system initialization and allows you to modify the configuration parameters to re-configure system before any application runs.

This application is to be described in the section *Build A pSOSystem Boot ROM* later.

DEMO APPLICATION

The demo application consists of a number of tasks which execute under the control of pSOS+ scheduler. In addition to multitasking execution, task communication and synchronization are also demonstrated in this example.

The demo program starts from the ROOT task. The ROOT creates six tasks, named as Task1 ~ Task6, and other system objects needed, such as semaphore (TSEM) and message queue (TQUE). After the objects are created, the ROOT suspends itself. The remaining tasks will then run.

```
void root(void)
{
    void *dummy;
    U32 args[4]={0};
    U32 ioretval, iopb[4];

    /*-----*/
    /* Initial UART driver */
    /*-----*/
    de_init((DEV_SERIAL), iopb, &ioretval, &dummy);
    UART_out("\n\rHello, welcome to use S3C44B0X Demo Board !\n\r");

    /*-----*/
    /* Generate task, message queue and semaphore */
    /*-----*/

    t_create("TSK1", 120, 1024, 1024, 0, &task_id1);
    t_create("TSK2", 105, 1024, 1024, 0, &task_id2);
    t_create("TSK3", 105, 1024, 1024, 0, &task_id3);
    t_create("TSK4", 110, 1024, 1024, 0, &task_id4);
    t_create("TSK5", 110, 1024, 1024, 0, &task_id5);
    t_create("TSK6", 107, 1024, 1024, 0, &task_id6);

    sm_create("TSEM", 1, SM_FIFO, &sm_id);
    q_create("TQUE", 20, Q_FIFO|Q_LIMIT, &queue_id);

    /*-----*/
    /* Start tasks */
    /*-----*/
    t_start(task_id1, T_NOPREEMPT|T_TSLICE|T_USER, Task1, args);

    t_start(task_id2, T_PREEMPT|T_TSLICE|T_USER, Task2, args);
    t_start(task_id3, T_PREEMPT|T_TSLICE|T_USER, Task3, args);

    t_start(task_id4, T_PREEMPT|T_TSLICE|T_USER, Task4, args);
    t_start(task_id5, T_PREEMPT|T_TSLICE|T_USER, Task5, args);

    t_start(task_id6, T_NOPREEMPT|T_TSLICE|T_USER, Task6, args);

    t_suspend(0L);
}
```

Task1 is the first task to execute among the six created tasks, because it is the highest priority task in them (priority 120). Task4 executes after task 1 suspends (priority 110), and the other tasks execute in the order of Task5 (priority 110), Task6 (priority 107), Task2 (priority 105) and Task3 (priority 105) depending on their priorities. The reason for why Task4 executes before Task5 even though they both have the same priority is that Task4 was created and started first in ROOT.

Like all of the tasks in this example, Task1 does some preliminary initialization and then starts execution of an endless loop. Processing inside of Task1 endless loop includes calling a sleep function and sending event to Task6. Because of `tm_wkafter(1000)` call, Task1's loop is executed once every 1000 timer ticks. Although Task6 is made ready on each event sending, it does not execute until Task1 executes sleep call again because the Task6 has a lower priority than Task1.

```
void Task1()
{
    U32 status = 0;
    char ch;

    UART_out("\n\rTask 1 starts ..... \n\r");

    Task_Time = 0;

    while(1) {

        // Sleep 1000 timer ticks
        tm_wkafter(1000);

        // Send an event to task 6
        status = ev_send(task_id6, 1);

        if(status == SUCCESS)
            Task_Time++;
        else
            UART_out("\n\r Fail to send event in task 1 !\n\r");

    }
}
```

Task2 continually sends a message to queue. If Task3 is already waiting at the queue, the message is passed to it, and the Task3 is then unblocked and made ready to run.

In another side, Task3 requests message from queue. When the queue becomes empty, Task3 is blocked, until the next message comes.

```
void Task2()
{
    U32 status = 0;
    U32 msg_s[4];

    UART_out("\n\rTask 2 starts ..... \n\r");

    Task_2_messages_sent = 0;
    msg_s[0] = Task_2_messages_sent;

    while(1) {

        // Send message to queue, which task 3 reads from
        status = q_send(queue_id, msg_s);

        if(status == SUCCESS)
            msg_s[0] = ++Task_2_messages_sent;
        else if (status != ERR_QFULL)
            UART_out("\n\r Fail to send message in task 2 !\n\r");
    }
}

void Task3()
{
    U32 status = 0;
    U32 msg_r[4];

    UART_out("\n\rTask 3 starts ..... \n\r");

    Task_3_messages_received = 0;

    while(1) {

        // Receive message from queue, which task 2 writes to
        status = q_receive(queue_id, Q_WAIT, 0, msg_r);

        if(status == SUCCESS)
            Task_3_messages_received++;
        else
            UART_out("\n\r Fail to send message in task 3!\n\r");
    }
}
```

Task4 and Task5 execute the similar loop. Both of them compete for an identical semaphore. Once the semaphore is obtained by one task, this task sleeps for 40 timer ticks before releasing the semaphore again. This action make the another task attempting to obtain the same semaphore to be blocked, until the semaphore is released.

```
void Task4()
{
    U32 status = 0;

    UART_out("\n\rTask 4 starts ..... \n\r");

    Task_4_semaphore_obtained = 0;

    while(1) {
        // Acquire a semaphore token
        status = sm_p(sm_id, SM_WAIT, 0);

        if(status == SUCCESS) {
            Task_4_semaphore_obtained++;
            tm_wkafter(40);
            sm_v(sm_id);
        }
    }
}

void Task5()
{
    U32 status = 0;

    UART_out("\n\rTask 5 starts ..... \n\r");

    Task_5_semaphore_obtained = 0;

    while(1) {
        // Acquire a semaphore token
        status = sm_p(sm_id, SM_WAIT, 0);

        if(status == SUCCESS) {
            Task_5_semaphore_obtained++;
            tm_wkafter(40);
            sm_v(sm_id);
        }
    }
}
```


Task6 executes a loop waiting for at least one event flag to be set. The event flag is set by Task1 as mentioned before. The Task6 executes at the same frequency as Task1, and gives a status report on program execution each time when it executes.

```
void Task6()
{
    U32 status = 0, events_r;

    UART_out("\n\rTask 6 starts ..... \n\r");

    Task_6_event_received = 0;

    while(1) {

        // Send an event to task 6
        status = ev_receive((U32)0xffffffff, EV_WAIT|EV_ANY, 0, &events_r);

        if(status == SUCCESS)
        {
            Task_6_event_received++;

            // Report status
            printf("\n\r[Status Report %d]", Task_6_event_received);
            printf("\n\r Task 1 sent events: %d", Task_Time);
            printf("\n\r Task 2 sent messages: %d", Task_2_messages_sent);
            printf("\n\r Task 3 received messages: %d", Task_3_messages_received);
            printf("\n\r Task 4 obtained semaphores: %d", Task_4_semaphore_obtained);
            printf("\n\r Task 5 obtained semaphores: %d", Task_5_semaphore_obtained);
            printf("\n\r Task 6 received events: %d\n\r", Task_6_event_received);

        } else
            UART_out("\n\r Fail to receive event in task 6 !\n\r");

    }
}
```

The status report gives message similar to that shown below. The information includes that how many events and messages have been sent/received, how many semaphores have been obtained by Task4 and Task5 respectively, and so on.

[Status Report 1]**Task 1 sent events: 1****Task 2 sent messages: 188****Task 3 received messages: 167****Task 4 obtained semaphores: 13****Task 5 obtained semaphores: 13****Task 6 received events: 1**

BUILD A pSOSystem BOOT ROM

A pre-built pSOSystem boot ROM image for the S3C44B0X Evaluation Board, **proberom.bin**, is supplied along with the BSP, which can be found in directory **apps\proberom**. It is built using a modified version of the standard code in proberom application. This boot ROM image only includes the pROBE+ component, without the pSOS+ kernel. When it is executed from ROM, it initializes the platform and runs the pROBE+ debugger in stand-alone mode or allows for connection from a remote program such as the pRISM+ Manager or a source-level debugger via a serial port.

The ROM image includes dialog code, and is set to boot into pROBE+ and wait for the host debugger via a serial connection when it starts up. The related settings can be found in the **sys_conf.h** file.

```

/*****
/*  BASIC PARAMETERS                                */
*****/

#define SC_SD_PARAMETERS      STORAGE
#define SC_STARTUP_DIALOG    YES      // Enable the dialog code
#define SC_BOOT_ROM          YES      // Build for Boot ROM
#define SD_STARTUP_DELAY     20
#define SE_DEBUG_MODE        DBG_XS   // Boot into pROBE+ and wait for the
                                     // host debugger via a serial connection

¼ ¼ ¼

/*****
/*  SERIAL CHANNEL CONFIGURATION                    */
*****/

#define SD_DEF_BAUD          115200
#define SC_APP_CONSOLE       2
#define SC_PROBE_CONSOLE     1
#define SC_RBUG_PORT         1

¼ ¼ ¼

```

You can re-build the pSOSytem boot ROM image to be your own if necessary. To do it, you can modify code in **apps\proberom** directory as you want, and use **psosmake** utility to build your own boot ROM image in a DOS window as below.

- cd %PSS_ROOT%\apps\proberom
- psosmake clean
- psosmake rom.bin

The file **rom.bin** can then be programmed into ROM devices and inserted in the proper sockets on the board.

For more information, please refer to *pSOSystem Getting Started* and *pSOSystem Advanced Topics*.

RUN DEMO ON S3C44B0X EVALUATION BOARD

The demo program can be downloaded and run on the S3C44B0X evaluation board in two ways. One is that using the on-board Boot ROM shipped with the board to download the demo image and run it; another way is to use the pROBE Boot ROM to do it. Though no difference to execute the demo program in standalone mode, the two Boot ROMs accept different format of image when downloading, which to be described later.

SETUP DEMO ENVIRONMENT

Before downloading and running the demo program, you should connect the Evaluation Board with the host PC via an RS-232 connection. By default, the on-board boot ROM and pROBE+ boot ROM use the serial port1 (P1) to give system startup message. You can use the same channel to download the demo program image.

Use a serial cable to connect the on-board serial port1 to one of PC COM ports, and run an ASCII terminal emulation program, for example the Hyper Terminal, on the host. Set the terminal characteristics to the following:

- 115200 baud
- 8-bit data
- stop bit
- no parity

Then, activate a MS-DOS window to prepare for image building.

DOWNLOAD DEMO APPLICATION WITH ON-BOARD BOOT ROM

With SAMSUNG standard on-board boot ROM (without pROBE+), you can download the demo program via on-board serial port 1.

1. Build the Image

The on-board ROM receives the download image and allocates it in DRAM from the address 0xc000000, and after download is over ROM program automatically jump to there to start executing the downloaded image. Therefore, you need to build your RAM image at address 0xc000000. By default, we kept this address in **bsps41100bsp.mk** file for RAM image building like below:

```
#-----*
# Common rules for this BSP
#-----*

#-----
# if using the on-board system boot rom
#-----
RAMOPTS    = -Base 0xc000000

#-----
# if using the pSOSystem boot rom
#-----
#RAMOPTS    = -Base 0xc030000

¼ ¼ ¼
```

The on-board boot ROM accepts the AXF format image. You should create an RAM image in AXF format for download use as below:

```
— cd %PSS_ROOT%\apps\demo
— psosmake clean
— psosmake ram.axf
```

2. Start the Boot ROM

Power up the Evaluation Board, the Boot ROM performs the system initialization and memory test, and the terminal displays the system startup message similar to that shown below.

```
44BMON Ver 0.01 for S3C44B0X May, 2000
COM:115.2kbps,8Bit,NP,UART0 <n+6>(4)+(n)+CS(2)
DNADDR:c000000 ISR_ADDR:c7fff00 SYSCFG:e
E-mail:kwark@sec.samsung.com

Memory Test(c000000h-c7f0000h):O.K.
```

3. Download the Image

After memory test is complete, you can give the following command in DOS window to download the image:

```
— wkcom2 ram.axf /1 /d:1
```

'wkcom2' is a tool which writes to the serial port of the PC. You may need to give proper PC COM port number setting when using this command. Wait till the file gets downloaded, after downloading is over the boot program jumps to the location 0xc000000 and runs the demo program.

When demo program starts running, it passes the control to pROBE debugger first because the pROBE+ component is included in the demo image, and gives a message followed by a pROBE prompt as following:

```
Now, Downloading... [FILESIZE:1103762(1103762)]

Download O.K.

pROBE+/ARM/BE PS V3.1.0
pROBE+/ARM/BE CE V3.1.0
pROBE+/ARM/BE RD V3.1.0
pROBE+/ARM/BE QS V3.1.0
pROBE+/ARM/BE DI V3.1.0

pROBE+>
```

At this point the Demo sample application can be executed.

DOWNLOAD DEMO APPLICATION WITH PROBE BOOT ROM

You can also download and run the demo program with the pSOSystem boot ROM. To do it, you should burn the pROBE+ Boot ROM image (the **proberom.bin** we supplied) into ROM devices and use the new ROMs to replace the old ones in the sockets U8 and U9 on the board.

1. Build the Image

When using the pROBE Boot ROM, the DRAM area from 0xc000000 to 0xc030000 is reserved for data use of Boot ROM. The download image should be allocated in DRAM after the address 0xc030000. Therefore, you need to modify the base address of RAM image in **bsps\41100\bsp.mk** file for RAM image building. We have supplied an RAM Base option for this case in the **bsp.mk** file. You only need to remark the RAMOPTS for on-board Boot ROM, and validate the RAMOPTS for pSOSystem Boot ROM as below:

```
#-----*
# Common rules for this BSP
#-----*

#-----
# if using the on-board system boot rom
#-----
#RAMOPTS    = -Base 0xc000000

#-----
# if using the pSOSystem boot rom
#-----
#RAMOPTS    = -Base 0xc030000

¼ ¼ ¼
```

Differ from the original on-board Boot ROM, the pROBE+ Boot ROM only accepts the standard Motorola S-record file, that is the HEX format, in standalone mode. You should create an RAM image in HEX format for it.

- cd %PSS_ROOT%\apps\demo
- psosmake clean
- psosmake ram.hex

2. Start the pROBE ROM

When you power up or reset the board, the terminal displays a pSOSystem startup message as below:

```
pSOSystem V2.2.3 for ARM/BE
Copyright (c) 1991 - 1998, Integrated Systems, Inc.
-----
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a serial connection
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 115200
  After board is reset, start-up code will wait 20 seconds
-----
To change any of this, press any key within 20 seconds
```

The message above shows the pSOSystem Boot ROM program configuration parameters. After this message appears, the Boot ROM program waits about 20 seconds to allow you to change the configuration parameters.

The pROBE+ debugger can operate in either *standalone* or *remote* mode. In standalone mode, the pROBE+ accepts and responds to user commands through a console port, which is normally an RS-232 line connected to an ASCII terminal or a terminal emulator (here, we use the HyperTerminal running in PC as a terminal emulator). In this mode, it functions as an enhanced assembly-level debugger. Commands, similar to those provided by other firmware monitors, are provided to download image, display/modify processor registers and memory, set breakpoints and control system execution.

In the remote mode, the pROBE+ debugger performs the low-level operations necessary to support a source-level debugger running on host PC and communicates with the source level debugger over an RS-232 connection.

By default, we set the Boot ROM pROBE+ debugger as remote mode when it starts up. If you don't press any key, the ROM pROBE+ debugger will enter remote mode directly after 20 seconds and give message like:

```
Updating non-volatile storage. This may take a while...Done
pROBE+ is now ready to talk to the host debugger over this serial channel...
```

Once debugger enters the remote debugging state, it waits for communication with the source level debugger running on host PC, and does not accept commands from the HyperTerminal any more.

However, you can change it to standalone mode by modifying the configuration. To do it, press any key within 20 seconds after the startup message appears, and do changes to enter standalone mode according to the prompt information as follows.

M)odify any of this or (C)ontinue? [M]

For each of the following questions, you can press <Return> to select the value shown in braces, or you can enter a new value.

How should the board boot?

1. pROBE+ stand-alone mode
2. pROBE+ waiting for host debugger via serial connection

Which one do you want? [2] 1

HARDWARE PARAMETERS:

Baud rate for serial channels [115200]

How long (in seconds) should CPU delay before starting up? [20]

START-UP MODE:

Boot into pROBE+ stand-alone mode

HARDWARE PARAMETERS:

Serial channels will use a baud rate of 115200

After board is reset, start-up code will wait 20 seconds

(M)odify any of this or (C)ontinue? [M] c

Updating non-volatile storage. This may take a while...Done

pROBE+/ARM/BE PS V3.1.0

pROBE+/ARM/BE CE V3.1.0

pROBE+/ARM/BE RD V3.1.0

pROBE+/ARM/BE QS V3.1.0

pROBE+/ARM/BE DI V3.1.0

pROBE+>

After the pROBE+ prompt appears as shown above, you can use the ROM pROBE+ debugger to download the demo program image.

3. Download the Image

To download the executable demo image, first type in the pROBE+ **dl** (download) command in HyperTerminal. The pROBE+ then waits for you to download **ram.hex**.

So give the following command in DOS window to download the image:

```
— wkcom2 ram.hex /1 /g /d:1
```

Note that you may need to give proper PC COM port number setting depending on your serial connection. In the above command, we assume that you are using PC COM1 for image downloading. After downloading is over, the debugger gives a message to show the total records it received, and a prompt again. The debugger does not pass control to the downloaded program automatically. To start the executable image, you should input the pROBE+ **go** command with the image start address which is defined in the **bsp.mk** file as mentioned before. This process is shown below:

```
pROBE+> dl

12837 records read

pROBE+>

pROBE+> go c030000
```

After downloaded image starts execution, a message similar to the following appears:

```
pROBE+/ARM/BE PS V3.1.0
pROBE+/ARM/BE CE V3.1.0
pROBE+/ARM/BE RD V3.1.0
pROBE+/ARM/BE QS V3.1.0
pROBE+/ARM/BE DI V3.1.0

pROBE+>
```

At this point, the demo application can run.

Note that although this message may look similar to the one that appeared when ROM debugger entered the standalone mode, the message comes from the downloaded pROBE+ debugger included in RAM image rather than the ROM pROBE+ debugger. Actually, the ROM pROBE+ debugger is no longer executing and can regain control only if you reset the board.

The downloaded pROBE debugger is set to run in standalone mode.

EXECUTE THE DEMO APPLICATION

Once the downloaded RAM pROBE+ debugger gets started (that is, the pROBE+ prompt appears), you can use it to run the demo application and monitor the program execution. Here, we only intend to show you some basic pROBE+ debugging operations on demo application, for more information on the pROBE+ debugger operations, please refer to *pROBE+ User's Guide*.

1. Initialize pSOS+ Kernel

The pROBE+ **gs** (go system) command starts the pSOS+ kernel. It passes control to the pSOS+ startup entry point and sets a *pSOS+ Initialized Break*. This break causes execution to halt immediately prior to execution of the first instruction in the ROOT task. The **gs** command should be carried out before running any application with pSOS+ kernel embedded.

Enter the **gs** command from the pROBE+ prompt, the following message appears.

```
pROBE+> gs

Kernel Event Break                                Running: 'ROOT' -#00020000
-----
pSOS Initialized Event
-----
CPSR = 00000013 (nzcvc if ARM SVC32)  SP =0C7FDBF0 LR =0C03AE60
R0 =00000000 R1 =00000000 R2 =00000000 R3 =00000000 R4 =00000000
R5 =00000000 R6 =00000000 R7 =00000000 R8 =00000000 R9 =0C06A038
R10=00000000 R11=00000000 R12=00000000 USP=0C7FCC00 ULR=00000000
SSP=0C7FDBF0 SLR=0C03AE60 SPSR_svc=40000013
PC =0C031F80-0C031F80: E92D4000  STMDB  sp!,{lr}

pROBE+>
```

The pSOS+ kernel is now initialized, and the execution of the ROOT task is pending.

2. Set Breakpoints

The pROBE+ debugger supports the following types of breakpoints:

- Instruction Breakpoints
- pSOS+ Service Breakpoints
- Dispatch Breakpoints
- Timer Breakpoints
- Memory Access Breakpoints

Here, we set a timer breakpoint which intends to stop the application execution after 500 clock ticks have elapsed.

```
pROBE+> db ti 500
```

```
TIMER_BREAK_____TICKS_____TICKS_LEFT_____
                00000500      00000500
-----
```

```
pROBE+>
```

3. Start or Resume Execution

With the **go** command, you can start execution of demo application and resume execution after the timer break occurs.

```
pROBE+> go
```

```
Hello, welcome to use S3C44B0X Demo Board !
```

```
Task 1 starts .....
```

```
Task 4 starts .....
```

```
Task 5 starts .....
```

```
Task 6 starts .....
```

```
Task 2 starts .....
```

```
Task 3 starts .....
```

```
[Status Report 1]
```

```
Task 1 sent events: 1
```

```
Task 2 sent messages: 188
```

```
Task 3 received messages: 167
```

```
Task 4 obtained semaphores: 13
```

```
Task 5 obtained semaphores: 13
```

```
Task 6 received events: 1
```

Kernel Event Break Running: 'TSK2' -#000E0000

TIMER Event

CPSR = 60000010 (nZCv if ARM User32) SP =0C7F56B0 LR =0C0322EC
R0 =00000035 R1 =00000014 R2 =00000020 R3 =00000000 R4 =0C7F56B8
R5 =00000000 R6 =00000000 R7 =00000000 R8 =00000000 R9 =0C06A06C
R10=00000000 R11=00000000 R12=00000027 USP=0C7F56B0 ULR=0C0322EC
SSP=0C7F5B20
PC =0C062060-0C062060: E8BD8010 LDMIA sp!,{r4,pc}

pROBE+> go

[Status Report 2]

Task 1 sent events: 2
Task 2 sent messages: 398
Task 3 received messages: 377
Task 4 obtained semaphores: 26
Task 5 obtained semaphores: 25
Task 6 received events: 2

Kernel Event Break Running: 'TSK2' -#000E0000

TIMER Event

CPSR = 60000013 (nZCv if ARM SVC32) SP =0C7F5AD0 LR =0C7F5AF0
R0 =00140000 R1 =0C7F56B8 R2 =00000020 R3 =00000000 R4 =0C7F56B8
R5 =00000000 R6 =00000000 R7 =00000000 R8 =00000000 R9 =0C06A06C
R10=00000000 R11=00000000 R12=00000027 USP=0C7F56B0 ULR=0C0322EC
SSP=0C7F5AD0 SLR=0C7F5AF0 SPSR_svc=40000093
PC =0C039FE8-0C039FE8: E89400F0 LDMIA r4,{r4-r7}

pROBE+>

4. Examine Objects

Using the query commands, you can examine user-created objects such as tasks (**qt**), message queues (**qq**), semaphores (**qs**), and other key pSOS+ structures.

pROBE+> qt

Name	TID	Prio	Mode	Status	Susp?	Parameters	Ticks
'IDLE'	-#00010000	00	2000	Ready			
'ROOT'	-#00020000	F0	2000	Ready	YES		
'pMNG'	-#00030000	F7	2001	Evwait		EVENTS = 00000003	forever
'pINP'	-#00040000	F6	2001	Ready	YES		
'pOUT'	-#00050000	F5	2001	Ready	YES		
'pROC'	-#00060000	F4	2001	Ready	YES		
'TSK1'	-#000D0000	78	0003	Wkafter			000001BC
'TSK2'	-#000E0000	69	0002	Running			
'TSK3'	-#000F0000	69	0002	Ready			
'TSK4'	-#00100000	6E	0002	Swait		SM = 'TSEM' -#00130000	forever
'TSK5'	-#00110000	6E	0002	Wkafter			00000006
'TSK6'	-#00120000	6B	0003	Evwait		EVENTS = FFFFFFFF	forever

pROBE+> qq

Name	QID	TQ Len	MQ Len	MQ Limit	Mgb	Qtype	Variable
'RXQ1'	-#00090000	00000000	00000000	none	Sys-pool	FIFO	No
'CNQ1'	-#000A0000	00000000	00000000	none	Sys-pool	FIFO	No
'TQUE'	-#00140000	00000000	00000014	00000014	Sys-pool	FIFO	No

Sys-pool total = 00000064

Sys-pool free = 00000050

pROBE+> qs

Name	SMID	Count	TQ Len	Qtype
'RDA1'	-#00070000	00000001	00000000	FIFO
'WRA1'	-#00080000	00000001	00000000	FIFO
'TXC1'	-#000B0000	00000000	00000000	FIFO
'CTL1'	-#000C0000	00000000	00000000	FIFO

pROBE+>

5. Display Memory and Registers

To display memory and register contents, use **dm** and **dr** commands respectively.

```
pROBE+> dm c7fff00
```

```
0C7FFF00  0C 03 2A A8 0C 03 2A EC  0C 03 9D 60 0C 03 2A D8  ..*...`..*
0C7FFF10  0C 03 2A C0 0C 03 2A A8  0C 03 2C 28 0C 03 2C D0  ..*...*,(.,.
0C7FFF20  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  .....
0C7FFF30  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  .....
```

```
pROBE+> dr
```

```
CPSR = 60000013 (nZCv if ARM SVC32)  SP =0C7F5AD0 LR =0C7F5AF0
R0 =00140000 R1 =0C7F56B8 R2 =00000020 R3 =00000000 R4 =0C7F56B8
R5 =00000000 R6 =00000000 R7 =00000000 R8 =00000000 R9 =0C06A06C
R10=00000000 R11=00000000 R12=00000027 USP=0C7F56B0 ULR=0C0322EC
SSP=0C7F5AD0 SLR=0C7F5AF0 SPSR_svc=40000093
PC =0C039FE8-0C039FE8: E89400F0  LDMIA  r4,{r4-r7}
```

```
pROBE+>
```

6. Restart Program

To restart the demo program without downloading again, enter the **gs** command.

```
pROBE+> gs
```

```
Kernel Event Break                                Running: 'ROOT' -#00020000
```

```
-----
pSOS Initialized Event
-----
```

```
CPSR = 00000013 (nzcvc if ARM SVC32)  SP =0C7FDBF0 LR =0C03AE60
R0 =00000000 R1 =00000000 R2 =00000000 R3 =00000000 R4 =00000000
R5 =00000000 R6 =00000000 R7 =00000000 R8 =00000000 R9 =0C06A038
R10=00000000 R11=00000000 R12=00000000 USP=0C7FCC00 ULR=00000000
SSP=0C7FDBF0 SLR=0C03AE60 SPSR_svc=40000013
PC =0C031F80-0C031F80: E92D4000  STMDB  sp!,{lr}
```

```
pROBE+>
```

You can then repeat the previous operations as you want.