S5N8947

ADSL/CABLE MODEM

32-BIT RISC

MICROCONTROLLER

PROGRAMMER'S GUIDE

Revision 1.0

SAMSUNG ELECTRONICS

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. Samsung assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained herein.

Samsung reserves the right to make changes in its products or product specifications with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

This publication does not convey to a purchaser of semiconductor devices described herein any license under the patent rights of Samsung or others.

Samsung makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Samsung assume any liability arising out of the application or use of any product or circuit and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

"Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by the customer's technical experts.

Samsung products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, for other applications intended to support or sustain life, or for any other application in which the failure of the Samsung product could create a situation where personal injury or death may occur.

Should the Buyer purchase or use a Samsung product for any such unintended or unauthorized application, the Buyer shall indemnify and hold Samsung and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorized use, even if such claim alleges that Samsung was negligent regarding the design or manufacture of said product.

S5N8947 32-Bit RISC Microcontrollers
Programmer's Guide, Revision 1.0
Publication Number: 51.0-S5-N8947-032002

© 2002 Samsung Electronics

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electric or mechanical, by photocopying, recording, or otherwise, without the prior written consent of Samsung Electronics.

Samsung Electronics' microcontroller business has been awarded full ISO-14001 certification (BSI Certificate No. FM24653). All semiconductor products are designed and manufactured in accordance with the highest quality standards and objectives.

Samsung Electronics Co., Ltd.
San #24 Nongseo-Ri, Giheung- Eup
Yongin-City, Gyeonggi-Do, Korea
C.P.O. Box #37, Suwon 449-900

TEL: (82)-(031)-209-1946
FAX: (82)-(031)-209-6547

Home Page: http://www.samsungsemi.com
Printed in the Republic of Korea

Table of Contents

Chapter 1 About S5N 8947 Evaluation Board

Chapter 2 Using the ARM SDT for S5N8947 Projects

Chapter 3 Memory Management

Table of Contents (Continued)

Chapter 4 Exception Handling

Chapter 5 Controlling S5N8947 Modules

Table of Contents (Continued)

Chapter 5 Controlling S5N8947 Modules

List of Figures

List of Tables

1 ABOUT S5N8947 EVALUATION BOARD

SYSTEM OVERVIEW

S5N8947 Board supports a code development of SAMSUNG's S5N8947 16/32-bit RISC microcontroller for ADSL and Cable modem applications.

S5N8947 consists of 16-/32-bit RISC(ARM7TDMI) CPU core, 8-Kbyte unified cache, I^2C-bus controller, 2-channel 10/100 Mbps Ethernet controller, SAR (Segmentation and Reassembly), UTOPIA (the Universal Test & Operations PHY Interface for ATM) Interface, Full-rate USB controller, 2-channel GDMA, UART, two 32-bit timers, 18 programmable I/O ports, interrupt controller, SPI interface, PCMCIA interface and a system manager. It also supports JTAG boundary scan for the application system testing.

S5N8947 Board consists of S5N8947 , boot EEPROM(Flash ROM), DRAM module, SDRAM, UART serial communication port, Ethernet interface with two external PHYs, ATM25 PHY interface, configuration switches, and status LEDs. The Ethernet interface has a complete IEEE802.3 physical layer interface with Ethernet hub/router side RJ45 connector configuration. ATM25 PHY interface chip is connected with S5N8947 with UTOPIA interface.

S5N8947 BOARD OVERVIEW

S5N8947 Evaluation Board shows the basic system-based hardware design which uses the S5N8947. It can evaluate the basic operations of the S5N8947 and develop codes for it as well.

When the S5N8947 is contained in the S5N8947 Board, you can use an in-circuit emulator(ICE).

This allows you to test and debug a system design at the processor level. In addition, the S5N8947 with embeddedICE^{TM} capability can be debugged directly using the EmbeddedICE Interface.

The S5N8947 function blocks are shown in Figure 1-1.

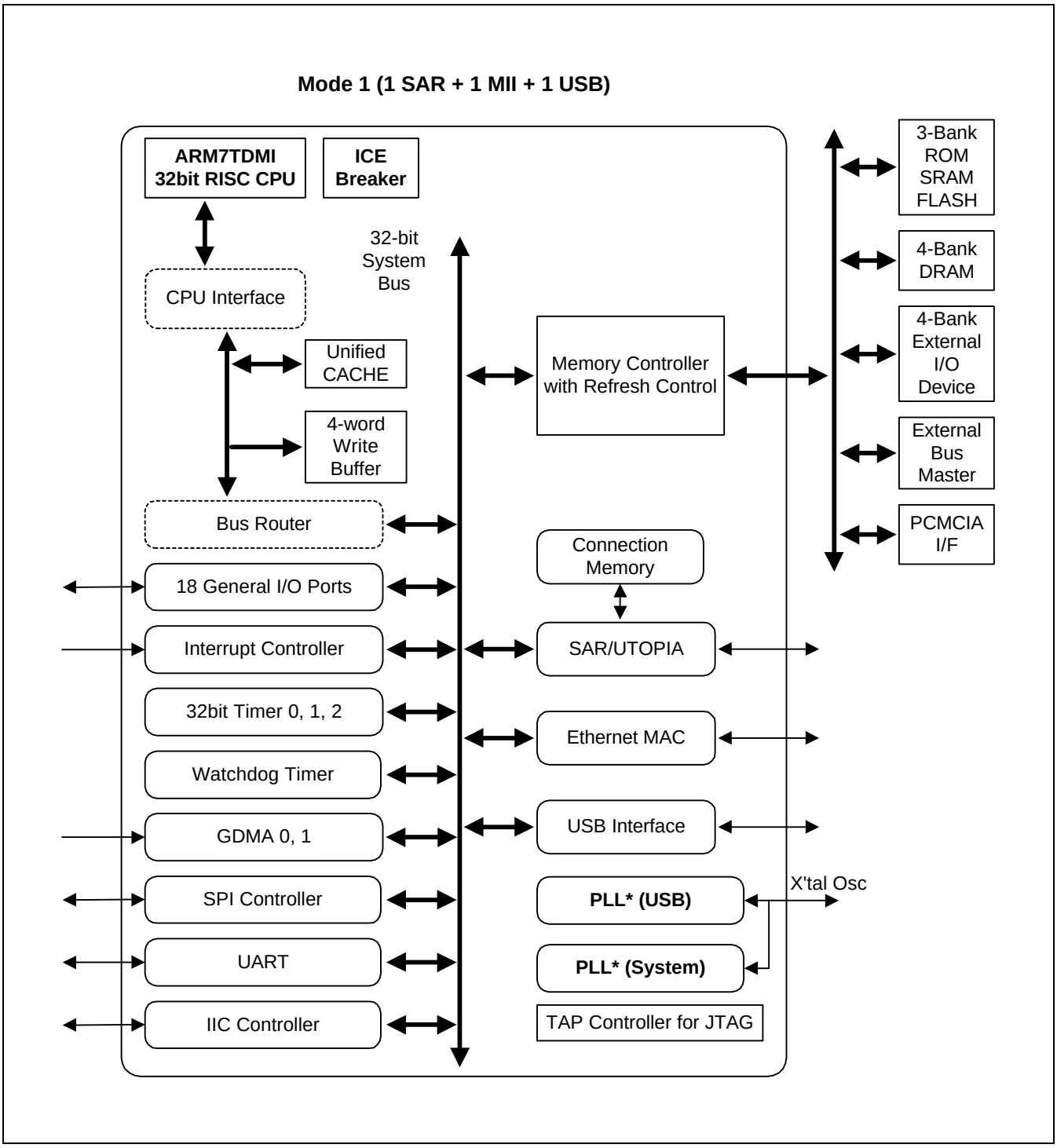


Figure 1-1. S5N8947 Block Diagram (MODE1)

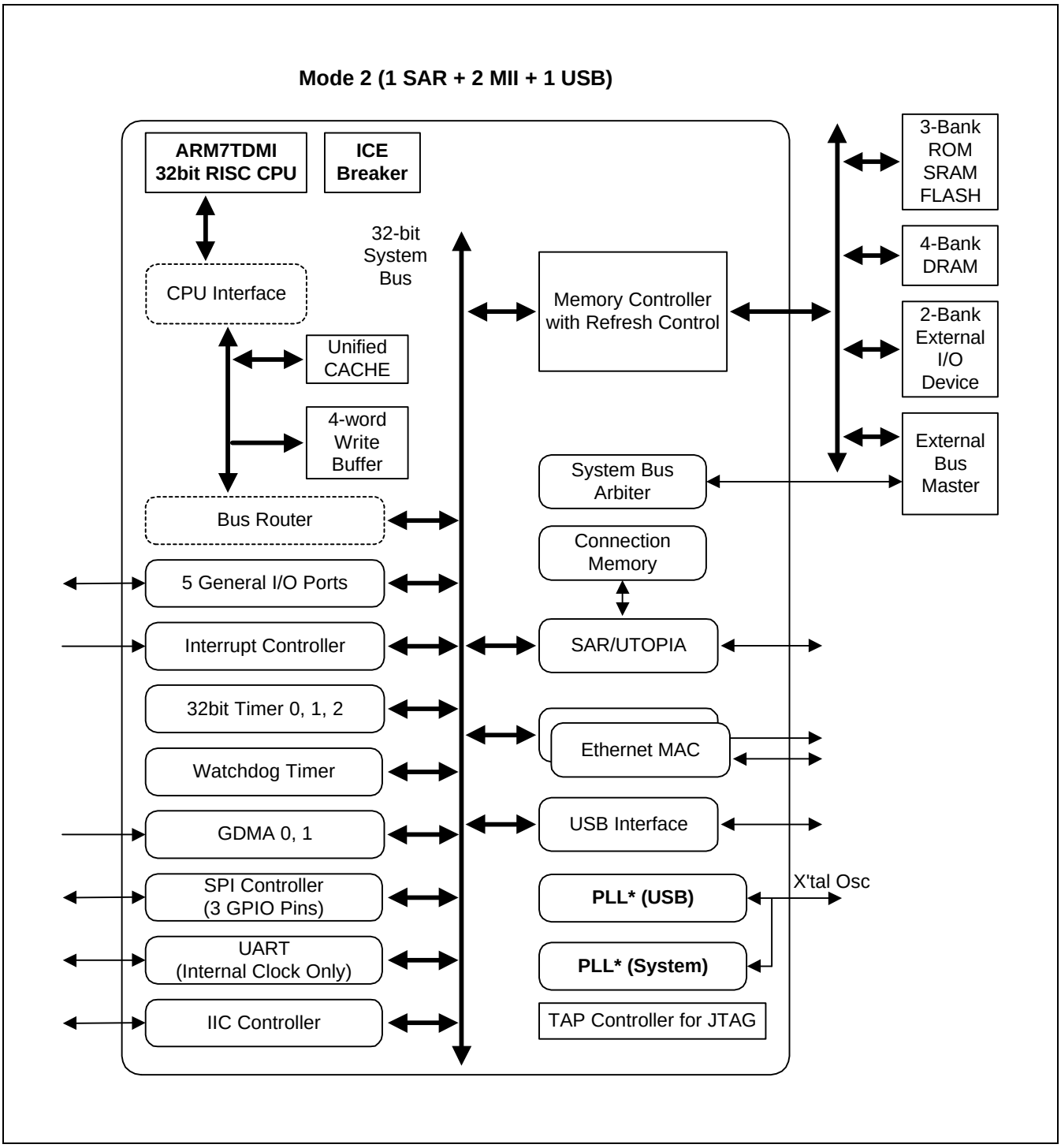


Figure 1-2. S5N8947 Block Diagram (MODE2)

FEATURES

- S5N8947 : 16/32-bit RISC microcontroller
- Boot ROM : 512K bit, 1M bit, 4M bit, support byte, half-word, word size boot ROM
- Flash : 8M bit support byte, half-word, word size
- DRAM : 72-pin SIMM module with two banks and EDO DRAM support
- SDRAM : Two 4Mx16 with 2banks SDRAM support
- General I/O
  - Control switches and status display LED
  - PCMCIA interface
  - 10/100Mbp Expansion Ethernet interface
- One UART serial port
- I2C-bus EEPROM
- SPI Interface
- Full rate USB controller
- ATM SAR block and UTOPIA interface
- 10/100Mbps Ethernet interface with two external PHYs
- EmbeddedICE^{TM} Interface

CIRCUIT DESCRIPTION

S5N8947 Board consists of logic components, several control/status display block, and a debug interface block. S5N8947 Board's detailed block diagram, and its components are shown in Figure 1-3. S5N8947 Board schematics are inserted at the end of this programmer's guide.

POWER SUPPLY

S5N8947 Board is designed to operate at 1.8V, 3.3V, 5V and 12V. Therefore, power to the S5N8947 is supplied through a DC jack power adapter which supports the voltage 5V and 12V and drives the current at least 1.5A. And it is possible to control the power supply by power switch1 (5V) and switch2 (12V).

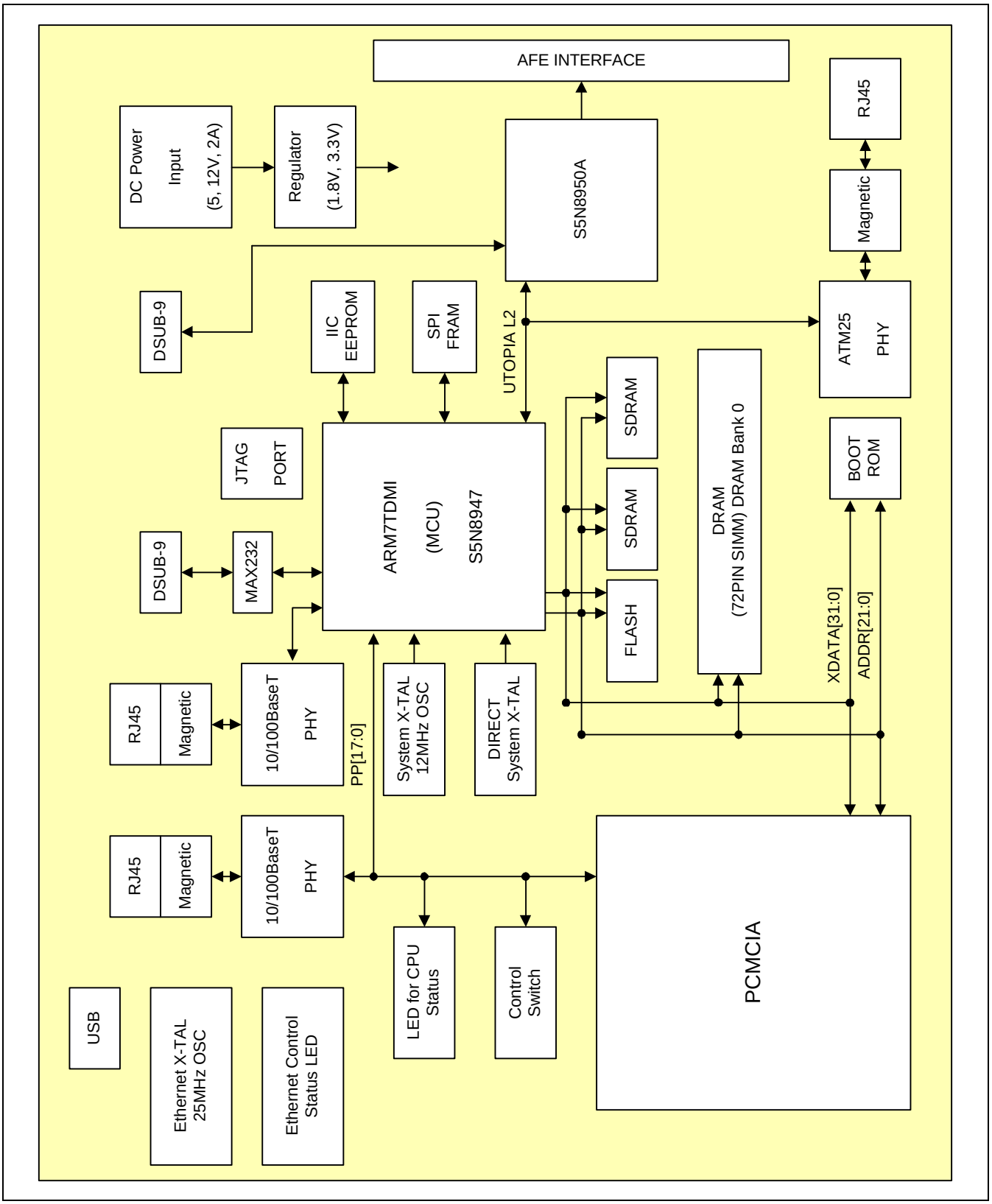


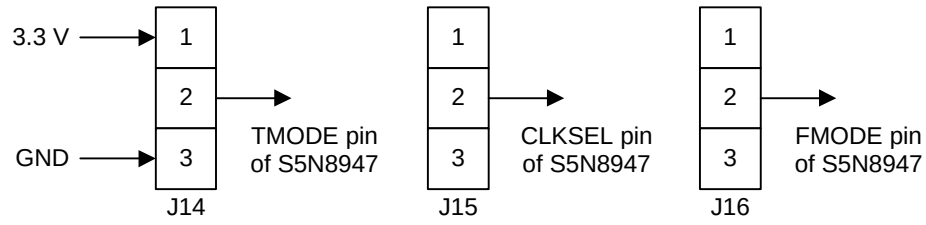
Figure 1-3. Detailed S5N8947 Board Diagram

CLOCK SOURCE AND DISTRIBUTION

The Following clock sources are supported at S5N8947 Board.

System Clock

You can use 12MHz oscillator and direct input (50MHz, 66MHz, 72MHz) oscillator (XCLK_I) for system clock source by jumper setting. The 12MHz clock source is also used for USB clock source with PLL enabled.



TMODE	CLKSEL	FMODE	Mode Description
-	-	2-3	1 Ethernet, PCMCIA, SPI
-	-	1-2	2 Ethernet
2-3	2-3	-	66 MHz
1-2	2-3	-	72 MHz
2-3	1-2	-	Direct clock input

Figure 1-4. System Clock Mode Selection and Functional Mode Selection

System Clock Out (MCLKO)

MCLKO is the same signal as internal system clock of S5N8947. This clock can be monitored at MCLKO pin. If you want to use SDRAM with S5N8947, MCLKO should be used. You can monitor it at MCLKO pin.

Ethernet Control Clock

25MHz crystal or oscillator have to be used for 10/100Mbps Ethernet PHY control clock. In S5N8947 Board, 25MHz oscillator is used.

External UART Clock

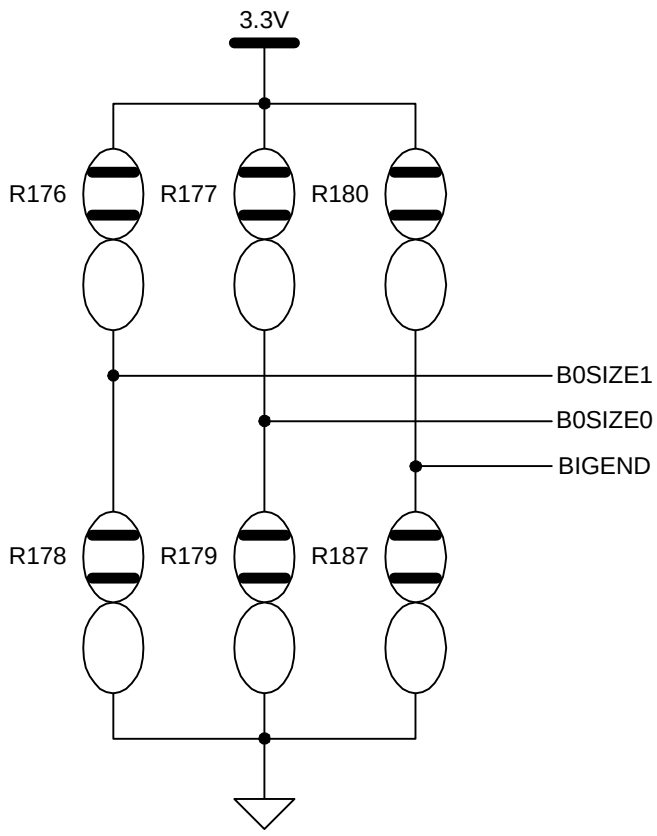
S5N8947 support the External UART Clock input pin (UCLK :U12). But you can also use system clock source with internal PLL as a UART clock source. In S5N8947 Board, the system clock source is used.

RESET LOGIC

The nRESET (System Reset Signal) must be held to low level at least 540 master clock cycles to reset S5N8947. nRESET and nTRST (JTAG Reset signal) are anded logically. But, if you want to use circuit emulator (ex, Embedded ICE) to dubug without BOOT ROM, you should have the nTRST is floated. If not, whenever the ADW (ARM Debug Window) were invoked SW interrupt will be occurred.

S5N8947 System Configurations

S5N8947 Board provides Big-/Little- endian mode with S5N8947 and also contains a selectable resistor for the Byte/Halfword/Word size data bus access of ROM. R176~R179 are used for the selection for the ROM access data bus size. In S5N8947 Board, byte size ROM (B0SIZE1 = "low", B0SIZE0 = "high") and halfword size ROM (B0SIZE1 = "high", B0SIZE0 = "low") is possible to use. R180 and R187 are used for Big-/Little- endian mode selection.



B0SIZE0	B0SIZE1	Description
low	low	Reserved
high	low	Byte
low	high	Half-word
high	high	Word

BIGEND	Description
high	Big endian
low	Little endian

Figure 1-5. Boot Device Size Selection and Endian Mode Selection

DRAM/SDRAM Configurations

S5N8947 Evaluation board has the 72-pin SIMM module on the board for one bank DRAM. S5N8947 can support Synchronous DRAM (SDRAM). In this case, SDRAM or DRAM memory can be selected alternatively using by SYSCFG register. Using these devices with S5N8947 Evaluation board, RAS selection jumpers has to be set to select DRAM or SDRAM.

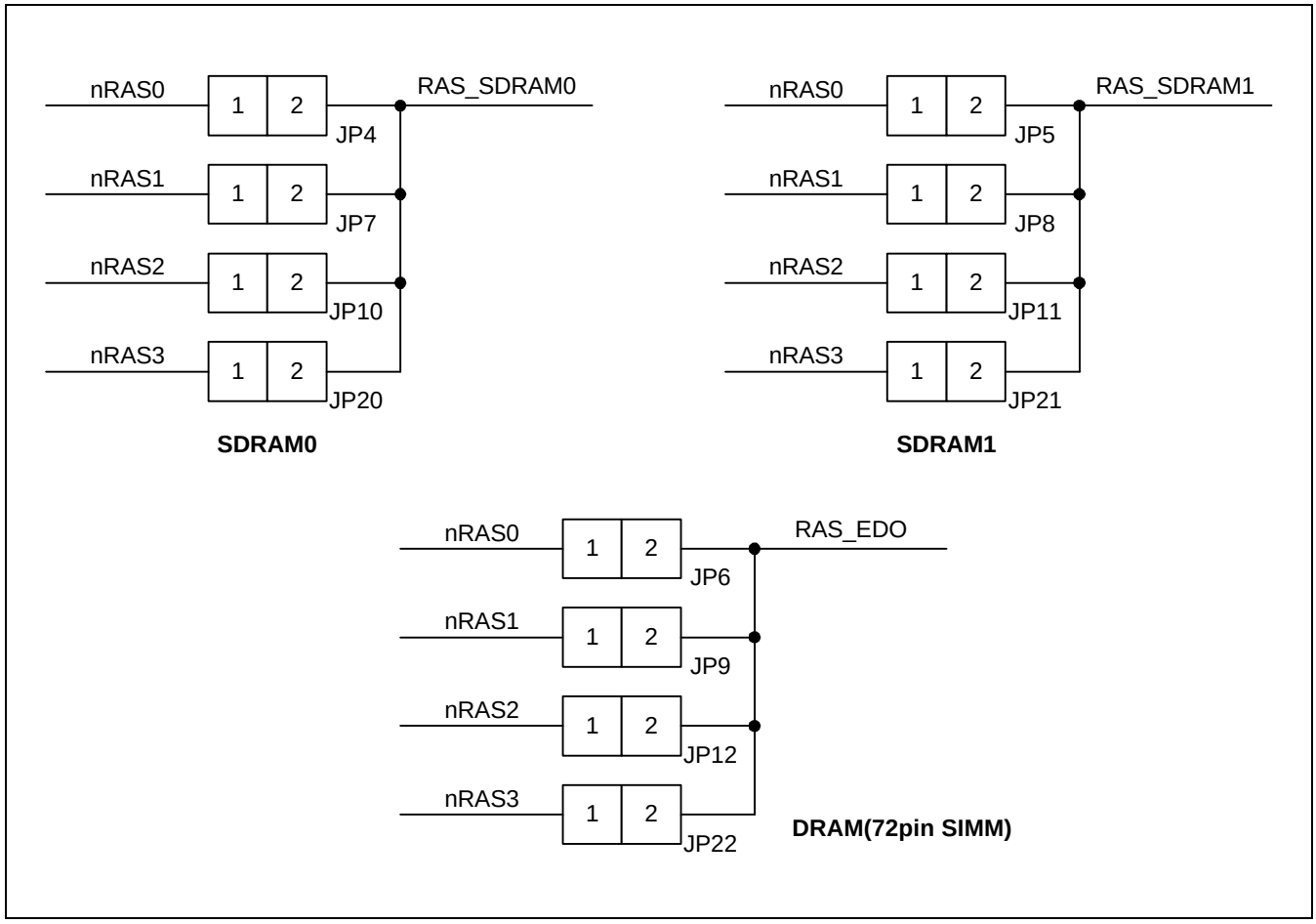


Figure 1-6. Extended Device Chip Selection

Bank select jumpers for DRAM/SDRAM, RAS selection jumpers on S5N8947 Evaluation board are provided just only for the purpose of each bank test. So, you want to use SDRAM, you have to enable a SDRAM bank and remove same DRAM bank's jumper.

BOOT ROM code find out the type of memory which is installed on S5N8947, and then initialize the memory banks, base/end pointer and the timing of CAS/RAS after the system power on reset or the reset key pressed and released. If DRAM banks are found, each bank can be configured as an EDO DRAM mode using the system management block DRAM bank control register.

GENERAL I/O PORTS

S5N8947's general I/O ports are used for several purpose; key interrupt input, LED status display, second Ethernet PHY interface and PCMCIA interface. So the signal collision might be occurred when each interface is tested, therefore, you should set the collision protecting switch (S1,S2,S8,S9,S11) on S5N8947 evaluation board before each function is operated. GPIO and the switch configurations are shown in following Table 1-1.

Table 1-1. General I/O and the Switch Configurations on S5N8947 Evaluation Board

Function	Switch Setting	Description
LED Status Display	S1: ON, J16(low) S2, S8, S9, S11: OFF	PP[7:0] : D3 – D10
Key Interrupt Input	S2: ON, J16(low) S1, S8, S9, S11: OFF	PP5 – PP8,PP10 : S3 – S7
Second Ethernet	S8, S9: ON, J16(high) S1, S2, S11: OFF	PP4 (sTXD0), PP5(sTXD1), PP6(sTXD2), PP7(sRX_DV), PP11(sCRS), PP12(sRXD0), PP13(sRXD1), PP14(sRXD2), PP15(sRXD3), PP17(sRX_ERR)
PCMCIA	S11: ON, J16(low) S1, S2, S8, S9: OFF	PP0(PnCE1), PP1(PnCE2), PP2(PIOISI6), PP3(ALE) PP5(PnIREQ), PP6(PnCD), PP7(VS1), PP8(VS2), P10(PBVD1), PP12(PRESET), PP13(BufCon), P14(DATA_P), PP15(CLOCK_P), PP17(LATCH_P)

The function of control switch and the status of LED can be defined by user software.

MODEM CLOCK

S5N8947 Evaluation board has two method for modem clock input. You should set the modem clock setting resistor (R152, R153)

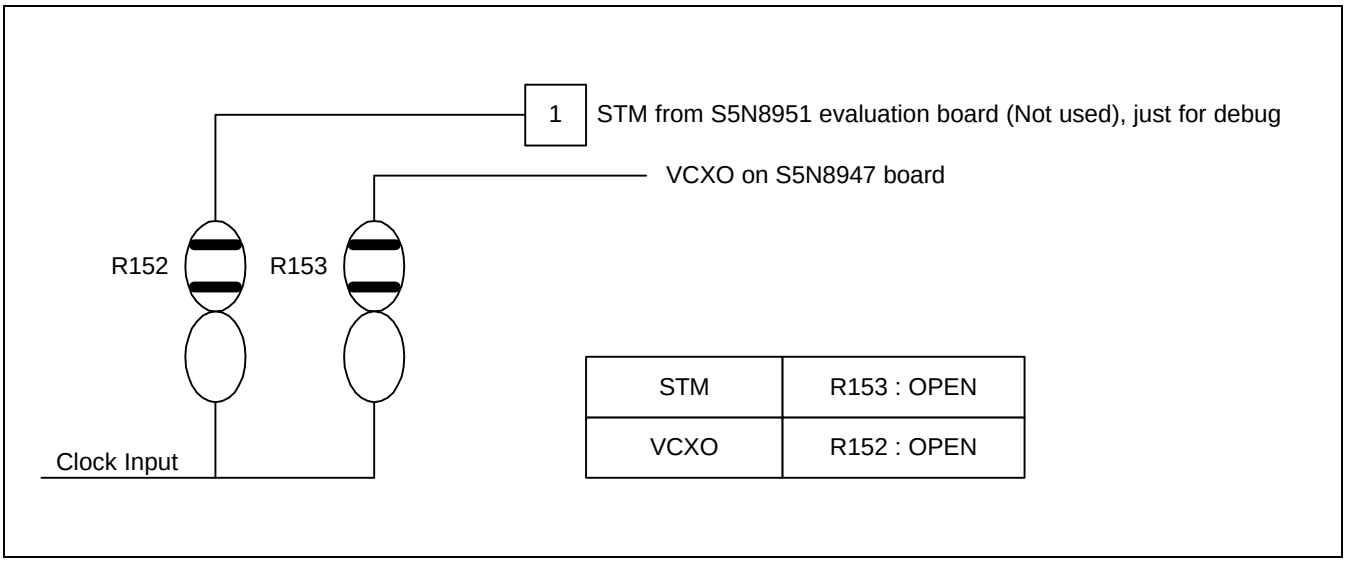


Figure 1-7. ADSL Modem Clock Signal Selection

ETHERNET INTERFACE

S5N8947 has two 10/100-Mbps Ethernet controller. S5N8947 Evaluation board supports two 10/100-Mbps Ethernet interface, Two External PHY chips used in S5N8947 Board is able to operate 10/100-M bps.

S5N8947 Board's Ethernet connector(RJ45) has Ethernet adapter side pin configuration which supports communication with the host PC's NIC. You can also connect S5N8947 Evaluation board to hub or router direct without twisting cable.

Table 1-2. RJ45 Pin Configurations for Adapter Side

[External PHY interface: J10]

Pin Number	Descriptions	Pin Number	Descriptions
1	CMT	5	NC
2	CT_T	6	CT_R
3	TXD+	7	RXD+
4	TXD-	8	RXD-

Ethernet Status LED

- LED location: D11 – D20

Table 1-3. Ethernet Status LED.

LED	Functions	Descriptions
D11,D16	100Mbps	Collision detection LED for External PHY
D12,D17	Collision	Collision detection LED for External PHY
D13,D18	100Mbps(idle), 10Mbps(Link)	Link integrity LED for external PHY.
D14,D19	Transmitter	Transmit data LED for external PHY.
D15,D20	Receiver	Receive data LED for external PHY

Serial (UART) & JTAG Interface

S5N8947 Evaluation board supports a 9DIP SUB serial connector for UART.

S5N8947 MCU supports JTAG port[JP2]. It can be used as Circuit Emulator(ex, Embedded ICE) interface for boundary scan test and debugging channel for application.

NOTES

2 USING THE ARM SDT FOR S5N8947 PROJECTS

This chapter explains how to use the ARM Software Development Toolkit (ARM SDT) to set up a project for developing and debugging code for the S5N8947 microcontroller. Information is presented according to the following Table of Contents:

INTRODUCTION

The ARM Software Development Toolkit consists of two applications that let you to write and debug applications for the S5N8947 microcontroller, with its embedded ARM7TDMI RISC core:

- Project Manager : Write source code, and build the source code into image files or libraries
- Debugger : Debug your source files.

The ARM SDT is used mainly for software development. This involves building either C or ARM assembler source code into ARM object code, which can then be debugged using the ARM symbolic debugger. The debugger software supports single stepping, breakpoint and watch point settings, and register viewing. You can perform testing and debugging on code running under a emulation in the host system, or on a S5N8947 target board system.

The following section reviews the specific SDT components as they relate to S5N8947 projects, and provides a brief description of a typical application development working flow.

For complete documentation on the ARM SDT, please refer to the "*ARM Software Development Toolkit User's Guide*."

S5N8947 DEVELOPMENT ENVIRONMENT

We recommend the following hardware environment for S5N8947 application development and debugging:

- Host computer: An IBM compatible PC running Windows 95/98 or Windows NT 4.0/2000, with two serial ports and one parallel port. (One serial port is used to interface with Embedded ICE. The other serial port can be used for UART communication, if required.)
- Evaluation board: The type of board used depends on the target system.
- EmbeddedICE: To support the debugging interface using the S5N8947 JTAG controller.

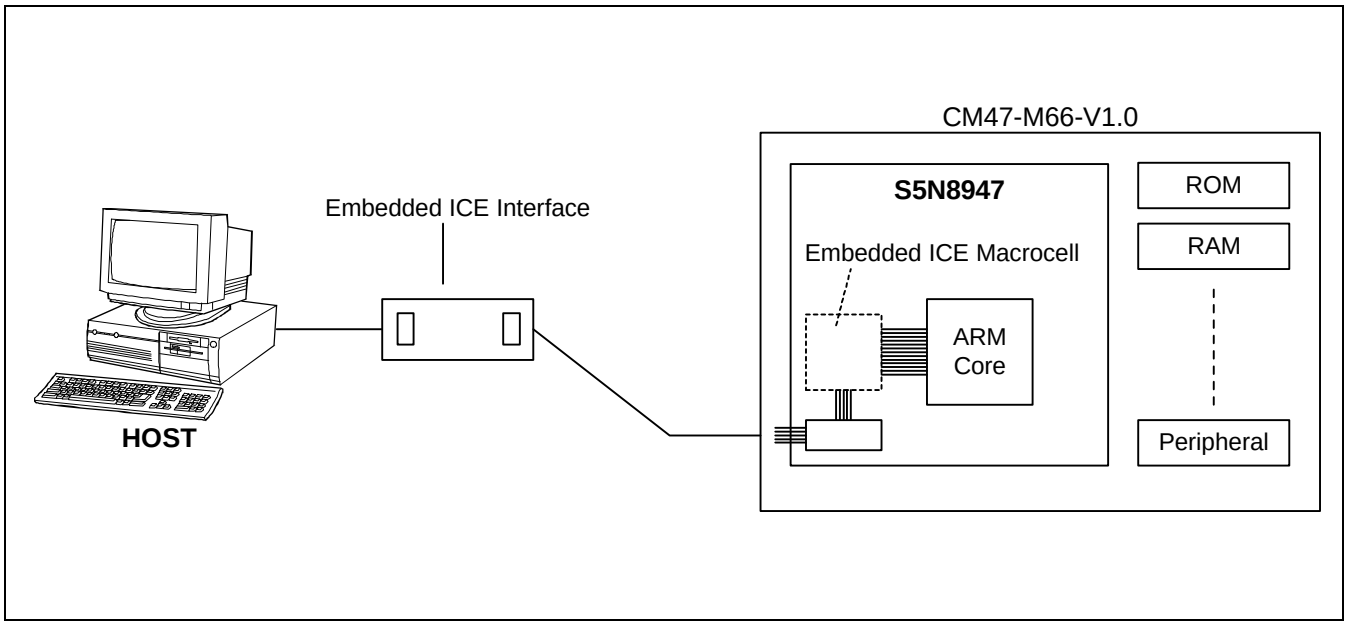


Figure 2-1. S5N8947 Development Environment

BUILDING A DEMO PROJECT (WINDOWS 95/98 OR NT/2000)

To build a S5N8947 demo project, you must perform the following operations using the Project Manager. (Sample source files for a typical demo project can be found on the S5N8947 “demo diskette”.)

Creating a New Project

1. Select **New** from the File menu.
2. Choose Project from the New Dialog box.
3. Choose ARM Executable **Image** as the project type from the list of project templates.
4. Enter the **Project Name** and **Project Directory**. The project filename extension **.apj** is added automatically. The name of the executable target is the same as the project name. When you have created a new project, the information in the **Project Window** is displayed as a hierarchical flow diagram.

In the Project Window, you can select one of two project variants. These variants are automatically constructed from project templates supplied with the ARM Project Manager:

Debug: For creating a target image that is suitable for debugging, and which includes debugging information.

Release: For creating a target images that is suitable for release, but which includes no debugging information.

Adding Source Files to the Project

1. Select Add Files to Project from the Project menu.
2. Choose the files you wish to add. The project window is updated to reflect the change.
3. Select Debug as the project variant.

Setting C Compiler Options

1. Select Tool Configuration for <project_name>. apj from the Project menu, and choose <cc>=armcc.
2. Configure the compiler options as follows:

   In Target page: [*Processor*] | ARM7TM
   [*Byte Format*] | Big Endian
3. For the remaining options, select the default values.

Setting Assembler Options

1. Select **Tool Configuration for <project_name>.** apj from the Project menu, and choose <asm>=armasm
2. Configure the assembler options as follows:
2. Configure the compiler options as follows:

   For the Target page: [Processor] | ARM7TM
   [Byte Format] | Big **Endian**
3. For the remaining options, select the default values.

Setting Linker Options

1. Select Tool Configuration for <project_name>. apj from the Project menu, and choose armlink.
2. Configure the linker options as follows.

   For the **Output** page: [*Output Formats*] | **Absolute ELF**

   For the **Entry and Base** page: [*Base of Image*]
   <*Read-Only*> | **0x1300000** (start address of code area)
   <*Read-Write*> | ***blank*** (start address of data area)

NOTE

When leaving Read-Write base blank, the Read-Write area starts from the next address to the end of Read-Only Area.

For the **Image Layout** page: [Place at start of image]
<Object File> | **Init.o** (beginning object file name)
<Area Name> | INIT (beginning area name)

3. For the remaining options, select the default values.

NOTE

For additional options and for descriptions of the compiler, assembler, and linker, please refer to chapters 1, 2, and 3 of the "*ARM Software Development Toolkit Reference Guide*."

Building Your Project

When you have added the files you require, and have set up the global options and parameters for specific files, you are ready to build your project:

1. Select **Build <project_name>. apj "Debug"** from the Project menu or click the **Build** button. The “Force Build” option lets you rebuild all project files simultaneously.
2. The current build status is reflected in the **Build Log** which appears at the bottom of the Project Window when you build the project.

USING COMMAND LINE TOOLS TO BUILD PROJECTS

Three activities are required to build a project: compilation, assembly, and linking. You can perform these activities using the toolkit components, or you can enter the commands directly on the command line:

Compile: **armcc** < -options > < C source file >

Assemble: armasm < -options > < assembly source file >

Link: **armlink** < -options > < object files list >

When using the command line to build a project, it is convenient to use the **armmake** utility. Before you can execute armmake, you must create a **makefile** in the working directory. (A sample makefile is included in the S5N8947 demo project.)

armmake (parameters not required)

Sample Makefile

```
ARMINC = c:\isiarm\arm251\include
ARM_LIB = c:\isiarm\arm251\lib\armlib_cn.32b
BSP_DIR = BSP

#-------------------------------------------------------------------------*
# Common rules for this BSP                                              *
#                                                                        *
# RAMOPTS  - Linker flags for location of ram.axf/ram.bin image          *
# ROMOPTS  - Linker flags for location of rom.bin image                  *
#-------------------------------------------------------------------------*

RAMOPTS         = -RO -Base 0x1000050
ROMOPTS         = -RO 0x0 -RW 0x1300000

#*************************************************************************
# A few important make options.  The following must be defined:          *
#                                                                        *
# CC          - Command to run the C compiler                            *
# AS          - Command to run the assembler                             *
# OBJ_DIR     - Name of the object dirtectory for this model (note that  *
#               it must end with a '\')                                  *
# BSP_AOPTS   - Assembler options specific to this model                 *
# BSP_COPTS   - Compiler options specific to this model                  *
# INTERWORK   - Set to /interwork to allow ARM/THUMB interworking        *
#                                                                        *
# In these options, you can set following modes.                         *
#    Endian : Big / Little                                               *
#    Processor modes : ARM / THUMB                                       *
#*************************************************************************
OBJ_DIR  = OBJ

CC       = armcc        # or, = tcc
AS       = armasm       # or, = tasm
ENDIAN   = -bi          # or, = -li
INTERWORK=              # or, = /interwork

BSP_AOPTS= $(ENDIAN) #-PD "BSP_LITTLE_ENDIAN SETL {FALSE}" -PD "THUMB SETL {FALSE}"
BSP_COPTS= $(ENDIAN) -DBSP_LITTLE_ENDIAN=0
```

```
DBGFLG  = -g            # or, =

#--------------------------------------------------------------------------*
# ARM C compiler options                                                   *
#--------------------------------------------------------------------------*
COPTS1  = $(DBGFLG) -zz-1 -za1 -fy -c
COPTS2  = -I$(BSP_DIR) -W -G -I$(ARMINC)
COPTS3  = -apcs 3/noswst/nofp$(INTERWORK) -cpu ARM7TM -ARCH 4T
COPTS4  = $(BSP_COPTS) -errors bspc.err

COPTS   = -VIA c.opt

#--------------------------------------------------------------------------*
# ARM assembler options                                                    *
#--------------------------------------------------------------------------*
AOPTS1  = $(DBGFLG) -I$(BSP_DIR) -I$(ARMINC)
AOPTS2  = -cpu ARM7TM -ARCH 4T -apcs 3/noswst/nofp
AOPTS3  = $(BSP_AOPTS) -errors bspa.err

AOPTS   = -VIA a.opt

#--------------------------------------------------------------------------*
# ARM linker options                                                       *
#--------------------------------------------------------------------------*
LOPTS = -info interwork -remove -nozeropad -MAP -Symbols - -list ram.map
DEBUG   = -Debug

#--------------------------------------------------------------------------*
# Command Macros                                                           *
#--------------------------------------------------------------------------*
LD      = armlink
LIB     = armlib
MKDIR   = md
ECHO    = echo
RM      = del

#--------------------------------------------------------------------------*
# Macros for various lists of object modules                               *
#--------------------------------------------------------------------------*
RAM_OBJ = $(OBJ_DIR)\begin.o
ROM_OBJ = $(OBJ_DIR)\rombegin.o
```

```
#*********************************************************************
# BSP OBJECT LIST                                                    *
# These form the dependency list for everything in the BSP           *
#*********************************************************************
BSP_OBJ1 = $(OBJ_DIR)\init.o     $(OBJ_DIR)\except.o $(OBJ_DIR)\sysinit.o
BSP_OBJ2 = $(OBJ_DIR)\intrhndl.o $(OBJ_DIR)\timer.o  $(OBJ_DIR)\uart.o
BSP_OBJ3 = $(OBJ_DIR)\Diagnosis.o $(OBJ_DIR)\i2c.o $(OBJ_DIR)\MemoryTest.o
BSP_OBJ4 = $(OBJ_DIR)\GDMATest.o $(OBJ_DIR)\IOPort.o $(OBJ_DIR)\down.o
BSP_OBJ5 = $(OBJ_DIR)\SPI.o $(OBJ_DIR)\USBfunc.o $(OBJ_DIR)\usb.o
BSP_OBJ6 = $(OBJ_DIR)\sramtest.o $(OBJ_DIR)\EMFlash.o $(OBJ_DIR)\AllTest.o
BSP_OBJS = $(BSP_OBJ1) $(BSP_OBJ2) $(BSP_OBJ3) $(BSP_OBJ4) $(BSP_OBJ5) $(BSP_OBJ6)

#*********************************************************************
# APPLICATION OBJECT LIST                                            *
# These form the dependency list for everything in the APPLICATION   *
# Add your applicaiton object file lists.                            *
#*********************************************************************
APP_OBJ1 = $(OBJ_DIR)\root.o
APP_OBJS = $(APP_OBJ1)

#*********************************************************************
# Rules for target image type.                                       *
#                                                                    *
#    1. ram.axf - ARM executable image for embbedded ICE.            *
#    2. ram.bin - ARM plain binary image for downloading to RAM.     *
#    3. rom.bin - ARM plain binary image for ROM.                    *
#                                                                    *
# Most of the linker command file is common so it is generated with a*
# common rule.  If a make is aborted, the file lnkram.cmd must be    *
# removed.                                                           *
#*********************************************************************
ram.axf: c.opt a.opt dir \
    $(RAM_OBJ) $(BSP_OBJS) $(APP_OBJS)
    @$(ECHO) $(LOPTS) $(RAM_OBJ) $(BSP_OBJS) $(APP_OBJS) $(ARM_LIB) \
    > lnkram.opt
      $(LD) -first begin.o(BEGIN) $(DEBUG) -elf -o ram.axf $(RAMOPTS) \
    -VIA lnkram.opt
#     @$(RM) *.opt

ram.bin: c.opt a.opt dir \
    $(RAM_OBJ) $(BSP_OBJS) $(APP_OBJS)
    @$(ECHO) $(LOPTS) $(RAM_OBJ) $(BSP_OBJS) $(APP_OBJS) $(ARM_LIB) \
    > lnkram.opt
      $(LD) -first begin.o(BEGIN) -nodebug -bin -o ram.bin $(RAMOPTS) \
    -VIA lnkram.opt
#     @$(RM) *.opt

rom.bin: c.opt a.opt dir \
    $(ROM_OBJ) $(BSP_OBJS) $(APP_OBJS)
    @$(ECHO) $(LOPTS) $(ROM_OBJ) $(BSP_OBJS) $(APP_OBJS) $(ARM_LIB) \
    > lnkram.opt
      $(LD) -first rombegin.o(ROMBEGIN) -nodebug -bin -o rom.bin $(ROMOPTS) \
    -VIA lnkram.opt
#   @$(RM) *.opt
```

```
#-------------------------------------------------------------------------*
# Rules for board specific files                                          *
#-------------------------------------------------------------------------*
$(OBJ_DIR)\begin.o: $(BSP_DIR)\begin.s $(BSP_DIR)\board.a \
      makefile
      $(AS) $(AOPTS) $(BSP_DIR)\begin.s  -o $(OBJ_DIR)\begin.o

$(OBJ_DIR)\rombegin.o: $(BSP_DIR)\rombegin.s $(BSP_DIR)\board.a \
      makefile
      $(AS) $(AOPTS) $(BSP_DIR)\rombegin.s  -o $(OBJ_DIR)\rombegin.o

$(OBJ_DIR)\init.o: $(BSP_DIR)\init.s $(BSP_DIR)\board.a \
      makefile
      $(AS) $(AOPTS) $(BSP_DIR)\init.s  -o $(OBJ_DIR)\init.o

$(OBJ_DIR)\except.o: $(BSP_DIR)\except.s $(BSP_DIR)\board.a \
      makefile
      $(AS) $(AOPTS) $(BSP_DIR)\except.s  -o $(OBJ_DIR)\except.o

$(OBJ_DIR)\intrhndl.o: $(BSP_DIR)\intrhndl.c $(BSP_DIR)\isr.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\intrhndl.c  -o $(OBJ_DIR)\intrhndl.o

$(OBJ_DIR)\sysinit.o: $(BSP_DIR)\sysinit.c $(BSP_DIR)\isr.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\sysinit.c  -o $(OBJ_DIR)\sysinit.o

$(OBJ_DIR)\timer.o: $(BSP_DIR)\timer.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\timer.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\timer.c  -o $(OBJ_DIR)\timer.o

$(OBJ_DIR)\uart.o: $(BSP_DIR)\uart.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\uart.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\uart.c  -o $(OBJ_DIR)\uart.o

$(OBJ_DIR)\Diagnosis.o: $(BSP_DIR)\Diagnosis.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\Diagnosis.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\Diagnosis.c  -o $(OBJ_DIR)\Diagnosis.o

$(OBJ_DIR)\i2c.o: $(BSP_DIR)\i2c.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\i2c.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\i2c.c  -o $(OBJ_DIR)\i2c.o

$(OBJ_DIR)\SPI.o: $(BSP_DIR)\SPI.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\SPI.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\SPI.c  -o $(OBJ_DIR)\SPI.o

$(OBJ_DIR)\EMFlash.o: $(BSP_DIR)\EMFlash.c $(BSP_DIR)\board.h $(BSP_DIR)\isr.h \
      $(BSP_DIR)\EMFlash.h \
      makefile
      $(CC) $(COPTS) $(BSP_DIR)\EMFlash.c  -o $(OBJ_DIR)\EMFlash.o
```

```
#-------------------------------------------------------------------------*
# Rules for application files                                             *
#-------------------------------------------------------------------------*
$(OBJ_DIR)\root.o: root.c $(BSP_DIR)\board.h \
      makefile
      $(CC) $(COPTS) root.c -o $(OBJ_DIR)\root.o

#*************************************************************************
# Rules for making some of the assembler/compiler option files used      *
# to build applications.  These files are needed on in DOS Windows       *
# on Win98/NT due to command line length limitations.                    *
#*************************************************************************
#-------------------------------------------------------------------------*
# Cleanup and other rules                                                *
#-------------------------------------------------------------------------*
clean:
    @echo off
      @$(RM) *.opt
      @$(RM) $(OBJ_DIR)\*.o
      @$(RM) *.err
      @$(RM) *.axf
      @$(RM) *.bin
      @$(RM) *.map
    @echo on
      @echo "DONE"

c.opt: makefile
      @echo $(COPTS1)   >  c.opt
      @echo $(COPTS2)   >> c.opt
      @echo $(COPTS3)   >> c.opt
      @echo $(COPTS4)   >> c.opt

a.opt: makefile
      @echo $(AOPTS1)  >  a.opt
      @echo $(AOPTS2)  >> a.opt
      @echo $(AOPTS3)  >> a.opt

dir:
      @$(MKDIR) $(OBJ_DIR)
```

DEBUGGING AND EXECUTING YOUR DEMO PROJECT

When you build a project, the Project Manager creates a new subdirectory called ‘Debug’ or ‘Release’, depending on the variant you select. This subdirectory is located below the current working directory. Several object files are placed in this variant subdirectory. The executable ARM image code, **<project_name>.axf**, is also generated in this subdirectory.

If you are using the ARM Project Manager, you can then start the ARM Debugger for Windows and load the image of the application you are developing. This procedure is described below.

STARTING THE DEBUGGER

There are two ways to start the debugger and load the executable image:

Using the ARM Project Manager

Click the Debug button of the Project Manager to open the ARM Debugger For Windows. The Debugger loads your project image automatically and breaks at the first line of the code.

Not Using the ARM Project Manager

1. Execute the ARM Debugger For Windows. (This software is included in the “ARM Windows Toolkit 2.51” program group).
2. Choose **Load Image** from the **File** menu or click the **Open File** button to select the ARM Image file (project_name.axf) you want to load.

CONFIGURING THE DEBUGGER

1. Select **Configure Debugger** from the **Options** menu. The **Debugger Configuration** dialog box is displayed. There are two **Target Environments**, as follows.

   ARMulator: Lets you execute ARM programs without physical ARM hardware by simulating ARM instructions in software.

   Remote_A: Connects the ARM debugger directly to a target board or to an EmbeddedICE unit that is attached to a target board.

   NOTE

   Please note the following requirements for using Remote_A: (1) a direct target board connection requires Angel debug monitor software, (2) the EmbeddedICE software must be version 2.0 or greater, and (3) application programs must be linked using the SDT 2.5 C language library.

   Select the appropriate variants for your application, and click the **Configure** button. The **ARMulator Configuration** or **Angel Remote Configuration** dialog box is displayed.

2. For ARMulator configuration, set the Processor Variant to ARM7TDMI and the Processor Clock Speed to 33.00 MHz. For Angel Remote Configuration, set the Remote Connection type, Ports, and Serial Line Speed to the proper values for your system environment.
3. Select the Debugger page and set the Endian value to Big.
4. Click OK. If you are using Remote_A, the image file is downloaded to the DRAM.

EXECUTING THE IMAGE FILE

1. Initialize system variables. After a download, several windows are displayed such as the **Execution** window, **Console** window, and the **Command** window. In the Command window, you must initialize the system variables, *$semihosting_enabled* and *$vector_catch*, by entering the following commands.

   let $vector_catch = 0x00
   let $semihosting_enabled = 0x00

   Or, you can initialize these variables in the following way:

   First, create a text file called “armsd.ini”, that includes the above commands. Then, enter the following command in the command window:

   ob c:\arm251\armsd.ini

   For more information about this step, please refer to chapter 6 of the “ARM Software Development Toolkit User Guide.

2. Execute the program. To do this, select **Go** from **Execute** menu, or click the **Go** button. When you do this, program execution starts. When you execute the image, the program is displayed in the Execution window as source. The program halts at any breakpoints or watchpoints you have applied.

STEPPING THROUGH THE PROGRAM

To step through the program execution flow, you can select from the following three options:

- **Step**: Advances the program to the next line of code that is displayed in the Execution window.
- **Step Into**: Advances the program to the next line of code that follows all function calls. If the code is in a called function, the function source is displayed in the Execution window and the current code line is highlighted.
- **Step Out**: Advances the program from the current function to the point from which it was called immediately after the function call. The appropriate line of code is displayed in the Execution window.

SETTING A BREAKPOINT

A breakpoint is a point you set in program code where the ARM debugger will halt program execution. When you set a breakpoint, it appears as a red marker in the left side of the window.

To set a simple breakpoint on a line of code, follow these steps:

1. Double-click on the line where you want to place the break, or choose Toggle Breakpoint from the Execute menu. The Set or Edit Breakpoint dialog box appears.
2. Set the count to the required value or expression, as required. (The program only halts when this expression is true.)

To set a breakpoint on a line of code within a particular program function:

1. Display a list of function names by selecting Function Names from the View menu.
2. Double-click on the Function Name you wish to open. A new source window is displayed containing the function source.
3. Double-click on the line where the breakpoint is to be placed, or choose Toggle Breakpoint from the Execute menu. The Set or Edit Breakpoint dialog box appears.
4. Set the count to the required value or expression, as required. (The program only halts when this expression is true.)

SETTING A WATCHPOINT

A watchpoint halts a program when a specified register or variable changes or becomes a specified value.

To set a watchpoint, follow these steps:

1. Display a list of registers, variables, and memory locations you wish to watch, by selecting the Registers, Variables, and Memory options from the View menu.
2. Click the register, variable, or memory area in which you wish to set the watchpoint. Then choose Set or Edit Watchpoint from the Execute menu or the Context menu. The Set or Edit Watchpoint dialog box appears.
3. Enter a Target Value in the Set or Edit Watchpoint dialog box. Program execution halts when the variable reaches the specified target value.

VIEWING VARIABLES, REGISTERS, AND MEMORY

You can view and edit the value of variables, registers, and memory by choosing the respective heading from the **View** menu, as follows:

Variables for global and local variables.

Registers for the current mode and for each of the six register view modes.

Memory for the memory area defined by the address you enter.

DISPLAYING THE CODE INTERLEAVED WITH THE DISASSEMBLY

To display the source code interleaved with the disassembly, choose **Toggle Interleaving** on the **Options** menu or the Context menu. This command toggles between Displaying source only and Displaying source interleaved with disassembly. When source code is shown interleaved with disassembly, machine instructions appear in a lighter gray color.

For additional information about the ARM Debugger, please refer to Chapter 3 of the "*ARM Software Development Toolkit User's Guide*."

COMMUNICATING WITH THE HOST USING UART

When debugging or executing a program, it is useful to monitor communications between the S5N8947 MCU and the host. Using UART and a communication terminal program, you can view data or messages that are transferred from S5N8947 and you can transmit data or a command to S5N8947.

Using the HyperTerminal program that is supported by Windows 95/98 or NT/2000, the host can communicate with the S5N8947's UART module. (You can also use other communication terminal programs.) Guidelines for using the HyperTerminal and UART functions are described below.

USING HYPERTERMINAL

To start the HyperTerminal program (on Windows 95/98 or NT/2000):

1. Select the Start | Programs | Accessories | HyperTerminal group.
2. Double click on Hypertrm.exe, enter the Connection Name, and select the Icon. Click on OK.
3. In the Phone Number dialog box, choose Direct to COM1 or COM2 as the Connect Using setting from the list of communication port templates for your system environment. Click on OK. The COM1 or COM2 Properties dialog box is displayed.
4. Set **Bits per Second** to its proper value. The other properties should be set to their default value.

You have now created a new empty terminal that is connected to the specified serial port. The terminal window is displayed.

HYPERTERMINAL SETUP FOR UART COMMUNICATION

Follow these steps to set up the HyperTerminal software for UART communication with the S5N8947:

1. Select **Properties** from the File menu.
2. When the <Connection_Name> Properties window is displayed, click the Configure button.
3. Click the ASCII Setup button, and select the following check menus:

Baud Rate	38400
Data Bit	8
Parity	None
Stop Bit	1
Flow Control	None

NOTE

These values are default values for S5N8947 Diagnostic ROM. User can change these values when needed after change UART Configuration.

USING THE S5N8947 UART WITH HYPERTERMINAL

For information about how to set up the UART, please refer to the flowcharts and sample source code in Chapter 4 this manual. To send and receive messages to/from the host using the S5N8947 UART, follow these steps:

Sending a Message to the Host

When you have properly set up the UART, you can send a message to the host by entering the following command:

```
Print("Hello!... \n");
```

Receiving a Message from the Host

To receive a message from the host, enter the following command:

```
message = getch( );
```

3 MEMORY MANAGEMENT

This chapter describes the default S5N8947 memory map and explains how to write programs to allocate memory areas. Information is presented according to the following Table of Contents:

INTRODUCTION

The S5N8947 MCU uses its System Manager block to manage memory. Specifically, the System Manager uses special registers to manage the control signals, addresses, and data those are required by external devices.

To control external memory operations, the System Manager uses programmed settings in a dedicated set of special registers. You can, for example, specify the memory type, the external data bus width for each bank, and the number of access cycles for each memory bank. You can also define the memory map by specifying bank locations and sizes that correspond to address spacing assignments.

Using these special registers, you can configure up to three ROM banks, one PCMCIA bank, four SRAM banks, four DRAM banks, and four external I/O banks in a 32-Mbyte addressable system space. The ordering of these banks with the system space is arbitrary.

For more information about the System Manager, please refer to the "*S5N8947 32-Bit RISC Microcontroller User's Manual*."

S5N8947 MEMORY MAP

Figure 3-1 shows a typical memory map for a boot ROM program written for the S5N8947 evaluation board. More information about each area of the memory map is provided below.

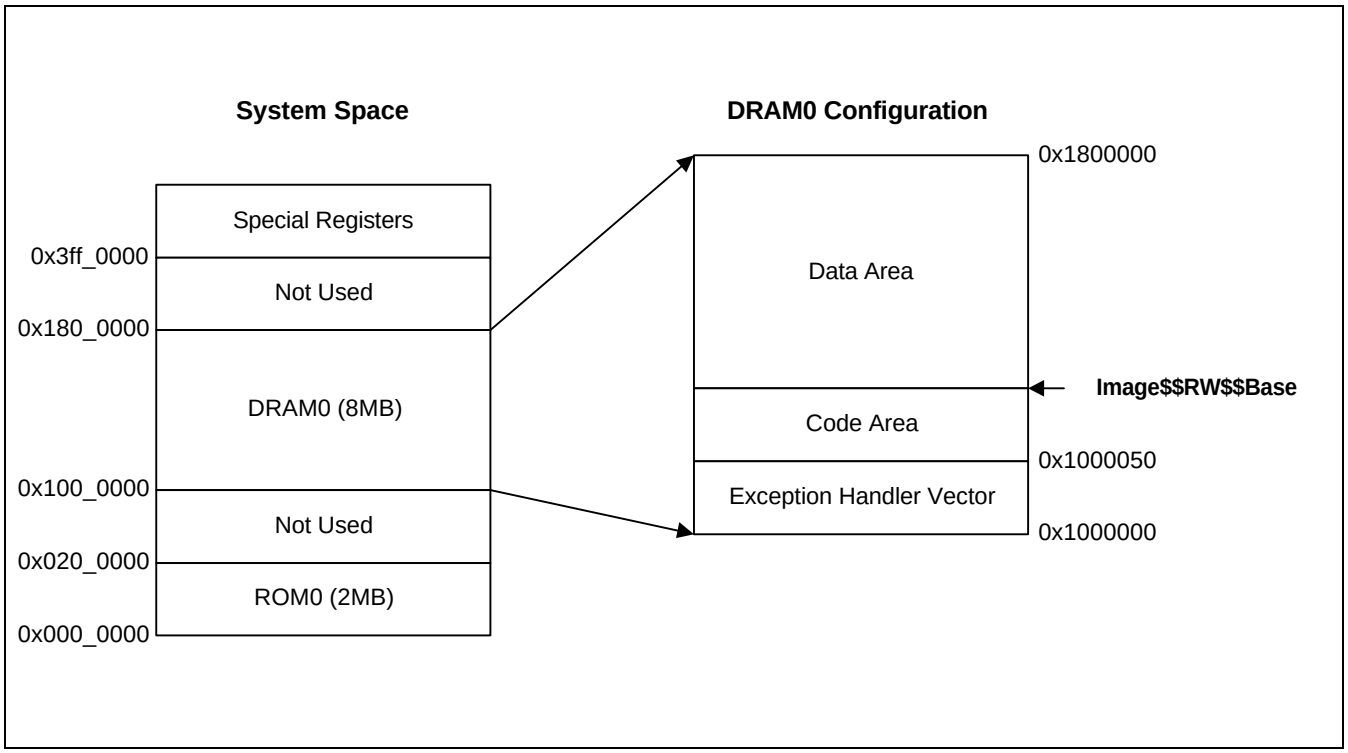


Figure 3-1. S5N8947 Memory Map

MEMORY MAPPED AREAS

SPECIAL REGISTERS

The boot ROM program defines and initializes each function block according to specified values. The special register area definition is controlled by the base address in the SYSCFG register, which indicates the start address of the special registers. The start address of individual special registers is defined as the special register base address (SYSCFG value) + the register offset value.

DRAM

The DRAM area contains a R/W (read/write) area, a R/O (read-only) area, and several user-defined areas, as follows:

- The *exception handler vector table* contains an exception handler vector address for each of the eight exception sources.
- The *data area* is used for global variables. Memory in this space can be freely allocated or released by the user application.
- The *code area* contains executable code which the user downloads from the host. This area is therefore meaningful only for program debugging.

ROM

The ROM area contains executable code and read-only data. You can use the ROM area for a boot ROM program that initializes the system environment for application development, as well as for standalone ROM code that contains complete execution codes and data.

EXAMPLE CODE FOR MEMORY AREA DEFINITION

All of the user-definable memory areas can be configured by the user program except for the code and data areas, which are defined by linker option. Several examples are provided below.

Define Data and Code Area Using the armlink Command or Windows Toolkit <makefile>

```
#-------------------------------------------------------------------------*
# Common rules for this BSP                                               *
#                                                                         *
# RAMOPTS  - Linker flags for location of ram.axf/ram.bin image           *
# ROMOPTS  - Linker flags for location of rom.bin image                   *
#-------------------------------------------------------------------------*

RAMOPTS         = -RO -Base 0x1000050
ROMOPTS         = -RO 0x0 -RW 0x1300000
```

Set Up the Special Register Area Using 0x3FF0000 as Special Register Base Address <init.s>

```
  LDR     r0, =0x3FF0000          ; Default Address of SYSCFG = 0x3FF0000
  LDR     r1, =0x83FF004          ; Base_addr = 0x3FF00000
  STR     r1, [r0]                ; Cache OFF, Write Buffer ON
```

Define an Exception Handler Area From 0x1000000 ~ 0x1000020 <board.a>

```
;*************************************************************************
;* Align Exception Handler Area for 8 exception sources.                 *
;* : 0x1000000 ~ 0x1000020                                               *
;*************************************************************************
EXTHND_BASE     EQU     0x1000000

                ^       EXTHND_BASE
HandleReset     #       4
HandleUndef     #       4
HandleSwi       #       4
HandlePrefetch  #       4
HandleAbort     #       4
HandleReserv    #       4
HandleIrq       #       4
HandleFiq       #       4
```

ABOUT <BEGIN.S>

C language programs often employ many variable of various types and scopes. Auto-variables use registers and the stack area, and global variables are allocated to memory area. When you link programs that use global variables, you must then copy these variables from the read-only area in the DRAM to the DRAM's read/write area. The ARM linker has several system variables that you can use to execute this operation:

- **Image$$RO$$Limit** Defines the end address of the ROM code area, which includes execution code and read-only data.
- **Image$$RW$$Base** Defines the RAM base address to be initialized. RAM is used for the read/write area. The global variables are grouped into two categories:

  Zero initialized area, with variables initialized to zero by the compiler.

  Initialized area, which contains variables initialized by the program.
- **Image$$ZI$$Base** Defines the base address of the Zero initialized area of the RAM.
- **Image$$ZI$$Limit** Defines an end address of the Zero initialized area of the RAM.

EXAMPLE CODE FOR <BEGIN.S>

The example code shown below uses the system variables described above to initialize the DRAM read/write area.

```
      IMPORT  HdwInit
      EXPORT  SysInitVars
      AREA    BEGIN,   CODE,   READONLY

      ALIGN
      ENTRY

;*********************************************************************
;* __main: Pseudo C entry point                                      *
;*                                                                   *
;* The C compiler genereates a reference to the symbol "__main"to ensure*
;* that the object module containing the entry point gets pulled in.    *
;*                                                                   *
;* This entry point is never actually called.                        *
;*********************************************************************
      EXPORT  main
main

;*********************************************************************
;* First instruction to be executed.                                 *
;*                                                                   *
;* Branch to the HdwInit entry point in the BSP.                     *
;*********************************************************************
        B           HdwInit

;*********************************************************************
;* The linker defines the following symbols which allow us to locate    *
;* the various sections (CODE, DATA, BSS) within the image.          *
;*                                                                   *
;*                            ROM Image                              *
;*                            +-----------------+                    *
;* Image$$RO$$Base       -> | CODE            |                    *
;*                            | ...             |                    *
;*                            +-----------------+ <- Image$$RO$$Limit     *
;*                            | Initialising    |                    *
;*                            | data ....       |                    *
;*                            +-----------------+                    *
;*                                                                   *
;*                            RAM Image                              *
;*                            +-----------------+                    *
;* Image$$RW$$Base       -> | Initialised     |                    *
;*                            | data            |                    *
;* Image$$ZI$$Base       -> +-----------------+ <- Image$$RW$$Limit     *
;*                            | Zero init (BSS) |                    *
;*                            | data            |                    *
;*                            +-----------------+ <- Image$$ZI$$Limit     *
;*                                                                   *
;*********************************************************************
        IMPORT  |Image$$ZI$$Base|
        IMPORT  |Image$$ZI$$Limit|
        IMPORT  |Image$$RO$$Limit|
        IMPORT  |Image$$RW$$Base|

;*********************************************************************
```

```
;* SysInitVars: Initialise the DATA and BSS sections.                    *
;*                                                                       *
;* The DATA section is initialised by copying data from the end of the   *
;* ROM image (given by Image$$RO$$Limit) to the start of the RAM image   *
;* (given by Image$$RW$$Base), stopping when we reach Image$$RW$$Limit.  *
;*                                                                       *
;* All data from Image$$RW$$Limit to Image$$ZI$$Limit is then            *
;* cleared to 0                                                          *
;************************************************************************
SysInitVars
;*-----------------------------------------------------------------------*
;* Load up the linker defined values for the static data copy            *
;*-----------------------------------------------------------------------*
        LDR     r0, =|Image$$RO$$Limit|
        LDR     r1, =|Image$$RW$$Base|
        LDR     r3, =|Image$$ZI$$Base|

;*-----------------------------------------------------------------------*
;* But first check whether we are trying to copy to the same address.    *
;* If so, this means that the image was linked as an application image   *
;* with the DATA section immediately following the CODE section.         *
;* Therefore there is nothing to copy since the data is already in place*
;*-----------------------------------------------------------------------*
        CMP     r0, r1
        BEQ     %1

;*-----------------------------------------------------------------------*
;* Stop on CS (ie R1 becomes >= R3). Carry has the same sense as 6502    *
;* (ie Carry = NOT Borrow).                                              *
;*-----------------------------------------------------------------------*
0       CMP     r1, r3
        LDRCC   r2, [r0], #4
        STRCC   r2, [r1], #4
        BCC     %0

;*-----------------------------------------------------------------------*
;* Clear remainder of data to Image$$ZI$$Limit to 0                      *
;*-----------------------------------------------------------------------*
1       LDR     r1, =|Image$$ZI$$Limit|
        MOV     r2, #0
2       CMP     r3, r1
        STRCC   r2, [r3], #4
        BCC     %2
        MOV     pc, lr

        END
```

NOTES

4 EXCEPTION HANDLING

This chapter explains how exceptions are handled. It also tells you how to set up an exception vector table and how to write an exception handler routine. Information is presented according to the following Table of Contents:

INTRODUCTION

An exceptions occurs when the normal flow of execution through a user program is diverted to allow the processor to handle events generated by internal or external sources. Two examples of such events are:

- Externally generated interrupts
- An attempt by the processor to execute an undefined instruction

When handling exceptions, the previous processor status must be preserved so that the execution of the original user program can resume immediately after the appropriate execution routine is completed.

The ARM processor recognizes seven types of exceptions:

Reset	Occurs when the CPU reset pin is asserted. This exception normally occurs to signal a power-up, or to initiate a reset following CPU power-up. It is therefore useful for initiating a "soft" reset.
Undefined Instruction	Occurs if neither the CPU nor any attached coprocessor recognizes the instruction currently being executed.
Software Interrupt	A user-defined synchronous interrupt instruction which allows a program running in User mode to request privileged operations that run in Supervisor mode.
Pre-fetch Abort	Occurs when the CPU attempts to execute an instruction which has been pre-fetched from an illegal address. In this case, an illegal address is an address that the memory management subsystem has determined is inaccessible to the CPU in its current mode.
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
IRQ	Occurs when the CPU's external interrupt request pin is asserted (Low) and the I bit in the CPSR is clear.
FIQ	Occurs when the CPU's external fast interrupt request pin is asserted (Low) and the F bit in the CPSR is clear.

EXCEPTION HANDLING

THE EXCEPTION VECTOR TABLE

The ARM processor supports seven types of exceptions. Each type is assigned a privileged processor mode. When an exception occurs, program execution is forced from a fixed memory address corresponding to the exception type. These fixed addresses are called “hard vectors.”

Exception handling is controlled by a vector table. The vector table is a reserved 32-byte area at the bottom of the memory map with one word of space allocated to each exception type. Because there is not enough space to contain the complete code for an exception handler, the vector entry for each exception type typically contains a branch or load PC instruction that passes control to the appropriate handler routine.

Table 4-1 lists the exception types, the corresponding processor modes, and the “hard” vector addresses for each type.

Table 4-1. Exception Processing Modes and Vectors

Exception Type	Mode	Vector Address
Reset	SVC	0x00000000
Undefined Instruction	UNDEF	0x00000004
Software Interrupt (SWI)	SVC	0x00000008
Pre-fetch Abort	ABORT	0x0000000c
Data Abort	ABORT	0x00000010
IRQ	IRQ	0x00000018
FIQ	FIQ	0x0000001c

ENTERING AN EXCEPTION

When an exception is generated, the processor executes the following steps. (*Again, these operations are performed by the processor, not by a user program)*:

1. SPSR_<exception_mode> := CPSR

   Copy the contents of the Current Status Register (CPSR) into the Saved Program Status Register (SPSR) of the mode in which the execution is to be handled. This saves the current mode, interrupt masks, and condition flags.

2. LR_<exception_mode> := PC - 4

   Store the return address (pc-4) in lr_<exception_mode>.

3. PC := Vector address_ <exception_mode>

   Set the PC to the appropriate vector address. That is, execution control is forced to an exception handler whose entry address is hard-wired to the lower memory area.

4. The processor then sets the appropriate CPSR mode bits:

   CPSR[5:0] := Exception mode number

   To change to the appropriate mode, also mapping in the appropriate banked registers for that mode.

   CPSR[6] := 1 (if the exception mode is RESET or FIQ), or

   CPSR[7] := 1 (to disable interrupts)

   Note that IRQs are disabled when any other type of exception occurs and FIQs are disabled whenever an FIQ exception occurs.

LEAVING AN EXCEPTION

To return control of program execution to the place where the exception occurred, the handler must perform the following steps. *(These steps must be performed by a user program)*:

1. CPSR := SPSR_<exception_mode>
2. PC := LR_<exception_mode>

The handler performs these two operations as an atomic instruction by performing a data processing instruction with the S flag set, and with the PC as the destination register:

```
MOVS    pc, lr
```

Each exception type requires the use of a different instruction. The actual value that is stored in the PC is also dependent on the exception type, as follows:

Undefined Instruction	MOVS	pc, lr
SWI	MOVS	pc, lr
Pre-fetch Abort	SUBS	pc, lr, #4
Data Abort	SUBS	pc, lr, #8
IRQ	SUBS	pc, lr, #4
FIQ	SUBS	pc, lr, #4

SAMPLE EXCEPTION HANDLER

The basic form of an exception handler is as follows:

```
Default_HandleException
        <Save value of registers to the stack area>
        <Exception service routine performed by user application>
        <Restore the values in the stack area to the corresponding registers>
        <Return the PC value according to exception type>
```

INSTALLING AN EXCEPTION HANDLER

You must install any new exception handler in the vector table. Once the installation is complete, the new handler executes wherever the corresponding exception occurs.

Installing the Handlers at Reset

Exception handlers can be included in a boot program or they can be standalone programs. Boot programs are used to initialize the system environment for application development. Standalone programs, on the other hand, do not rely on the debugger or debug monitor to start program execution. This makes it possible to load the vector table directly as part of your assembler reset (or startup) code. If your ROM area starts at memory location 0x0, you can use a simple branch statement for each vector at the start of your program.

Example Handler Installation Routine

S5N8947 boot ROM programs and standalone ROM programs must include an exception installation routine as part of the startup code. In the example below, only the program block related to FIQ exception handling is shown. For more information and examples for boot ROM programs, see Chapter 5 of this manual.

```
;// [ header file : board.a ]

          .
          .
          .

;*************************************************************************
;* Align Exception Handler Area for 8 exception sources.                 *
;* : 0x1000000 ~ 0x1000020                                               *
;*************************************************************************
EXTHND_BASE     EQU     0x1000000

                ^       EXTHND_BASE
HandleReset     #       4
HandleUndef     #       4
HandleSwi       #       4
HandlePrefetch  #       4
HandleAbort     #       4
HandleReserv    #       4
HandleIrq       #       4
HandleFiq       #       4
```

```
;// [ S5N8947 Boot Program : rombegin.s ]
GET     board.a

        IMPORT  RomHdwInit
        IMPORT  SetVector
        EXPORT  SysInitVars

;*************************************************************************
;* Entry point of RAM based project.                                     *
;*************************************************************************
        AREA    ROMBEGIN,   CODE,   READONLY

        ENTRY
;*************************************************************************
; __main: Pseudo C entry point
;
; The C compiler genereates a reference to the symbol "__main" to ensure
; that the object module containing the entry point gets pulled in.
;
; This entry point is never actually called.
;*************************************************************************
                    EXPORT  main
main
                    B                   RomHdwInit
                    B                   Undefined_Handler
                    B                   SWI_Handler
                    B                   Prefetch_Handler
                    B                   Abort_Handler
                    B                   RomHdwInit
                    B                   IRQ_Handler
                    B                   FIQ_Handler
;=========================================================
; The Default Exception Handler Vector Entry Pointer Setup
;=========================================================
FIQ_Handler
        SUB     sp, sp, #4
        STMFD   sp!,{r0}
        LDR     r0, =HandleFiq
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!,{r0, pc}

IRQ_Handler
        SUB     sp, sp, #4
        STMFD   sp!,{r0}
        LDR     r0, =HandleIrq
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!,{r0, pc}


Prefetch_Handler
        SUB     sp, sp, #4
        STMFD   sp!,{r0}
        LDR     r0, =HandlePrefetch
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!,{r0, pc}
```

```
Abort_Handler
        SUB     sp, sp, #4
        STMFD   sp!,{r0}
        LDR     r0, =HandleAbort
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!,{r0, pc}

Undefined_Handler
        SUB     sp, sp, #4
        STMFD   sp!,{r0}
        LDR     r0, =HandleUndef
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!,{r0, pc}

SWI_Handler
        SUB     sp, sp, #4
        STMFD   sp!, {r0}
        LDR     r0, =HandleSwi
        LDR     r0, [r0]
        STR     r0, [sp, #4]
        LDMFD   sp!, {r0, pc}

        IMPORT  |Image$$ZI$$Base|
        IMPORT  |Image$$ZI$$Limit|
        IMPORT  |Image$$RO$$Limit|
        IMPORT  |Image$$RW$$Base|
```

```
;*******************************************************************
; SysInitVars: Initialise the DATA and BSS sections.
;
; The DATA section is initialised by copying data from the end of the
; ROM image (given by Image$$RO$$Limit) to the start of the RAM image
; (given by Image$$RW$$Base), stopping when we reach Image$$RW$$Limit.
;
; All data from Image$$RW$$Limit to Image$$ZI$$Limit is then cleared to 0
;*******************************************************************
SysInitVars
;-------------------------------------------------------------------
; Load up the linker defined values for the static data copy
;-------------------------------------------------------------------
        LDR     r0, =|Image$$RO$$Limit|
        LDR     r1, =|Image$$RW$$Base|
        LDR     r3, =|Image$$ZI$$Base|
;-------------------------------------------------------------------
; But first check whether we are trying to copy to the same address.
; If so, this means that the image was linked as an application image
; with the DATA section immediately following the CODE section. Therefore
; there is nothing to copy since the data is already in place.
;-------------------------------------------------------------------
        CMP     r0, r1
        BEQ     %1

;-------------------------------------------------------------------
; Stop on CS (ie R1 becomes >= R3). Carry has the same sense as 6502
; (ie Carry = NOT Borrow).
;-------------------------------------------------------------------
0       CMP     r1, r3
        LDRCC   r2, [r0], #4
        STRCC   r2, [r1], #4
        BCC     %0

;-------------------------------------------------------------------
; Clear remainder of data to Image$$ZI$$Limit to 0
;-------------------------------------------------------------------
1       LDR     r1, =|Image$$ZI$$Limit|
        MOV     r2, #0
2       CMP     r3, r1
        STRCC   r2, [r3], #4
        BCC     %2
        MOV     pc, lr

Mode_Mask       EQU     3
IRQ_Mask        EQU     2


        END
```

5 CONTROLLING S5N8947 MODULES

This chapter presents concept diagrams and example programs for configuring various S5N8947 system control functions. Information is presented according to the following Table of Contents:

S5N8947 PROGRAMMER'S MODEL

The diagnostic code routines have been written to give practical examples of writing code and evaluating S5N8947 evaluation board. This is the target board of the S5N8947, which is embedded controller for network solutions based on ARM7TDMI.

This section will give you a brief descriptions about how to control the embedded functional blocks and also how to evaluate the S5N8947 Evaluation board.

HARDWARE OVERVIEW

- BOOT ROM : 8bit data bus, 4Mbit (512Kbytes) EEPROM, Located at ROM Bank0
- DRAM TYPE : Normal/EDO DRAM (8Mbytes) for S5N8947. DRAM Bank0,1,2,3 used.
- CONSOLE : UART (SIO) used.
- ETHERNET : RJ45 connector, 7-Wire interface or MII, 2-channel, 10/100Mbps.
- JTAG : Embedded ICE or Emulator can be interfaced with this for system debugging.

SYSTEM MEMORY MAP

The CM47-M66-V1.0 board has a ROM socket which can be used to boot ROM.

For the code development, S5N8947 Evaluation board supports SyncDRAM into component type and also one DRAM SIMM sockets. So you can select DRAM SIMM or SyncDRAM by jumper on the board.

The S5N8947 has a total 16M word memory space. Each memory banks can be located anywhere within a this address range by setup the appropriate memory control registers. The data bus size of each bank also can be configured by bus control registers.

The system configuration register (SYSCFG) also used to configure the start address of the special register, and also control the write buffer, cache. The special register’s address area is fixed at 64Kbytes. It’s initial value is 0x23FF0000.

You can use the 8-Kbyte Cache memory. Please refer to user’s manual for more details. For the Direct Memory Access, you have to configure this area as non-cachable region as set the bit [26] of memory access address.

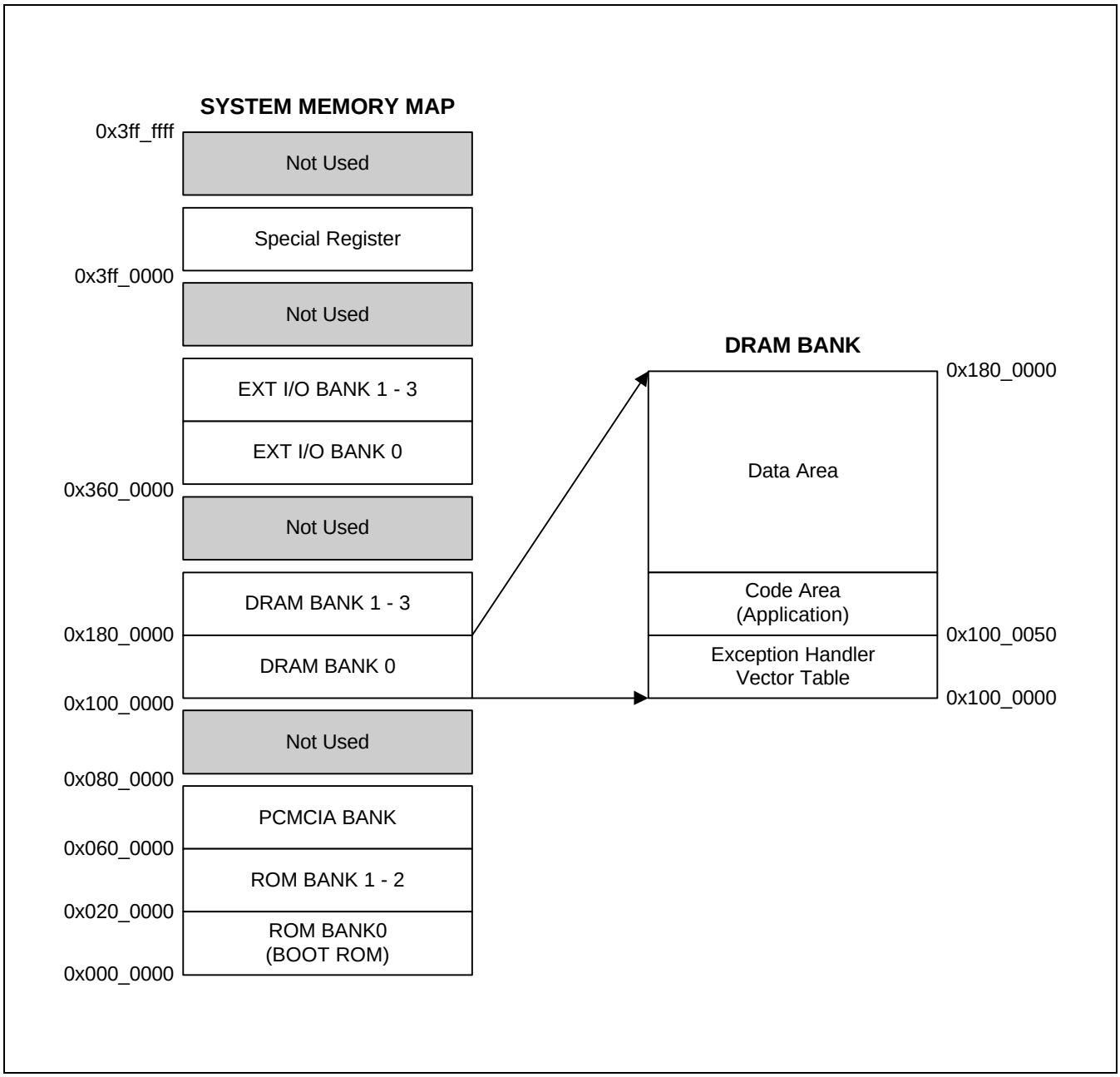


Figure 5-1. S5N8947 System Memory Map (for Sample Program)

SYSTEM MANAGER

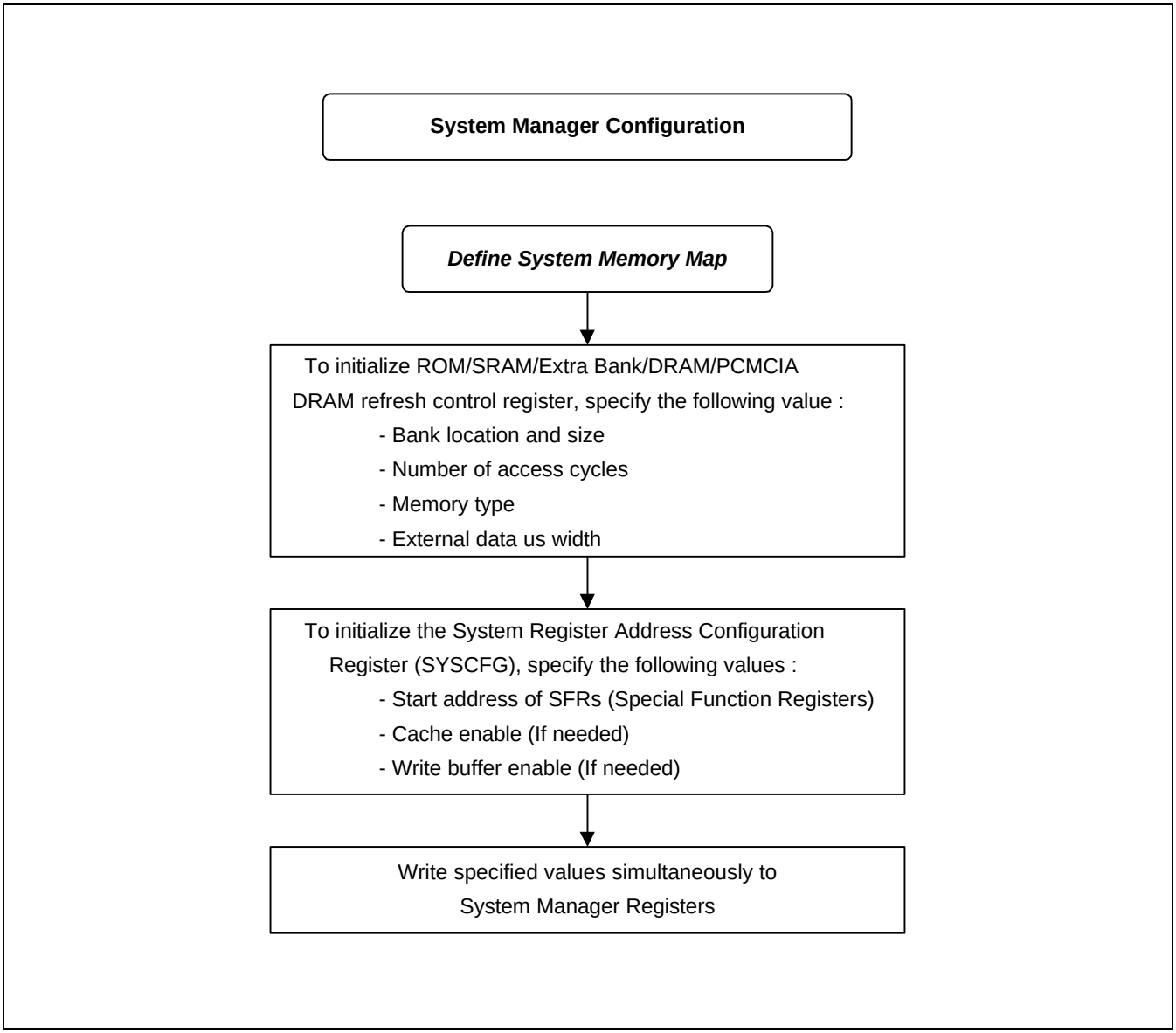


Figure 5-2. System Manager Concept Diagram

EXAMPLE FOR SYSTEM MANAGER CONFIGURATION PROGRAM

```
;/--------------------------------------------------------------------/
;/ SDRAM setting : S5N8947 only.                                      /
;/--------------------------------------------------------------------/
SYNC_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x83FF0004         ; Start_addr = 0x3FF00000
        STR     r1, [r0]                ; Cache OFF, Write Buffer ON

    ;/--------------------------------------------------------------------/
    ;/ Initialize Memory Configuration.                                   /
    ;/ This operation must be done at a time.                            /
    ;/--------------------------------------------------------------------/
        ADRL    r0, SystemInitData_SDRAM
        LDMIA   r0, {r1-r10}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCON Offset : 0x3010
        STMIA   r0, {r1-r10}

    ;/--------------------------------------------------------------------/
    ;/ Test whether SDRAM is used or not.                                 /
    ;/--------------------------------------------------------------------/
        LDR     r1, =EXTHND_BASE
        STR     r1,[r1]                 ; write to SDRAM base
        LDR     r2,[r1]                 ; read from SDRAM base
        CMP     r2,r1                   ; compare above two data
        BEQ     SetSVC                  ; if equal, branch to SetSVC
                                        ; else, set SDRAM environment

;/--------------------------------------------------------------------/
;/ EDO DRAM setting :                                                 /
;/--------------------------------------------------------------------/
EDO_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x23FF0004          ; Start_addr = 0x3FF00000
        STR     r1, [r0]                ; Cache OFF, Write Buffer ON

    ;/--------------------------------------------------------------------/
    ;/ Initialize Memory Configuration.                                   /
    ;/ This operation must be done at a time.                            /
    ;/--------------------------------------------------------------------/
        ADRL    r0, SystemInitData_EDO
        LDMIA   r0, {r1-r10}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCON Offset : 0x3010
        STMIA   r0, {r1-r10}
;// Additional code:
          •
          •
```

```
;/--------------------------------------------------------------------/
;/ System Memory Initialization Data :                                /
;/ For more, refer to the S5N8947 User's Manual                       /
;/--------------------------------------------------------------------/
SystemInitData_EDO
        DCD     0x0FFFFFFa          ; 32bit data bus...     (EXTDBWTH)
        DCD     0x02000060          ; 0x0000000 ~ 0x01FFFFF (ROMCON0)
        DCD     0x04008040          ; 0x0200000 ~ 0x03FFFFF (ROMCON1)
        DCD     0x06010040          ; 0x0400000 ~ 0x05FFFFF (ROMCON2)
        DCD     0x0                 ; 0x0600000 ~ 0x07FFFFF (PCMOFFSET)
        DCD     0x9804039b          ; 0x1000000 ~ 0x17FFFFF (DRAMCON0)
        DCD     0x0                 ; 0x1800000 ~ 0x1BFFFFF (DRAMCON1)
        DCD     0x9C0601AC          ; 0x1800000 ~ 0x1BFFFFF (DRAMCON2)
        DCD     0xA00701AC          ; 0x1C00000 ~ 0x1FFFFFF (DRAMCON3)
        DCD     0x9c398360          ; Refresh enable...     (REFEXTCON)

SystemInitData_SDRAM
        DCD     0x0FFFFFFa          ; 32bit data bus...     (EXTDBWTH)
        DCD     0x02000060          ; 0x0000000 ~ 0x01FFFFF (ROMCON0)
        DCD     0x04008040          ; 0x0200000 ~ 0x03FFFFF (ROMCON1)
        DCD     0x06010040          ; 0x0400000 ~ 0x05FFFFF (ROMCON2)
        DCD     0x0                 ; 0x0600000 ~ 0x07FFFFF (PCMOFFSET)
        DCD     0x18040380          ; 0x1000000 ~ 0x17FFFFF (DRAMCON0)
        DCD     0                   ; 0x1800000 ~ 0x1BFFFFF (DRAMCON1)
        DCD     0x1c060180          ; 0x1800000 ~ 0x1BFFFFF (DRAMCON2)
        DCD     0x20070180          ; 0x1C00000 ~ 0x1FFFFFF (DRAMCON3)
        DCD     0xce278360          ; Refresh enable...     (REFEXTCON)
```

INTERRUPTS

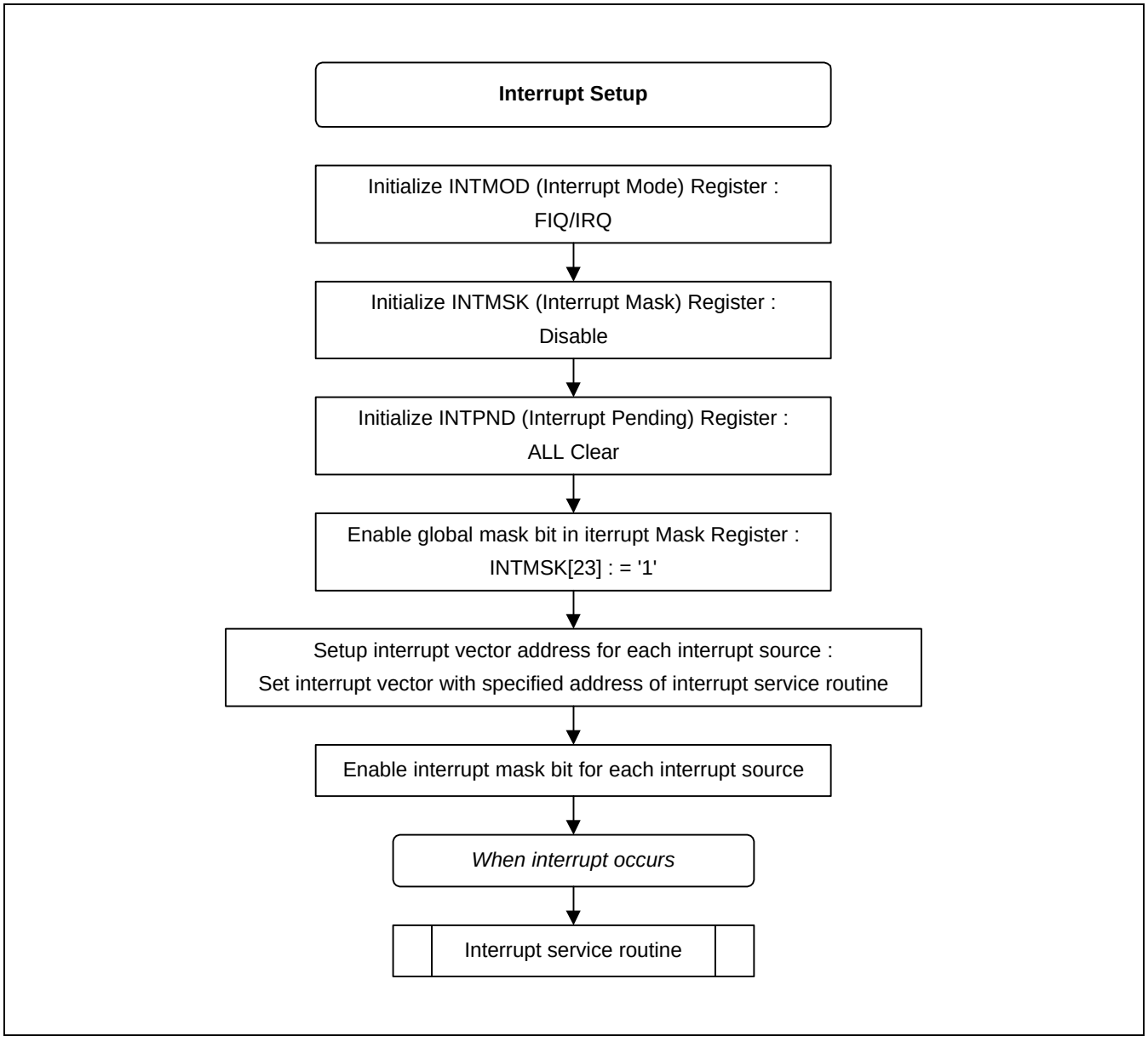


Figure 5-3. Interrupt Handler Setup Concept Diagram

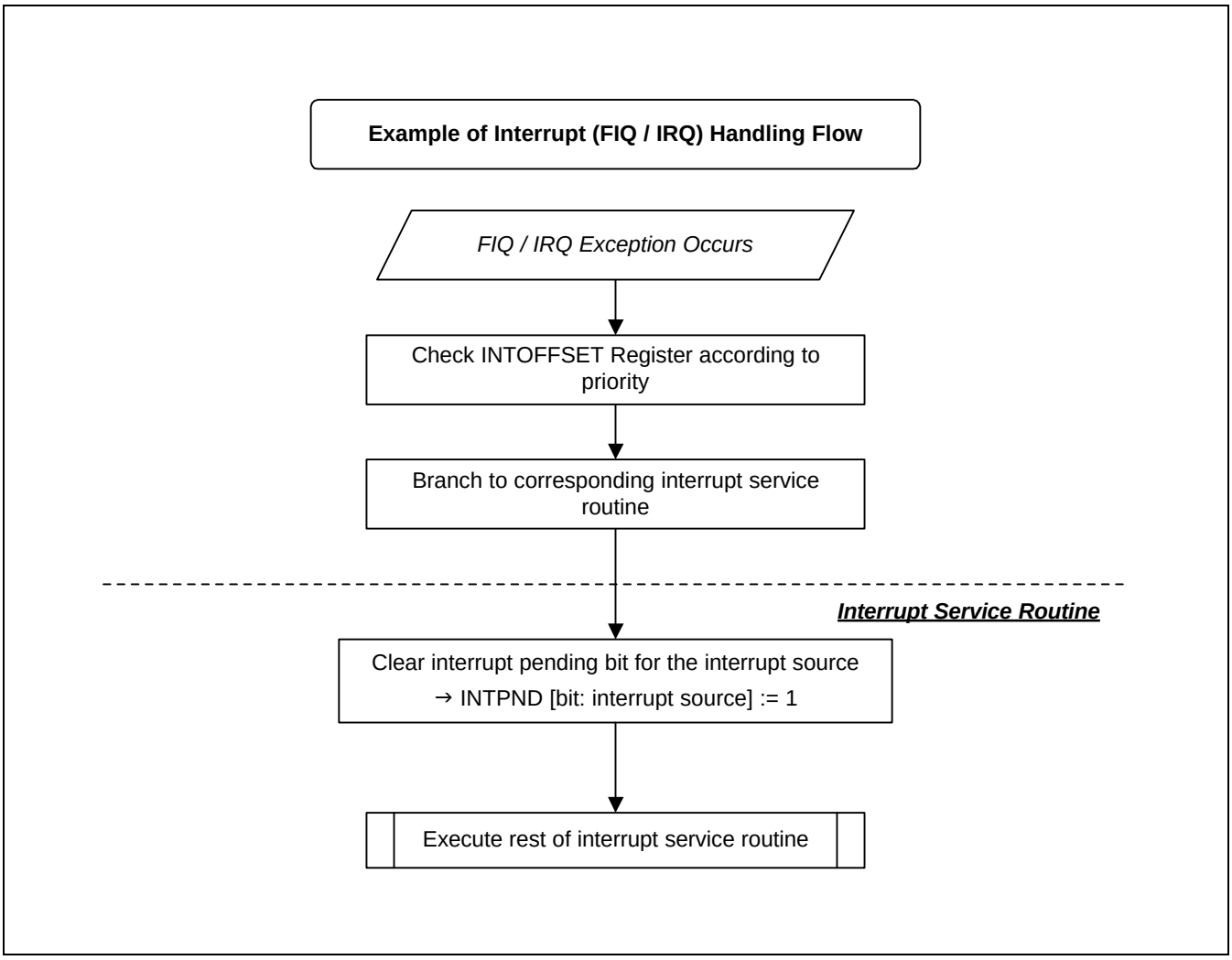


Figure 5-4. Interrupt Handler Concept Diagram

EXAMPLE CODE FOR INTERRUPT HANDLER ROUTINE

```
; IRQ/FIQ Handler  <except.s>
;/-------------------------------------------------------------------------/
;/ Wrapper : IRQ exception.                                                 /
;/-------------------------------------------------------------------------/
IRQWrapper
        STMFD   sp!, {r0-r12, lr}
        LDR     r1, =INTOFFSET
        LDR     r0, [r1]
        BL      MainIntHandler
        LDMFD   sp!, {r0-r12, lr}
        SUBS    pc, lr, #4

;/-------------------------------------------------------------------------/
;/ Wrapper : FIQ exception.                                                 /
;/-------------------------------------------------------------------------/
FIQWrapper
        STMFD   sp!, {r0-r7, lr}
        LDR     r1, =INTOFFSET
        LDR     r0, [r1]
        BL      MainIntHandler
        LDMFD   sp!, {r0-r7, lr}
        SUBS    pc, lr, #4

// Main Interrupt Handler <Intrhndl.c>
/*************************************************************************/
/* MainIntHandler: main interrupt handler                                */
/*                                                                       */
/*      INPUTS: vector: vector offset of interrupt                       */
/*       NOTES: FIQ/IRQ wrappers call this routine to handle all         */
/*              interrupts.                                              */
/*                                                                       */
/*************************************************************************/

void MainIntHandler(int offset)
{
    // Clear pending bit in the INTPEND register.
    Clear_PendingBit(offset>>2) ;

    // Call interrupt service routine.
    (*InterruptHandlers[offset>>2])();
}
```

EXAMPLE FOR INTERRUPT SETUP ROUTINE - HEADER FILE <Isr.h>

```
#include "board.h"

//--------------------------------------------------------------------------//
// The Bit value of Interrupt registers.                                    //
//--------------------------------------------------------------------------//
#define   EXT0_INT            0x000001
#define   EXT1_INT            0x000002
#define   EXT2_INT            0x000004
#define   EXT3_INT            0x000008
#define   EXT4_INT            0x000010
#define   EXT5_INT            0x000020
#define   EXT6_INT            0x000040

#define   UART0_TX_INT        0x000080
#define   UART0_RX_ERR_INT    0x000100
#define   TIMER0_INT          0x000200
#define   TIMER1_INT          0x000400
#define   TIMER2_INT          0x000800

#define   GDMA0_INT           0x001000
#define   GDMA1_INT           0x002000

#define   USB_INT             0x004000

#define   SAR_ERROR_INT       0x008000
#define   SAR_DONE_INT        0x010000

#define   MAC0_TX_INT         0x020000
#define   BDMA0_RX_INT        0x040000

#define   MAC1_TX_INT         0x080000
#define   BDMA1_RX_INT        0x100000

#define   IIC_INT             0x200000
#define   SPI_INT             0x400000

#define   GLOBAL_INT          0x800000
```

```
//--------------------------------------------------------------------------//
// Number of Interrupt sources.                                             //
//--------------------------------------------------------------------------//
#define   nEXT0_INT           0
#define   nEXT1_INT           1
#define   nEXT2_INT           2
#define   nEXT3_INT           3
#define   nEXT4_INT           4
#define   nEXT5_INT           5
#define   nEXT6_INT           6


#define   nUART0_TX_INT       7
#define   nUART0_RX_ERR_INT   8

#define   nTIMER0_INT         9
#define   nTIMER1_INT         10
#define   Ntimer2_INT         11
#define   nGDMA0_INT          12
#define   nGDMA1_INT          13

#define   nUSB_INT            14
#define   nSAR_ERROR_INT      15
#define   nSAR_DONE_INT       16

#define   nMAC0_TX_INT        17
#define   nBDMA0_RX_INT       18
#define   nMAC1_TX_INT        19
#define   nBDMA1_RX_INT       20
#define   nIIC_INT            21
#define   nSPI_INT            22
#define   nGLOBAL_INT         23

//--------------------------------------------------------------------------//
// Interrupt Macro Funtions.                                                //
//--------------------------------------------------------------------------//
#define Enable_Intr(n)      INTMASK   &= ~(1<<(n))
#define Disable_Intr(n)     INTMASK   |= (1<<(n))
#define Clear_PendingBit(n) INTPEND   =  (1<<(n))
#define SetPendingBit(n)    INTPNDTST =  (1<<(n))

//--------------------------------------------------------------------------//
// C funtion defines.                                                       //
//--------------------------------------------------------------------------//
extern void InitIntHandlerTable(void);
extern void SysSetInterrupt(unsigned long , void (*)());
extern void MainIntHandler(int);
extern void MainUndefHandler(void);
extern void MainPrefetchHandler(void);
extern void MainAbortHandler(void);
```

EXAMPLE INTERRUPT SETUP ROUTINE <Intrhndl.c>

```
#include "board.h"
#include "isr.h"

void (*InterruptHandlers[MAXHNDLRS])(void);

static void DummyIsr()
{

}

/*************************************************************************/
/* InitIntHandlerTable: Initialize the interrupt handler table         */
/*                                                                       */
/*     NOTE(S): This should be called during system initialization     */
/*              by HdwInit.                                              */
/*                                                                       */
/*************************************************************************/
void InitIntHandlerTable(void)
{
    int i;

    // Initialize interrupt handler table to dummy function.
    for (i = 0; i < MAXHNDLRS; i++)
        InterruptHandlers[i] = DummyIsr;
}

/*************************************************************************/
/* SysSetInterrupt: Set the interrupt handler.                          */
/*************************************************************************/
void SysSetInterrupt(unsigned long vector, void (*handler)())
{
    // Initialize each interrupt handler to user defined function.
    InterruptHandlers[vector] = handler;
}
```

UART

S5N8947 has one UART Channel. To setup the UART Channel, follow the steps in figure 5-5 below. After finishing setting up, with the concept diagram for using UART in figure 5-6, you can use UART as a communication method between host and target system.

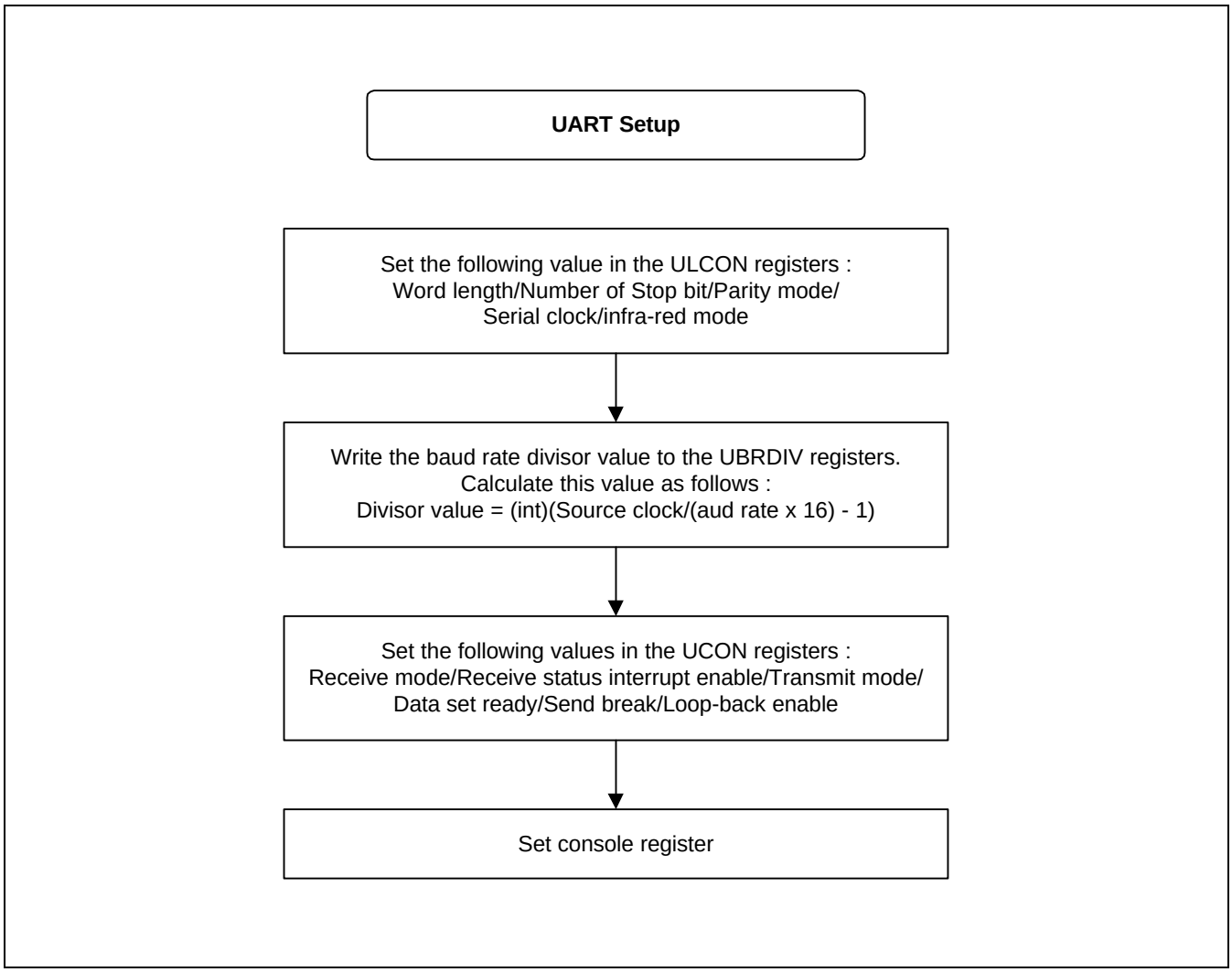


Figure 5-5. Concept Diagram for UART Initialization

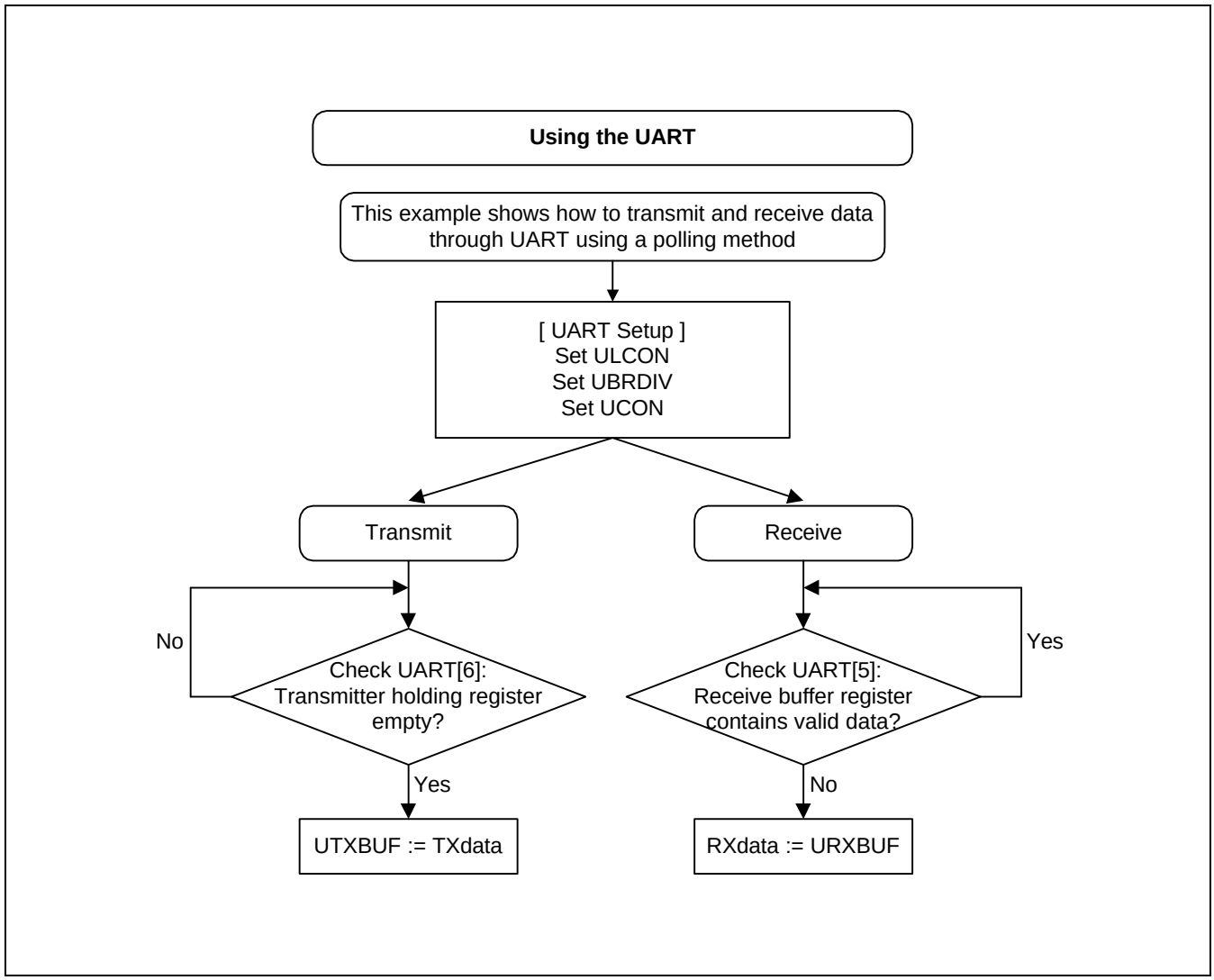


Figure 5-6. Concept Diagram for Using the UART

EXAMPLE CODE FOR UART SETUP ROUTINE <UART.c>

```
//=====================================================================//
// UART_Setup: Perform startup initialization of serial channels       //
//                                                                     //
//    ChanNum:  Indicates the serial channel to be opened.             //
//    Notice :  S5N8947 has only 1 UART Channel                        //
//                                                                     //
//=====================================================================//
void UART_Setup(int ChanNum)
{
    unsigned int WordLength=0;
    unsigned int tDIV, DivCnt0, DivCnt1, DivCntVal;

    WordLength = ULCON_WL8;

    // Set Buad Rate Divisor value.
    // You should select UART clock, defined in the file "uart.h".
    tDIV = UARTCLK / (16 * _BaudRate);
    if(((tDIV%16)==0) || ((tDIV-1)>0xffff)) {
        DivCnt1 = 1;
        DivCnt0 = tDIV/16 -1;
    }
    else  {
        DivCnt1 = 0;
        DivCnt0 = tDIV -1;
    }
    DivCntVal = DivCnt0*16 + DivCnt1;

    // Initialize the UART registers.
    if(ChanNum == chUART_0) {
    #if(UARTCLK==MAINCLK)
        UARTLCON0 = WordLength; // Added 11.2
    #else
        UARTLCON0 = WordLength | UCLK ; // 11.2
    #endif
        UARTCONT0 = (UCON_RXM_INTREQ|UCON_TXM_INTREQ | UCON_RXSTAT_INT);
        UARTBRD0  = DivCntVal;
    }
    else {
        UARTLCON1 = WordLength | UCLK;;
        UARTCONT1 = (UCON_RXM_INTREQ|UCON_TXM_INTREQ|UCON_RXSTAT_INT);
        UARTBRD1  = DivCntVal;
    }

    // Set Console registers.
    if(chCONSOLE == chUART_0)  {
        CNSL_TxBuf = &UARTTXH0;
        CNSL_RxBuf = &UARTRXB0;
        CNSL_STAT  = &UARTSTAT0;
    }
    else   {
        CNSL_TxBuf = &UARTTXH1;
        CNSL_RxBuf = &UARTRXB1;
        CNSL_STAT  = &UARTSTAT1;
    }
}
```

```
// ================== [ General Function for Polling ] ==================== //

// Transmit one byte through UART...
void putb(char ch )
{
// Wait until UART0 Transmit buffer register is empty...and.
// when empty, tramsmit new data...
          while(ChkTxStatus(0))  ;
          UARTTxH = ch;
}

// Transmit string through UART...
void PutString(char *ptr )
{
          while(*ptr )
                    Putb(*ptr++ );
}

// Receive one byte through UART...
char getch(void)
{
          char ch;

// Wait until UART0 Receive buffer register has valid data...
// When data is valid, receive new data...
          while(!ChkRxStatus(0));
          ch = UARTRxB;

          return ch;
}
```

UART TEST PROGRAM DESCRIPTION IN DIAGNOSTIC ROM

S5N8947 Diagnostic program contains a program offers UART function test and shows the values of UART specific registers. For the detail of these functions, refer to timer diagnostic source code provided, ***"uart.c".***

<Main Menu>

```
=======================================================
            CM47-M66-V1.0 Board Diagnostic Ver 1.0
=======================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=======================================================
Select One... :
```

Selecting '[2] UART test' menu shows sub-menu of UART test below.

```
================================================
                UART TEST MENU
================================================
   [1] View Current UART Configuration
   [2] Change UART Configuration
   [3] Set Baud Rate
   [4] UART Tx Interrupt Test
   [5] UART Rx Interuupt Echo Test
   [Q] EXIT UART Test
================================================
Select One...
```

Descriptions of each menu are below

- View Current UART Configuration : shows values of UART specific registers.
- Change UART Configuration : change the values of UART registers.
- Set Baud Rate : changes UART baud rate.
- UART Tx Interrupt Test : tests UART Tx Interrupt.
- UART Rx Interrupt Echo Test : UART echo test by Interrupt method.
- EXIT UART Test : returns to main menu.

TIMERS

S5N8947 has three timers. To setup and using timers, you must control several registers described in figure 5-7 below. To run timers, you only have to set timer enable bit in the TMOD register.

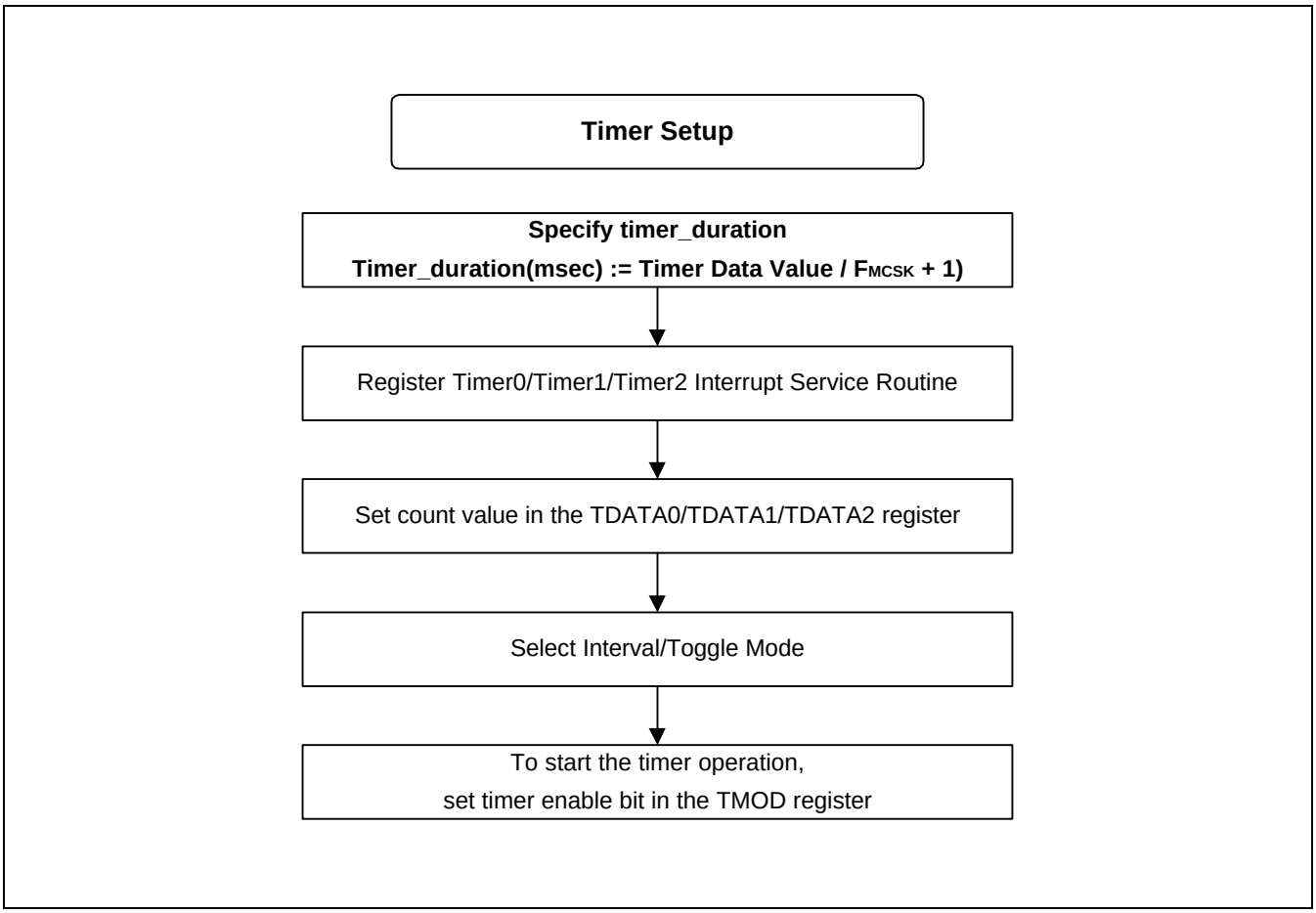


Figure 5-7. Concept Diagram for Setting a Timer

EXAMPLE CODE FOR SETTING A TIMER <Timer.c>

```
//---------------------------------------------------------------------//
// RunTimers : Sets timer registers values & Interrupts,               //
//             and runs the timers                                     //
//---------------------------------------------------------------------//
void RunTimers(unsigned int msec_t0, unsigned int msec_t1, unsigned int msec_t2)
{
      SysSetInterrupt(nTIMER0_INT, isr_TestTimer0);
      SysSetInterrupt(nTIMER1_INT, isr_TestTimer1);
      SysSetInterrupt(nTIMER2_INT, isr_TestTimer2);

      Enable_Intr(nTIMER0_INT);
      Enable_Intr(nTIMER1_INT);
      Enable_Intr(nTIMER2_INT);

      TDATA0 = (msec_t0 * 50000 - 1); // set timer0
      TDATA1 = (msec_t1 * 50000 - 1); // set timer1
      TDATA2 = (msec_t2 * 50000 - 1); // set timer2

      // Run Timers interval mode
      TMOD |= (Timer0_Run | Timer1_Run | Timer2_Run);
}

//---------------------------------------------------------------------//
// Timer0/1/2 Interrupt Service Routine                                //
//---------------------------------------------------------------------//
void isr_timer0(void)
{
    #if(DebugFlag_LED == YES)
    tm_cnt++;
    IOPDATA = ~(1 << (tm_cnt%8) );
    #endif

    //-----------------------------------------------------------------//
    // User codes needs here.                                          //
    //-----------------------------------------------------------------//
}

void isr_timer1(void)
{
      #if(DebugFlag_Timer == YES)
      Print("\nTimer1 Expired!!");
      #endif
    //-----------------------------------------------------------------//
    // User codes needs here.                                          //
    //-----------------------------------------------------------------//
}
void isr_timer2(void)
{
      #if(DebugFlag_Timer == YES)
      Print("\nTimer2 Expired!!");
      #endif
    //-----------------------------------------------------------------//
    // User codes needs here.                                          //
    //-----------------------------------------------------------------//
}
```

TIMER TEST PROGRAM DESCRIPTION IN DIAGNOSTIC ROM

S5N8947 Diagnostic program contains a test program for timers. You can test basic functions of timers and check the values of timer specific registers. For the details of these functions, refer to timer diagnostic source code provided, ***"timer.c".***

<Main Menu>

```
=========================================================
            CM47-M66-V1.0 Board Diagnostic Ver 1.0
=========================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=========================================================

Select One... :
```

Selecting '[3] Timer test' menu shows sub-menu of timer test below.

```
---------------------------------
        Timer Test Menu
---------------------------------
     [1] Run Timers
     [2] Test TOUT0/1/2
     [3] Test Watch-dog Timer
     [4] View Timer Configuration
     [Q] Exit Timer Test
---------------------------------
  Select One..
```

Descriptions of each menu are below:

- Run Timers : runs timers as pre-defined timer duration values and executes timer interrupt service routine. It's set for timer0 interrupt to occur every 1 sec and 2 sec for timer1.
- Test TOUT0/1/2 : test TOUT0/1/2 as outputs of timers through IO Port.
- Test Watch-dog Timer : gets the counting values and test Watch-dog timer.
- View Timer Configuration : shows values of timer specific registers.
- Exit Timer Test : returns to main menu.

GDMA

The S5N8947 has a two-channel general DMA controller. After finishing setting up, you can use GDMA as a communication method with memory. To setup and using GDMAs, you must control registers described in figure 5-8 below.

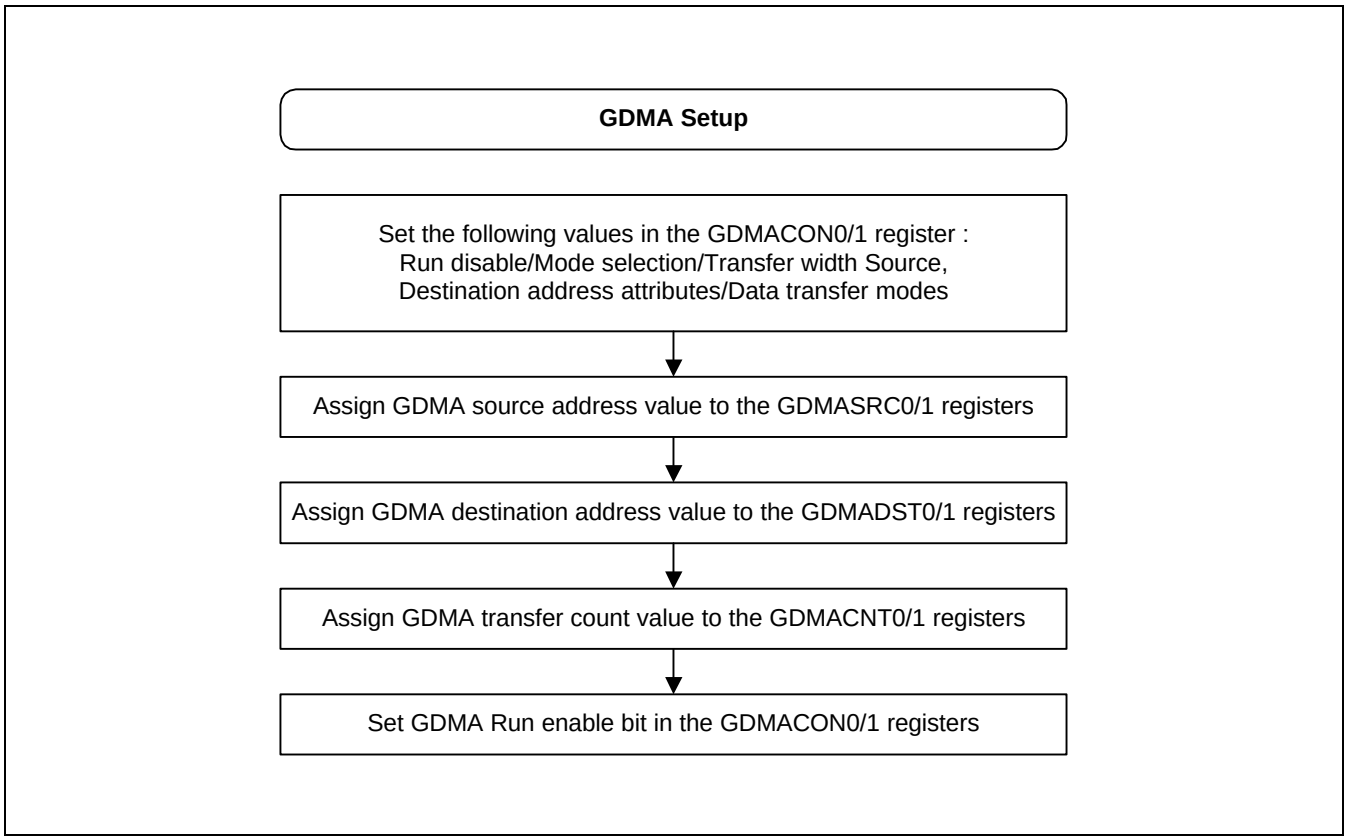


Figure 5-8. Concept Diagram for Setting Up GDMA

EXAMPLE CODE FOR SETTING UP AND EXECUTING DMA <GDMATest.c>

```
//=====================================================================//
// RunGDMA()                                                           //
//                                                                     //
//  Run GDMA function                                                  //
//=====================================================================//
void RunGDMA(int num)
{
    if(num==0)      GDMACON0 |= RUN_ENABLE;
    else  GDMACON1 |= RUN_ENABLE;
}

void MemToMemGDMA(int DmaNum)
{
    int width;      // transfer width

    GDMACON0=0;                         GDMACON1=0; // mode initialization

    gSrcAddr = (unsigned int)gSrcBuf | NON_CACHE_FLAG;
    gDstAddr = (unsigned int)gDstBuf | NON_CACHE_FLAG;

    FillMemory(gSrcAddr, BUF_SIZE, 0x1234abcd);

    Print("\n Select Transfer Width(0:byte 1:half-word 2:word)..");

    switch(getch())
    {
          case '0': // byte transfer
                    if(DmaNum==0)
                        GDMACON0 = (MEM_TO_MEM | GEN_STOP_INT);
                    else
                        GDMACON1 = (MEM_TO_MEM | GEN_STOP_INT);
                    gCount = BUF_SIZE*4;
                    break;
          case '1': // half word transfer
                    if(DmaNum==0)
                        GDMACON0 = (MEM_TO_MEM | GEN_STOP_INT | HALF_WORD);
                    else
                        GDMACON1 = (MEM_TO_MEM | GEN_STOP_INT | HALF_WORD);
                    gCount = BUF_SIZE*2;
                    break;
          case '2': // word transfer
                    if(DmaNum==0)
                        GDMACON0 = (MEM_TO_MEM | GEN_STOP_INT | WORD);
                    else
                        GDMACON1 = (MEM_TO_MEM | GEN_STOP_INT | WORD);
                    gCount = BUF_SIZE;
                    break;
          default : // byte transferS
                    Print("\n Wrong number!! set as default(1byte)..");
                    if(DmaNum==0)
                        GDMACON0 = (MEM_TO_MEM | GEN_STOP_INT);
                    else
                        GDMACON1 = (MEM_TO_MEM | GEN_STOP_INT);
                    gCount = BUF_SIZE*4;
                    break;
          }
```

```
        if(DmaNum == 0)
        {
            SysSetInterrupt(nGDMA0_INT, isr_TestGDMA0);

            // setting SrcAddr, DstAddr, Count
            GDMASRC0 = gSrcAddr;
            GDMADST0 = gDstAddr;
            GDMACNT0 = gCount;

            RunGDMA(0); // Run GDMA
        }
        else
        {
            SysSetInterrupt(nGDMA1_INT, isr_TestGDMA1);

            // setting SrcAddr, DstAddr, Count
            GDMASRC1 = gSrcAddr;
            GDMADST1 = gDstAddr;
            GDMACNT1 = gCount;

            RunGDMA(1); // Run GDMA
        }
}
```

GDMA TEST PROGRAM DESCRIPTION IN DIAGNOSTIC ROM

S5N8947 Diagnostic program contains a program offers GDMA function test and shows the values of GDMA specific registers. For the detail of these functions, refer to timer diagnostic source code provided, ***"GDMATest.c".***

<Main Menu>

```
=========================================================
            CM47-M66-V1.0 Board Diagnostic Ver 1.0
=========================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=========================================================

Select One... :
```

Selecting '[4] GDMA test' menu shows sub-menu of GDMA test below.

```
---------------------------------
        GDMA Test Menu\n
---------------------------------
     [1] Memory to Memory
     [2] Memory to UART
     [3] UART to Memory
     [4] LoopBack GDMA(GDMA0->UART->GDMA1)
     [5] View GDMA Configuration
     [Q] Exit GDMA Test
---------------------------------
  Select One..
```

Descriptions of each menu are below:

- Memory to Memory : Memory to Memory GDMA function test. In this function, there are two memory areas. The one is the source memory area to be transferred and the other is the destination memory area. The contents of memory is assigned in the code. User's option is selecting transfer width (byte/half-word/word). After transfer, It compares the destination area with the source area and gives user a message whether the transfer is successful.
- Memory to UART : Memory to UART GDMA function test. In this function there is a memory area which is going to be transferred to UART. User fills this area with specific data and transfers this area to UART byte by byte. User can see the byte data displayed in the console window.
- UART to Memory : UART to Memory GDMA function test. After fill UART Rx buffer with specific data, transfer to memory.
- Loopback GDMA (GDMA0->UART->GDMA1) : Loopback GDMA test. GDMA0 → UART (Tx Buffer) → UART (Rx Buffer) → GDMA1.
- View GDMA Configuration : shows the values of GDMA specific registers
- Exit GDMA Test : returns to main menu

IIC BUS CONTROLLER

The S5N8947 device' IIC bus controller supports only single master mode. The 64K IIC serial EEPROM, KS24L321/641, is used as the slave device for the usage of S5N8947 device' IIC interface.

This IIC serial EEPROM used as the storage of the target system configuration parameters like as UART Baud Rate, MAC address, IIC serial prescaler clock etc.

FUNCTIONAL DESCRIPTIONS OF KS24L321/641

I^2C-Bus Interface

The KS24L321/641 supports the I^2C-bus serial interface data transmission protocol. The two-wire bus consists of a serial data line (SDA) and a serial clock line (SCL). The SDA and the SCL lines must be connected to V_{CC} by a pull-up resistor that is located somewhere on the bus.

Any device that puts data onto the bus is defined as a "transmitter" and any device that gets data from the bus is a "receiver". The bus is controlled by a master device which generates the serial clock and start/stop conditions, controlling bus access. Using the A0, A1, and A2 input pins, up to eight KS24L321/641 devices can be connected to the same I^2C-bus as slaves (see Figure 5-9. Both the master and slaves can operate as a transmitter or a receiver, but the master device determines which bus operating mode would be active.

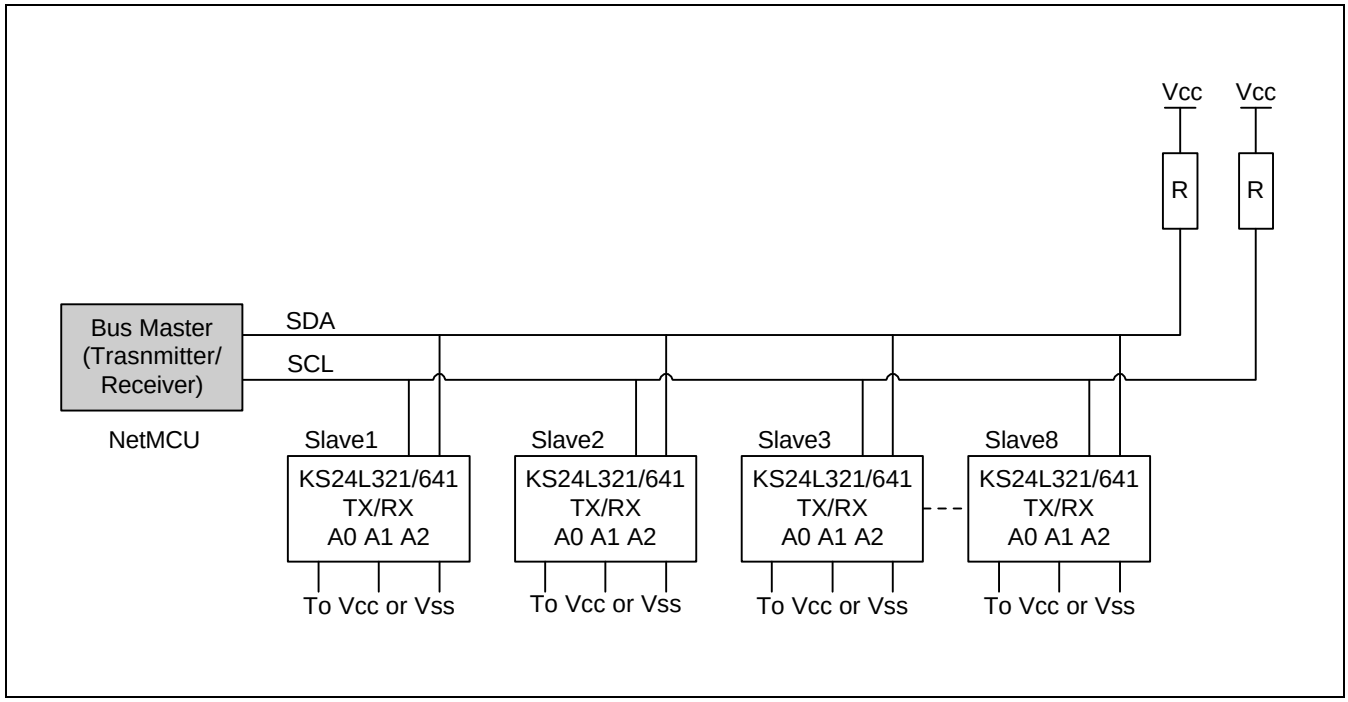


Figure 5-9 Typical Configuration

I²C-BUS PROTOCOLS

Here are several rules for I²C-bus transfers:

- A new data transfer can be initiated only when the bus is currently not busy.
- MSB is always transferred first in transmitting data.
- During a data transfer, the data line (SDA) must remain stable whenever the clock line (SCL) is High.

The I²C-bus interface supports the following communication protocols:

Bus not busy: The SDA and the SCL lines remain in High level when the bus is not active.

Start condition: A start condition is initiated by a High-to-Low transition of the SDA line while SCL remains in High level. All bus commands must be preceded by a start condition.

Stop condition: A stop condition is initiated by a Low-to-High transition of the SDA line while SCL remains in High level. All bus operations must be completed by a stop condition (see Figure 5-10).

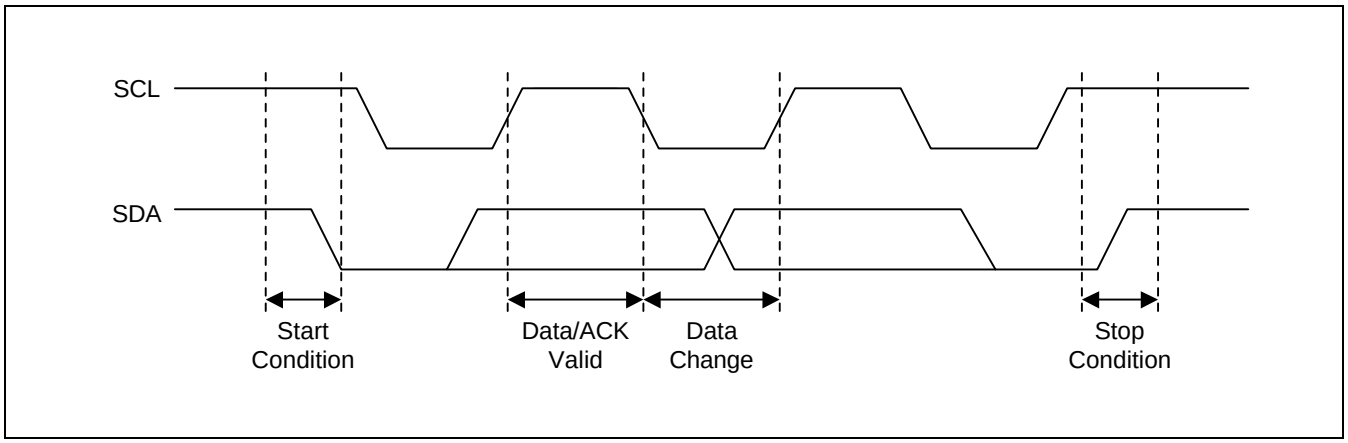


Figure 5-10. Data Transmission Sequence

Data valid: Following a start condition, the data becomes valid if the data line remains stable for the duration of the High period of SCL. New data must be put onto the bus while SCL is Low. Bus timing is one clock pulse per data bit. The number of data bytes to be transferred is determined by the master device. The total number of bytes that can be transferred in one operation is theoretically unlimited.

ACK (Acknowledge): An ACK signal indicates that a data transfer is completed successfully. The transmitter (the master or the slave) releases the bus after transmitting eight bits. During the 9th clock, which the master generates, the receiver pulls the SDA line low to acknowledge that it has successfully received the eight bits of data (see Figure 5-11). But the slave does not send an ACK if an internal write cycle is still in progress.

In data read operations, the slave releases the SDA line after transmitting 8 bits of data and then monitors the line for an ACK signal during the 9th clock period. If an ACK is detected but no stop condition, the slave will continue to transmit data. If an ACK is not detected, the slave terminates data transmission and waits for a stop condition to be issued by the master before returning to its stand-by mode.

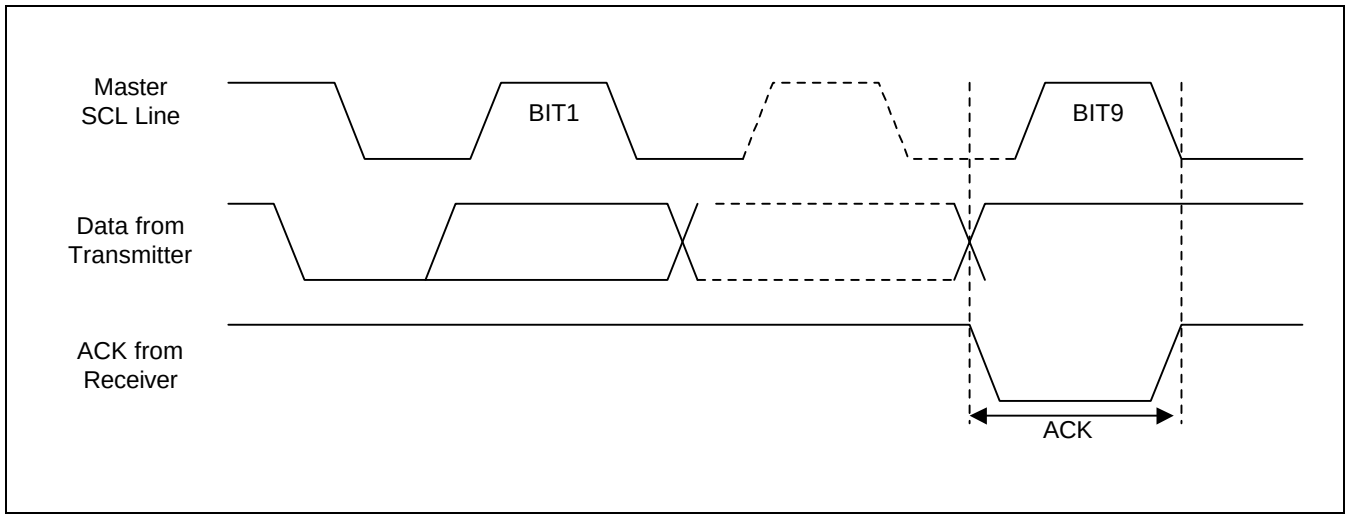
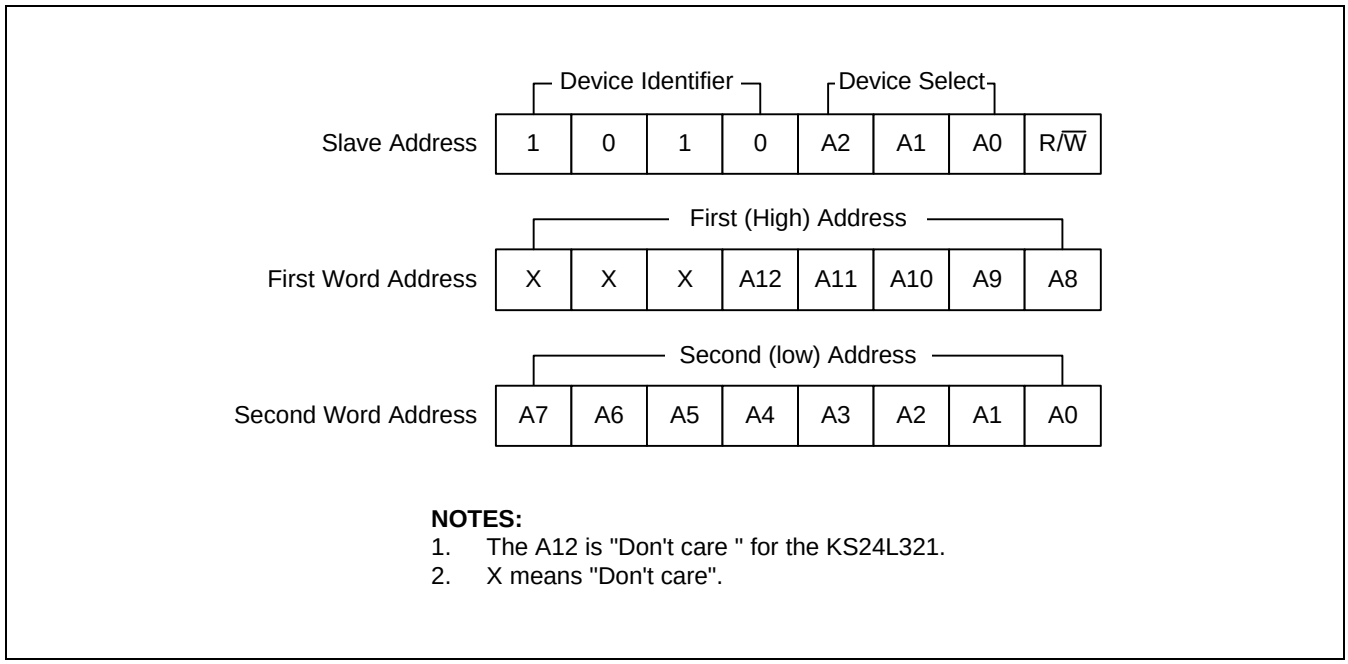


Figure 5-11. Acknowledge Response From Receiver

Slave Address: After the master initiates a start condition, it must output the address of the device to be accessed. The most significant four bits of the slave address are called the “device identifier.” The identifier for the KS24L321/641 is “1010B”. The next three bits comprise the address of a specific device. The device address is defined by the state of the A0, A1, and A2 pins. Using this addressing scheme, you can cascade up to eight KS24L321/641s on the bus.

Read/Write: The final (eighth) bit of the slave address defines the type of operation to be performed. If the R/W bit is "1", a read operation is executed. If it is "0", a write operation is executed



NOTES:
1. The A12 is "Don't care " for the KS24L321.
2. X means "Don't care".

Figure 5-12. Device Address

INITIALIZE THE IIC BUS CONTROLLER

Before the use of IIC bus controller, IIC control status register[IICCON] and IIC prescaler register[IICPS] have to be initialized.

Code below shows the IIC initialize C-routine. Where, the IIC prescaler frequency, fSCL is defined at sysconf.h which is the S5N8947 target system configuration header file.

EXAMPLE FOR IIC BUS SETUP ROUTINE <I2C.C>

```
//=====================================================================//
// IICSetup()                                                          //
//                                                                     //
//  I2C Setup Routine. Initialize IIC control block to use IIC EEPROM  //
//=====================================================================//
void IicSetup(void)
{
          // Reset IIC Controller
          IICCON = IICRESET ;

          // Set Prescale Value: fSCL is IIC serial clock frequency
          // fSCL defined at sysconf.h
          IICPS = SetPreScaler((int)fSCL);  //support upto 100KHz

          // Enable  IIC Interrupt
          SysSetInterrupt(nIIC_INT,IICisr) ;
          Enable_Intr(nIIC_INT) ;
}

//=====================================================================//
// SetPreScaler()                                                      //
//                                                                     //
// Calculate prescaler register value from Serial clock frequency(sclk)//
//=====================================================================//
UINT  SetPreScaler(UINT sclk)
{
    return((UINT)(((fMCLK/sclk)-3.0)/16.0)-0.5);
}
```

IIC WRITE C-FUNCTION LIBRARY

The KS24C321/641 writes up to 32-bytes of data (=page size). A page write operation is initiated in the same way as byte write operation. However, instead of finishing the write operation after the first data byte is transferred, the master can transmit up to 31 additional bytes. The KS24L321/641 responds with an ACK each time it receives a complete byte of data (See Figure 5-13).

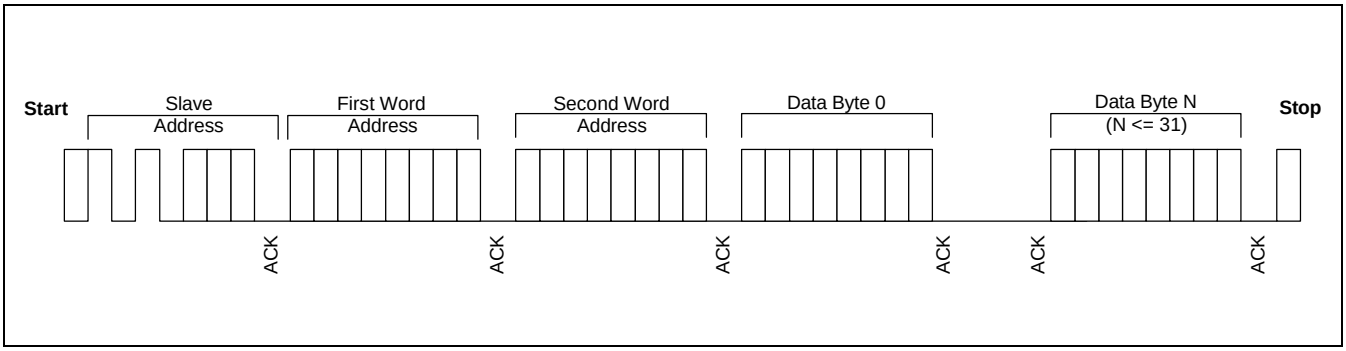


Figure 5-13. Page Write Operation

The KS24L321/641 automatically increments the word address pointer each time it receives a complete data byte. When one byte is received, the internal word address pointer increments to the next address so that the next data byte can be received.

If the master transmits more than 32 bytes before it generates a stop condition to end the page write operation, the KS24L321/641 word address pointer value “rolls over” and the previously received data is overwritten. If the master transmits less than 32 bytes and generates a stop condition, the KS24L321/641 writes the received data to the corresponding EEPROM address.

During a page write operation, all inputs are disabled and there would be no response to additional requests from the master until the internal write cycle is completed.

You can writes data to a Slave (IIC Serial EEPROM) using the IIC write library functions. This write function divides the transfer data to page size and transmit it to Slave until all transfer data is sent. S5N8947 device’s IIC bus controller has the only one interrupt source for IIC. So, whenever the write operation or read operation is started, IIC interrupt service routine have to be installed at vector table using SysSetInterrupt ( ).

The data structure for IIC read & write library function is defined at IIC header file (**i2c.h**).

EXAMPLE CODE FOR IIC WRITE FUNCTION ROUTINE <i2c.c>

```
//====================================================================//
// IICWrite()                                                         //
//                                                                    //
//  The CAT24WC32/64 writes up to 32 bytes of data, in a single write  //
//  cycle, using the page write operation.                            //
//====================================================================//
void IICWrite(UINT WriteAddr, char WriteData[], UINT SizeOfData)
{
    IICPageWrite((UINT)IIC_DEV_0,WriteAddr,WriteData,SizeOfData);
}

void IICPageWrite(UINT SlaveAddr,UINT WriteAddr,char WriteData[],UINT SizeOfData)
{
          UINT WriteDataCnt ;
          UINT ByteAddrMsb;   /* Bytes Address : A15-A8  */
          UINT ByteAddrLsb;   /* Bytes Address : A7-A0  */

          //by ydy
          UINT wait; // Added 12.9 time delay for ACK from slave

          ByteAddrMsb = ((WriteAddr>>8) & 0xff);
          ByteAddrLsb = (WriteAddr & 0xff);

          // Step1. Setup IICCON Register
          IICCON = START | ACK | IEN ;

          // Step2. Send Slave Address with read,write Command
          IICBUF = (SlaveAddr|S_WRITE);

          //while( !IICDoneFlag ) ;
          wait = 1000000;
          while( !IICDoneFlag ) {
              if(!wait) {
                  Print("\n IIC ERROR!!");
                  return;
              }
              wait--;
          }

          IICDoneFlag = 0 ;

          // Step3. Send Bytes Address : A15-A8
          IICBUF = ByteAddrMsb;
          //while( !IICDoneFlag ) ;
          wait = 1000000;
          while( !IICDoneFlag ) {
              if(!wait) {
                  Print("\n IIC ERROR!!");
                  return;
              }
              wait--;
          }
          IICDoneFlag = 0 ;
```

```
            // Step4. Send Bytes Address : A7-A0
            IICBUF = ByteAddrLsb;
            //while( !IICDoneFlag ) ;
            wait = 1000000;
            while( !IICDoneFlag ) {
                if(!wait) {
                    Print("\n IIC ERROR!!");
                    return;
                }
                wait--;
            }
            IICDoneFlag = 0 ;

            // Step5. Send Multiple Data
            for(WriteDataCnt=0;WriteDataCnt<SizeOfData;WriteDataCnt++)
            {
                      IICBUF = WriteData[WriteDataCnt] ;
                      //while( !IICDoneFlag ) ;
                      wait = 1000000;
            while( !IICDoneFlag ) {
                if(!wait) {
                    Print("\n IIC ERROR!!");
                    return;
                }
                wait--;
            }
                      IICDoneFlag = 0 ;
            }

            // Step6. Stop IIC Controller
            IICCON = STOP;
}
```

IIC READ C-FUNCTION LIBRARY

IIC read C library function, IICReadInt ( ), were implemented by using the following Random read byte and Sequential read operation.

Random Address Byte Read Operation

Using random read operations, the master can access any memory location at any time. Before it issues the slave address with the R/W bit set to “1”, the master must first perform a “dummy” write operation. This operation is performed in the following steps:

1. The master first issues a start condition, the slave address, and the word address (the first and the second addresses) to be read. (This step sets the internal word address pointer of the KS24L321/641 to the desired address.)
2. When the master receives an ACK for the word address, it immediately re-issues a start condition followed by another slave address, with the R/W bit set to “1”.
3. The KS24L321/641 then sends an ACK and the 8-bit data stored at the pointed address.
4. At this point, the master does not acknowledge the transmission, generating a stop condition.
5. The KS24L321/641 stops transmitting data and reverts to stand-by mode

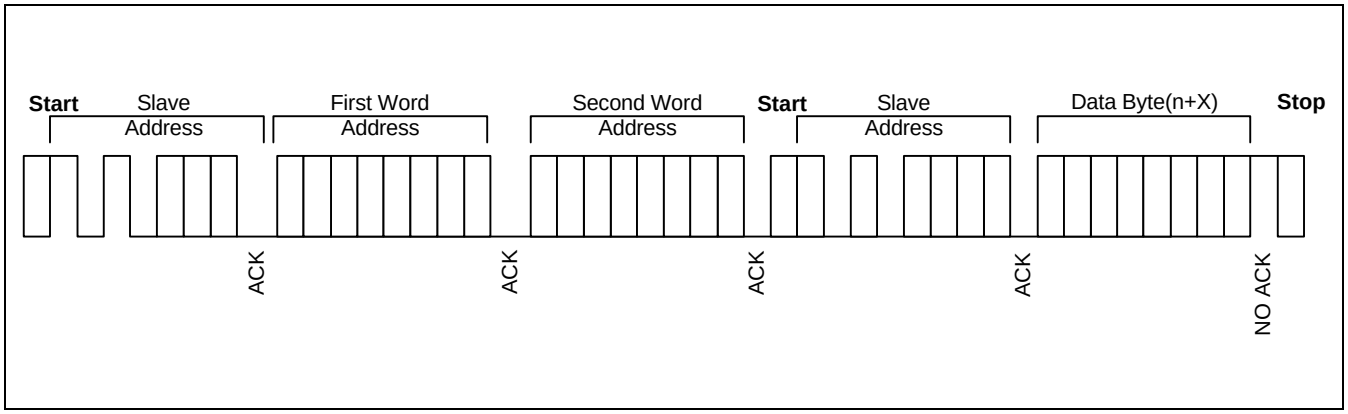


Figure 5-14. Random Address Byte Read Operation

Sequential Read Operation

Sequential read operations can be performed in two ways: current address sequential read operation, and random address sequential read operation. The first data is sent in either of the two ways, current address byte read operation or random address byte read operation described earlier. If the master responds with an ACK, the KS24L321/641 continues transmitting data. If the master does not issue an ACK, generating a stop condition, the slave stops transmission, ending the sequential read operation.

Using this method, data is output sequentially from address “n” followed by address “n+1”. The word address pointer for read operations increments to all word addresses, allowing the entire EEPROM to be read sequentially in a single operation. After the entire EEPROM is read, the word address pointer “rolls over” and the KS24L321/641 continues to transmit data for each ACK it receives from the master.

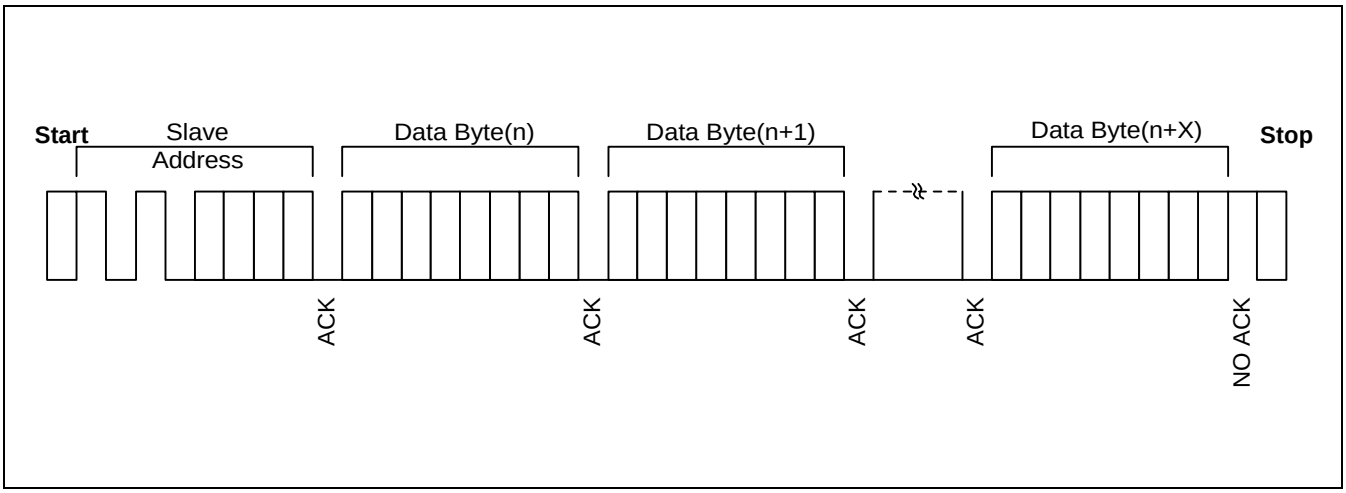


Figure 5-15. Sequential Read Operation

EXAMPLE CODE FOR IIC READ FUNCTION ROUTINE <i2c.c>

```
//=====================================================================//
// IICRead()                                                           //
//                                                                     //
//  The Sequential READ operation can be initiated by either           //
//  the Immediate Address READ or Selective READ operation             //
//=====================================================================//
void IICRead(UINT ReadAddr, UINT ReadDataSize, char ReadValue[])
{
          IICSeqRead((UINT)IIC_DEV_0, ReadAddr, ReadValue, ReadDataSize);
}

void IICSeqRead(UINT SlaveAddr,UINT ReadAddr,char ReadValue[],UINT ReadDataSize)
{
          UINT ReadCnt ;
          UINT ByteAddrMsb;   /* Bytes Address : A15-A8  */
          UINT ByteAddrLsb;   /* Bytes Address : A7-A0  */
          UINT wait;

          ByteAddrMsb = ((ReadAddr>>8) & 0xff);
          ByteAddrLsb = (ReadAddr & 0xff);

          // Step1. Setup IICCON Register
          IICCON = START | ACK | IEN ;

          /* Step2. Send Slave Address with read,write Command */
          IICBUF = (SlaveAddr|S_WRITE) ;
          //while( !IICDoneFlag ) ;
          wait = 1000000;
          while( !IICDoneFlag ) {
              if(!wait) {
                  Print("\n IIC ERROR!!");
                  return;
              }
              wait--;
          }
          IICDoneFlag = 0 ;

          /* Step3. Send Bytes Address : A15-A8  */
          IICBUF = ByteAddrMsb;
          //while( !IICDoneFlag ) ;
          wait = 1000000;
          while( !IICDoneFlag ) {
              if(!wait) {
                  Print("\n IIC ERROR!!");
                  return;
              }
              wait--;
          }
          IICDoneFlag = 0 ;

          /* Step4. Send Bytes Address : A7-A0  */
          IICBUF = ByteAddrLsb;
          //while( !IICDoneFlag ) ;
          wait = 1000000;
          while( !IICDoneFlag ) {
              if(!wait) {
```

```
            Print("\n IIC ERROR!!");
            return;
        }
        wait--;
    }
    IICDoneFlag = 0 ;

    // Step5. Repeat Start
    IICCON = RESTART ;
    IICCON = START | ACK | IEN ;
    IICBUF = SlaveAddr | S_READ ;
    //while( !IICDoneFlag ) ;
    wait = 1000000;
    while( !IICDoneFlag ) {
        if(!wait) {
            Print("\n IIC ERROR!!");
            return;
        }
        wait--;
    }
    IICDoneFlag = 0 ;

    // Step6. Receive Multiple Data
    IICCON = ACK|IEN ;
    for (ReadCnt=0 ; ReadCnt < ReadDataSize - 1 ; ReadCnt++) {
              //while( !IICDoneFlag ) ;
              wait = 1000000;
              while( !IICDoneFlag ) {
                                  if(!wait) {
                                  Print("\n IIC ERROR!!");
               return;
                                  }
              wait--;
              }
              IICDoneFlag = 0 ;
              ReadValue[ReadCnt] = IICBUF ;
    }

    // Receive Last Data and ACK Disable
    IICCON = AckDisable | IEN ;
    //while( !IICDoneFlag ) ;
    wait = 1000000;
    while( !IICDoneFlag ) {
        if(!wait) {
            Print("\n IIC ERROR!!");
            return;
        }
        wait--;
    }
    IICDoneFlag = 0 ;
    ReadValue[ReadCnt] = IICBUF ;

    // Step7. Stop IIC Controller
    IICCON = STOP ;
}
```

IIC BUS TEST PROGRAM DESCRIPTION IN DIAGNOSTIC ROM

S5N8947 Diagnositc program contains a program offers IIC bus controller function test and shows the values of IIC bus controller registers. For the detail of these functions, refer to IIC bus diagnostic source code provided, "i2c.c".

<Main Menu>

```
=========================================================
            CM47-M66-V1.0 Board Diagnostic Ver 1.0
=========================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=========================================================
Select One... :
```

Selecting '[5] I2C BUS TEST' at main menu shows sub-menu of I2C bus test below.

```
-------------------------------------------------
                IIC Test Menu
-------------------------------------------------
     [1] IIC Page Write Test(INT)
     [2] IIC Sequentail Read Test(INT)
     [3] IIC Read/Write Test(INT)
     [4] IIC Page Write Test(POLL)
     [5] IIC Sequentail Read Test(POLL)
     [6] IIC Read/Write Test(POLL)
     [7] IIC Byte random R/W Test
     [8] IIC Loopback Test(INT)
     [9] IIC configuration view
     [Q] Exit IIC Test
-------------------------------------------------
Select One..
```

[1] IIC Page Write Test (INT) :IIC write test by interrupt
[2] IIC Sequentail Read Test (INT) : IIC read test by interrupt
[3] IIC Read/Write Test (INT) : IIC Read/Write test program by interrupt method
[4] IIC Page Write Test (POLL) : IIC Write test program by polling method
[5] IIC Sequentail Read Test (POLL) : IIC read test by polling
[6] IIC Read/Write Test (POLL) : IIC Read/Write test program by polling method
[7] IIC Byte random R/W Test : IIC Write one byte of data to a given address of EEPROM
[8] IIC Loopback Test (INT) : IIC read/write Loopback test
[9] IIC Configuration view : Shows IIC related register value

I/O PORTS

The General I/O ports (P0 – P17) shared with a special function such as an external interrupt request, an external DMA request & acknowledge and timer signal outputs. I/O port mode can be configured by I/O port mode register (IOPMOD). But, the shared function will be configured by the IOPCON register not by IOPMOD.

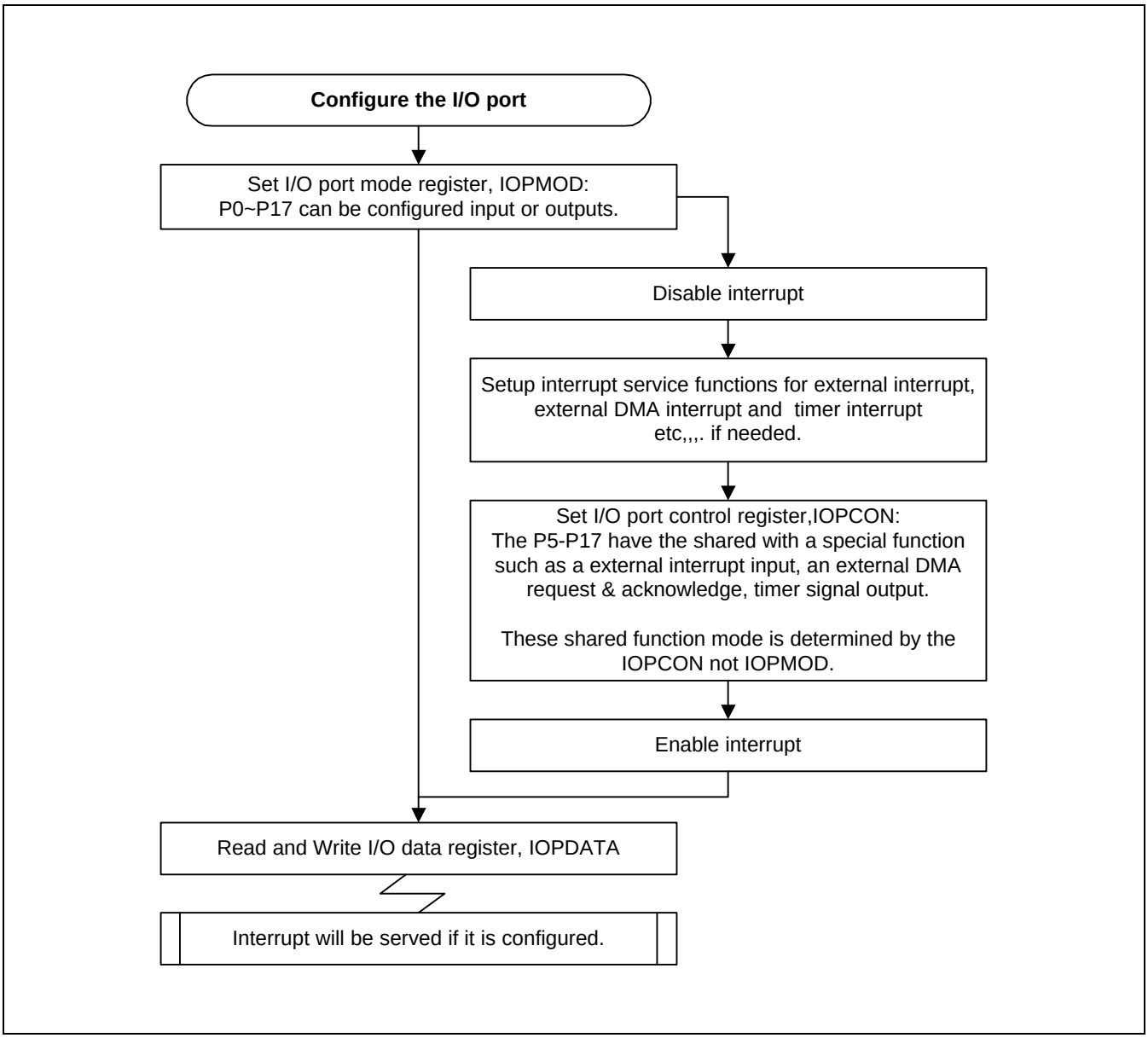


Figure 5-16. Concept Diagram for the Configuring of I/O Port

EXAMPLE I/O PORT SETUP ROUTINE (AS EXTERNAL INTERRUPT SOURCE)

```
//=====================================================================//
// ExternalInterruptTest()                                             //
//                                                                     //
//  Test I/O Ports as External Interrupt request Source                //
//=====================================================================//
void ExternalInterruptTest(void)
{
    IOPMOD &= 0xff01f; // set port5~11 into input ports
    IOPCON0 = 0; // initialize IOPCON0 register
   IOPCON1 = 0; // initialize IOPCON1 register

    // mapping the ISRs
    SysSetInterrupt(nEXT0_INT, isr_XIrq_0);
    SysSetInterrupt(nEXT1_INT, isr_XIrq_1);

    // Enables interrupts
    Enable_Intr(nEXT0_INT);
    Enable_Intr(nEXT1_INT);

    SetXIrqIOPCON();

    Print("\n\n>>xIRQn Enabled...(ESC To EXIT)\n");
    // To EXIT test
    while(1)
        if(getch() == ESC)  break;

    Disable_Intr(nEXT0_INT);
    Disable_Intr(nEXT1_INT);
}

//=====================================================================//
// SetXIrqIOPCON()                                                     //
//                                                                     //
//  Set IOPCON register for xINTREQ0 ~ xINTREQ6 IOPort[5..11]          //
//=====================================================================//
void SetXIrqIOPCON(void)
{
    ControlXIrq(IOPORT_8, ACTIVE_LOW, FILTERING_ON, BOTH_EDGE);
    ControlXIrq(IOPORT_9, ACTIVE_LOW, FILTERING_ON, FALLING_EDGE);
}
```

```
//=====================================================================//
// ControlXIrq()                                                       //
//                                                                     //
//  Set each IO Port as given parameters                               //
//     PortNum : IO Port number                                        //
//     Active : active low/high                                        //
//     Filtering : filtering on/off                                    //
//     Detect : level/rising edge/falling edge/both edge detection     //
//=====================================================================//
void ControlXIrq(UINT PortNum, UINT Active, UINT Filtering, UINT Detect)
{
    unsigned int con;

    con = (XIRQ_ENABLE | Active | Filtering | Detect);
    switch(PortNum)
    {
        case IOPORT_8:
            break;
        case IOPORT_9:
            con <<= 5;
            break;
        case IOPORT_10:
            con <<= 10;
            break;
        case IOPORT_11:
            con <<= 15;
            break;
    }
    IOPCON |= con;
}

//=====================================================================//
// isr_XIrq_0()                                                        //
//                                                                     //
//  Interrupt Service Routine for External Interrupt Request 0         //
//=====================================================================//
void isr_XIrq_0(void)
{
    Print("\n !! xIRQ_0 Occurred !! (ESC To EXIT Test)\n");
    l2Print("External IRQ0", "ESC To EXIT");
}

//=====================================================================//
// isr_XIrq_1()                                                        //
//                                                                     //
//  Interrupt Service Routine for External Interrupt Request 1         //
//=====================================================================//
void isr_XIrq_1(void)
{
    Print("\n !! xIRQ_1 Occurred !! (ESC To EXIT Test)\n");
    l2Print("External IRQ1", "ESC To EXIT");
}
```

I/O PORT TEST PROGRAM DESCRIPTION IN DIAGNOSTIC ROM

S5N8947 Diagnostic program contains a program offers I/O Port function test and shows the values of I/O Port specific registers. For the detail of these functions, refer to I/O Port diagnostic source code provided, ***"ioPort.c".***

<Main Menu>

```
=======================================================
            CM47-M66-V1.0 Board Diagnostic Ver 1.0
=======================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=======================================================
Select One... :
```

Selecting '[6] I/O Port test' at main menu shows sub-menu of I/O Port test below.

```
-----------------------------------------------
            IO Port Test Menu\n");
-----------------------------------------------
     [1] IO Port Read (port[0..7])
     [2] IO Port Write (port[0..7])
     [3] External Interrupt Test (port[5..11])
     [4] View Configuration
     [5] IOPDATA Read (Input mode)
     [Q] Exit IO Port Test
-----------------------------------------------
Select One..
```

[1] IO Port Read (port[0..7])

[2] IO Port Write (port[0..7])

In our S5N8947 Evaluation board, I/O Ports 0 through 7 are connected to LEDs (D12..D19). If you read I/O Port[0..7], It reflects LEDs status. If you write a value to I/O Port[0..7], the LEDs reflect the value you wrote. The LED is off when you write ‘1’ to corresponding I/O Port and the LED is on if you write ‘0’ to the port. With sub-menu [1] and [2], you can read and write to I/O Port and check the result from the LEDs status.

[3] External Interrupt Test (port[5..11])

I/O Ports[5..11] are used for External Interrupt Request input ports. In our S5N8947 Evaluation board, these are connected to 4 push-button SWs. With this SWs, you can test External Interrupts. In this function, if you push one of the SWs (S4..S7), you will see the message says external interrupts occurrence.

[4] View Configuration : Shows the values of I/O Port specific registers.

[5] IOPDATA Read (Input mode)

You can configure the I/O Ports as inputs or outputs with configuring the IOPMOD register. In this menu function, I/O Port[0..17] are set as input.

ETHERNET CONTROLLER

MAC DIAGNOSTIC CODE FUNCTION

The diagnostic source code for the MAC (Medium Access Controller) is composed of four files, MAC.H, MAC.C, MACINIT.C, and MACLIB.C.

- **MAC.H** : Definition file for MAC diagnostic code, Register bit value, frame structure, frame descriptor structure, and function prototype.
- **MAC.C** : The main diagnostic code function call for MAC function test.
- **MACINIT.C** : Initialize MAC and BDMA controller for normal operating environment, PHY configuration, and each interrupt service routine.
- **MACLIB.C** : The library functions for diagnostic code.

LAN INITIALIZE

The Ethernet controller initialize function is composed of configure PHY device and initialize MAC, BDMA controller. The configuration of PHY device is performed by MII station management function. This function is provided by MAC controller special function. The configuration method of PHY device is something different between each vendor, in this diagnostic code use only simple code for configure PHY, and Full/Half duplex mode.

The S5N8947 has MAC and BDMA controller for Ethernet interface. The BDMA is used for transfer receive data to memory and transfer the transmit data to MAC. The MAC can support 10/100Mbps and Full/Half duplex mode, So you need to set the MAC, and BDMA controller to work as your purpose.

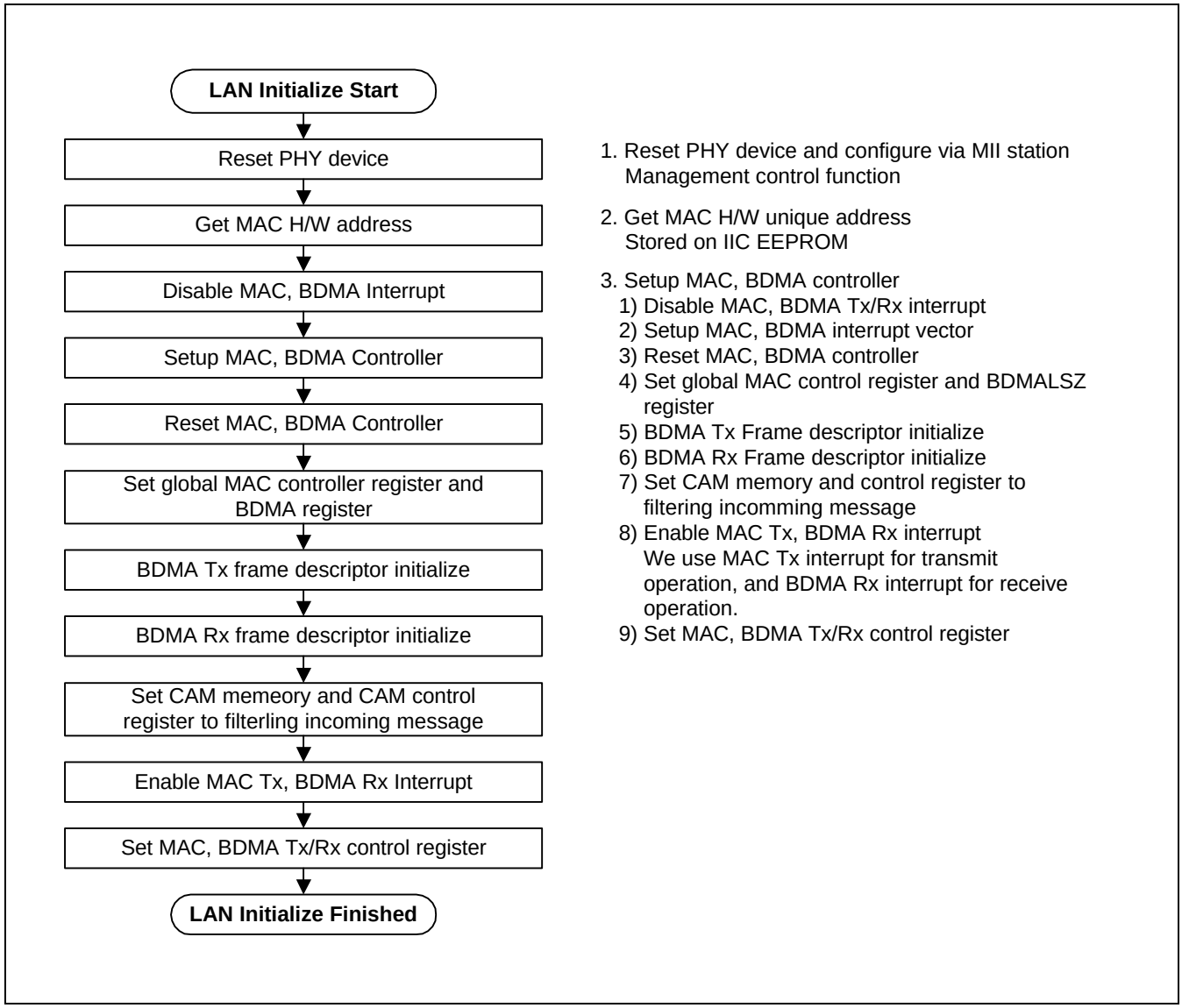


Figure 5-17. LAN Initialize Flow

1. Reset and configure PHY device

The station management operation is used to control PHY. The PHY has many control and status registers. MAC can control PHY, and it can read the PHY status. PHY has a unique address, and there is many addresses for PHY internal control and status registers.
The operation of read, and write to the station management register is described in below.

MII Station Management Read Operation

STEP 1. Specify the PHY address and a PHY internal register address in the STACON register
STEP 2. Set busy bit in STACON, then a PHY read operation is started.
STEP 3. Check Busy bit in STACON, after read-operation is finished, this bit is cleared.
STEP 4. Read STADATA, the STADATA register has the value of the PHY station register.

MII Station Management Write Operation

STEP 1. Write the station management data to STADATA register.
STEP 2. Specify the PHY address, and a PHY internal register address in STACON.
STEP 3. Set Write, and Busy bit in STACON, then PHY write operation is started.
STEP 4. Check Busy bit in STACON, when the write operation is finished, this bit is cleared.

2. Get unique H/W MAC address

The every MAC has unique H/W address, this is used when filtering incoming message from network. The CAM is used for filtering incoming message address. In CM47-M66-V1.0 use IIC EEPROM for store MAC H/W address for system.

3. Setup MAC, BDMA controller

This function is used for setup MAC and BDMA controller register and Interrupt service routine for receive and transmit.

The each step of MAC initialize function is described in below.

STEP 1 : Disable MAC and BDMA interrupt

Disable MAC Tx, BDMA Rx interrupt to avoid abnormal branch.

STEP 2 : BDMA and MAC interrupt vector setup

Interrupt vector setup for MAC, and BDMA interrupt, after this statement, MAC, and BDMA interrupt can be used.

STEP 3 : Set the initial condition of BDMA and MAC

Reset the MAC controller and BDMA controller. And set the duplex mode and interface mode to MACCON register. Ethernet interface can be configured as MII interface or old style 7-wire interface. The 7-wire interface can be set by MII-OFF bit.

STEP 4 : Set the BDMA Tx/Rx Frame Descriptor

BDMA Frame descriptor is used for control BDMA automatically. The transmit BDMA frame descriptor has frame buffer pointer, control field, length and status field, and next frame descriptor field, the BDMA receive frame descriptor is almost same as transmit frame descriptor except control field.

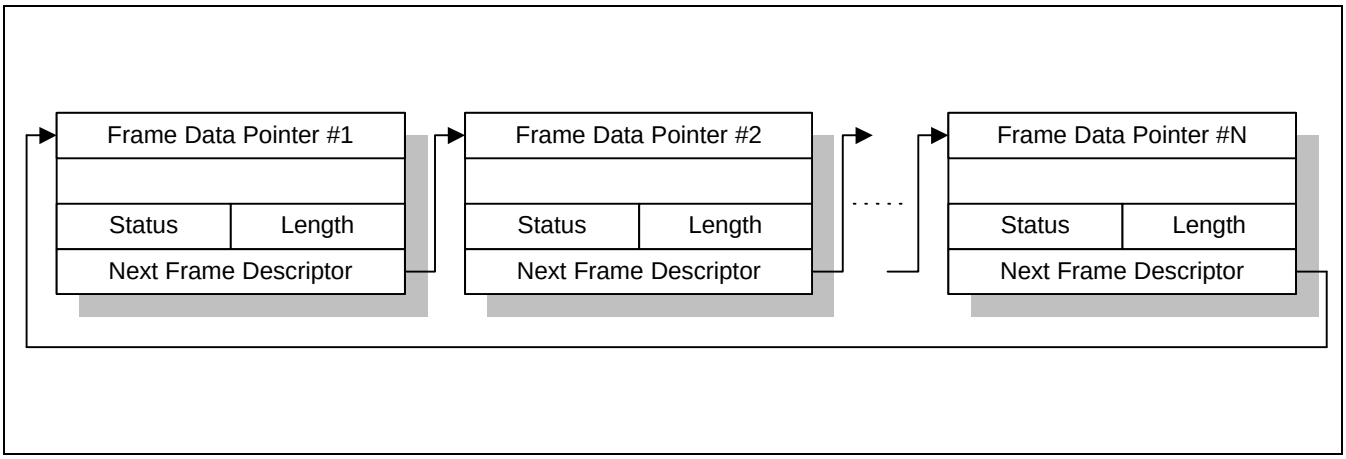


Figure 5-18. BDMA Frame Descriptor Structure

The BDMA Tx/Rx frame descriptor and frame buffer area should be non-cacheable, because BDMA can update the value, so when we initialize frame descriptor, add 0x4000000 for non-cacheable access.

The default owner of transmit BDMA frame descriptor is CPU, and the default owner of receive BDMA owner is BDMA, after receive frame, BDMA controller change the owner bit on BDMA frame descriptor to CPU owner, then it can be used by CPU.

STEP 5 : Set the CAM Control register and the MAC address value

CAM is used for filtering received frame from other frames. The S5N8947 has 21 CAM, but in the diagnostic code uses only 1 CAM. In diagnostic code for CM47-M66-V1.0 use IIC EEPROM for store MAC H/W address. So before enable the receive operation, We should load the MAC address to CAM, and set the value of CAM enable/control register. The **STEP 5** is doing this operation.

STEP 6 : Enable interrupt BDMA Rx and MAC Tx interrupt

In our diagnostic code, we only use BDMA Rx interrupt for receive operation, and MAC Tx interrupt for transmit operation. You can refer transmit and receive operation of Ethernet controller for more detail of BDMA Rx, and MAC Tx interrupt service routine.

STEP 7 : Configure the BDMA and MAC control registers

This step, prepare all BDMA and MAC control register to operate. After this set of BDMARXCON, and MACRXCON, Ethernet controller can receive incoming frame.

The value for MAC and BDMA basic operation is described in Table 5-1.

Table 5-1. MAC and BDMA Control Register Set Value

Register Name	Function	Control bit	Description
MACCON	MAC global control register	FullDup*	Full-Duplex mode set
		MII-OFF*	7-wire or MII interface set
MACTXCON	MAC Transmit control register	TxEn	Set this bit to enable transmission
MACRXCON	MAC Receive control register	RxEn	MAC Receive enable
		StripCRC	Check the CRC, but strip the from message
BDMATXCON	BDMA Transmit control register	BTxBRST	BDMA transmit burst size
		BTxMSL	BDMA Tx to MAC Tx start level
		BTxSTSKO	BDMA Tx stop when met not owned frame
BDMARXCON	BDMA Receive control register	BRxEn	BDMA receive enable
		BRxLittle	Received data is stored in Little-endian mode (Used when Little-endian is selected)
			Received data is stored in Big-endian mode (Used when Big-endian is selected)
		BRxMAINC	Received data is stored in memory address increment
		BRxBRST	BDMA Rx burst size
		BRxNLIE	BDMA Rx null list interrupt enable
		BRxNOIE	BDMA Rx not owner interrupt enable
		BRxSTSKO	BDMA Rx stop when met not owned frame

TRANSMIT ETHERNET FRAME

After set all control register, and BDMA transmit frame descriptor, we are ready to transmit packet. When transmit packet, you can follow this step.

STEP 1: Get transmit frame descriptor and data pointer

Get current frame descriptor and data pointer, that will be used for prepare Ethernet frame data and transmit control function.

STEP 2: Check BDMA ownership.

Check BDMA owner is CPU or not, when BDMA owner is CPU, CPU can be write control bit, and frame data. If owner is BDMA then exit send packet function.

STEP 3: Prepare Tx frame data to frame buffer.

Copy Ethernet frame data to BDMA frame buffer, this pointer is pointed by frame data field on frame descriptor.

STEP 4: Set Tx frame flag and length field.

After copy Ethernet frame to BDMA frame buffer, CPU write control bit and the length of frame data.

STEP 5: Change ownership to BDMA.

When ownership is BDMA, BDMA can work, so after prepare transmit frame descriptor and frame data ownership changed to BDMA owner.

STEP 6: Enable MAC and BDMA transmit control register to start transmit.

STEP 7: Change current frame descriptor to next frame descriptor.

The Ethernet frame transmit flow is depicted in Figure 5-19.

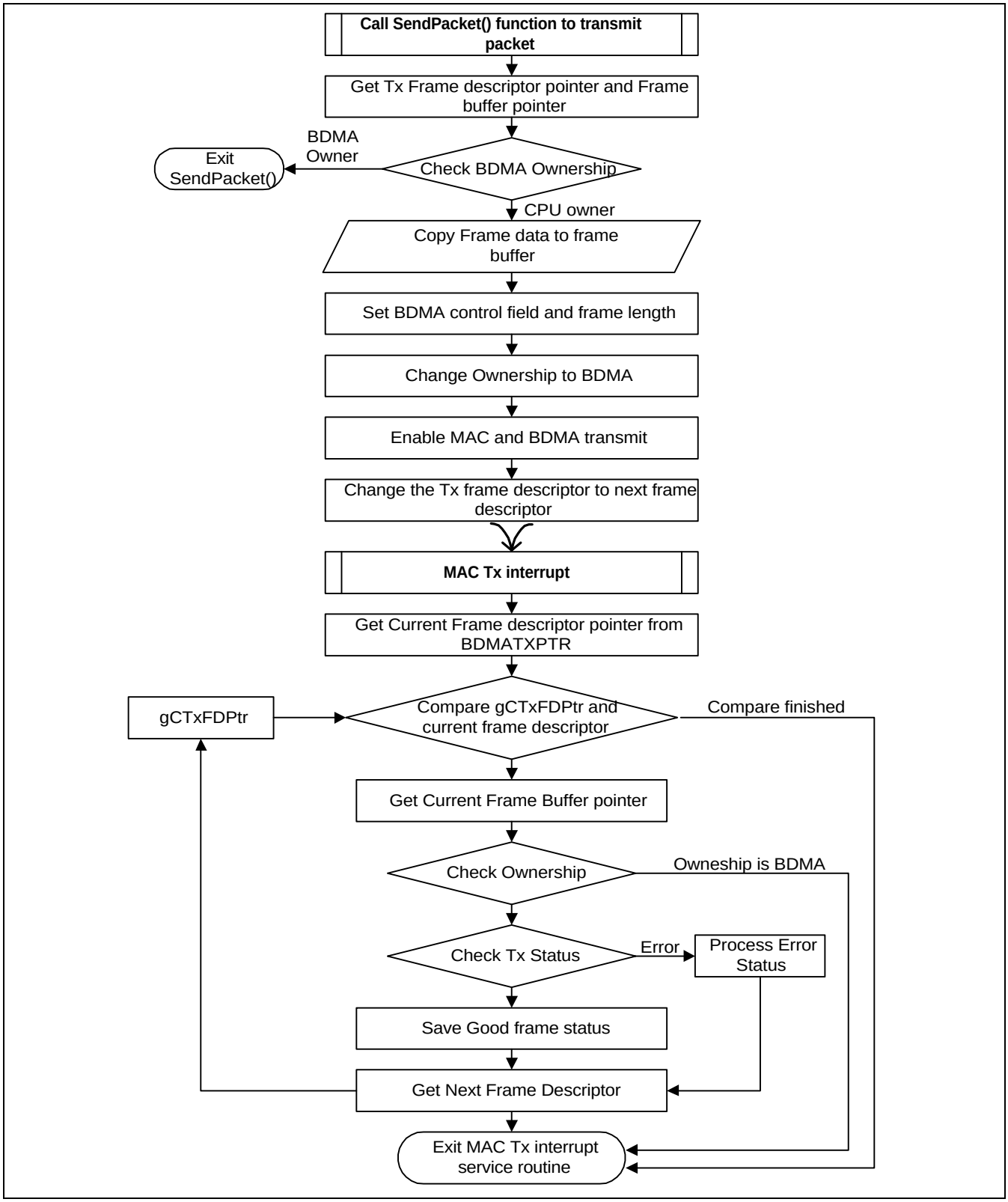


Figure 5-19. Ethernet Frame Transmit Flow

CONTROL FRAME TRANSMIT

You can transmit a control frame for a remote pause operation in full-duplex mode operation. S5N8947 Ethernet controller has a function of transmitting and receiving control frame. When transmit control frame, follow this step.

STEP 1. Set Destination address to CAM #0

STEP 2. Set Source address to CAM #1

STEP 3. Set length or type field, op-code, and operand to CAM #18

STEP 4. Set zero to double word that proceed CAM #18.

STEP 5. Enable CAM location by CAMEN register.

STEP 6. Enable transmit control frame by set the SendPause bit in MACTXCON register.

STEP 7. Wait control frame transmit is finished.

RECEIVE ETHERNET FRAME

Receive operation of Ethernet frame is performed only on the BDMA Rx interrupt service routine. A BDMA Rx interrupt is occurred, when a frame reception is finished. The detail Ethernet frame reception operation, follow this step.

STEP 1: Get current frame descriptor pointer and BDMA status

This step is used for get current frame descriptor's pointer from BDMARXPTR register. The BDMARXPTR register value denotes the current processing frame descriptor point or the next frame descriptor pointer. So this value is used for check last received frame or not.

STEP 2: Check Null list interrupt

Null list interrupt means, BDMARXPTR has 0x00000000 value, this value is not accepted, so when we met this interrupt, we should initialize MAC, and BDMA controller again.

STEP 3: Get Rx Frame Descriptor

In this step, we get receive frame descriptor's pointer to process data, every receive process, use BDMA receive frame descriptor pointer.

STEP 4: Check received frame is valid or not

Check received frame descriptor status field to check received frame is valid or not.

STEP 5-1: Get received frame to memory buffer

This step is main function that copy received frame to memory buffer to process. So in the various RTOS can announce received frame in this step.

STEP 5-2: Process error status

In this step check received frame descriptor status field, to check this frame has error or not.

STEP 6: Change ownership to BDMA

Change BDMA ownership to BDMA, because BDMA can use this frame descriptor after receive operation.

STEP 7: Get next frame descriptor pointer to process

When enter BDMA receive interrupt service routine, we process all received frame before receive interrupt.

STEP 8: Check BDMA Not Owner status bit in the BDMASTAT register

This Not Owner bit means all BDMA frame descriptor is used, so we need set MAC and BDMA control register to receive frame normally.

The receive operation in the BDMA Rx interrupt service routine is described in Figure 5-20.

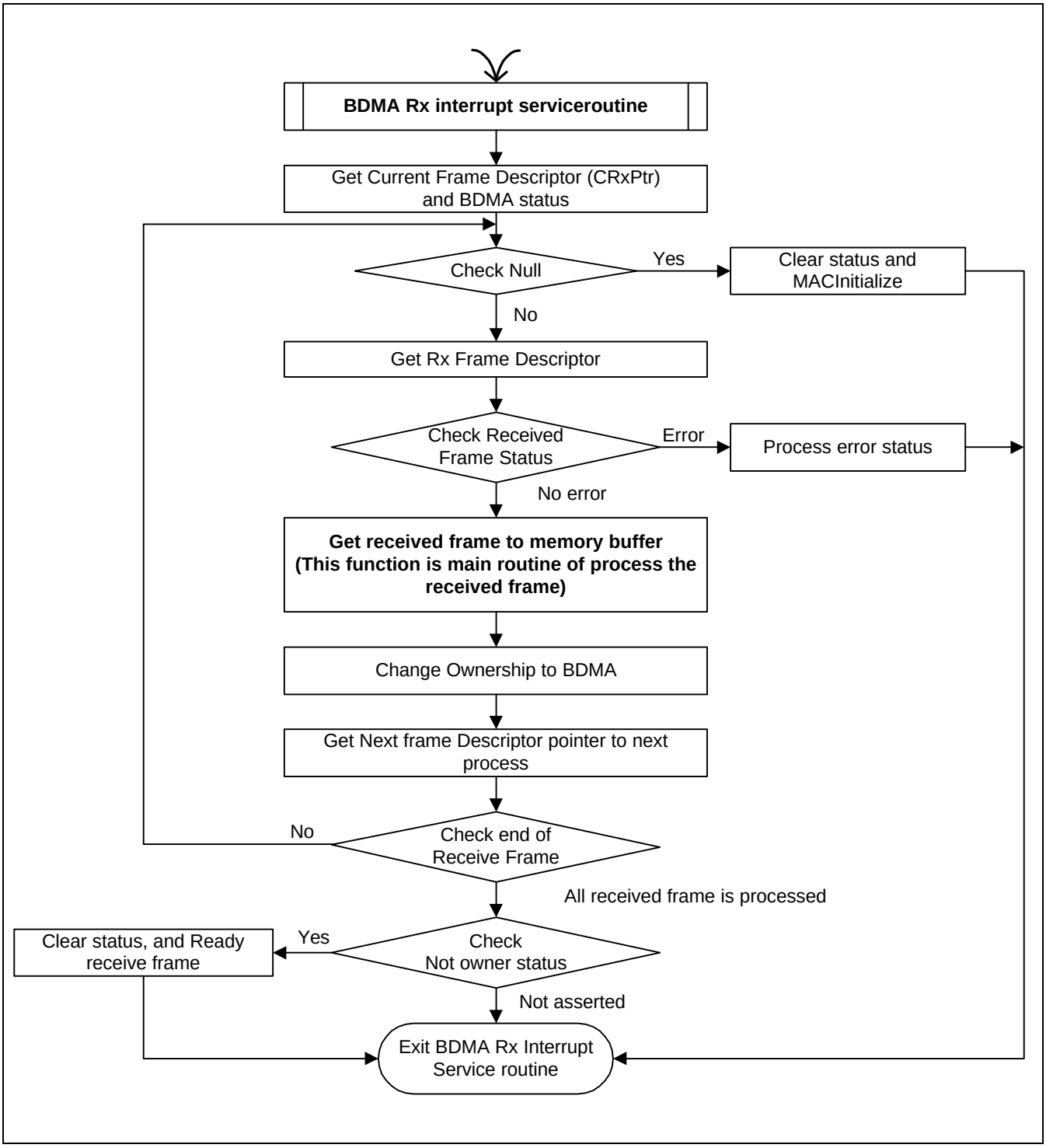


Figure 5-20. Ethernet Frame Reception Flow

UNIVERSAL SERIAL BUS CONTROLLER

USB DIAGNOSTIC CODE FUNCTION

CAUTION

1. Endpoint 1 is used for loopback test with endpoint 2, Do not use it in your product.

2. USB Controller in S5N8947 needs S/W DMA arbitration. If you want to use USB in your product, request us a new sample code to prevent data from corruption. Next version of Programmer's Guide will contain it.

The diagnostic code for the USB (Universal Serial Bus) is composed of five files, USB.H, USBFUNC.H, USBDESC.H, USB.C, and USBFUNC.C.

USB.H : Definition file for USB diagnostic code, It defines USB register address and global variables.

USBFUNC.H : Definition file for USB diagnostic code, It defines simple USB function, USB request type, global variables and register bit value.

USBDESC.H : Definition file for USB diagnostic code, It defines USB configuration descriptor, USB device descriptor and USB endpoint descriptor.

USB.C : Initialize USB controller for normal operating environment and Register interrupt service routine.

USBFUNC.C : The library functions for diagnostic code which are called by interrupt service routine. It includes USB Tx/Rx function and USB request service function.

First USB sends descriptor's contents in USBDesc.h to Host, Host checks Product ID and Vendor ID of descriptor and transfers data packet. All data transfer are same behavior method and each endpoint is assigned only transmitter or receiver. But, endpoint0 can transmit and receive. Endpoint1, 3 of S5N8947's USB is BulkOut endpoints that can receive host's data, Endpoint2, 4 of S5N8947's USB is BulkIn endpoints that can send data to host.

— Host Application program contains Test Device Driver and DOS Application program.

To test the USB controller, plug in the USB cable to the host and install Test Device Driver. (Bulkusb.sys, Bulkusb.inf) If USB device is installed well, execute Dos application program.

(ex) Bulk34 –w (byte length) –r (byte length) –c (loop count) -> endpoint 3, 4 loopback test.

(ex) Int12 –w (byte length) –r (byte length) –c (loop count) -> endpoint 1, 2 loopback test.

USB INITIALIZE

(1) Initialize USB registers

(2) Register USB interrupt service routine and enable interrupt.

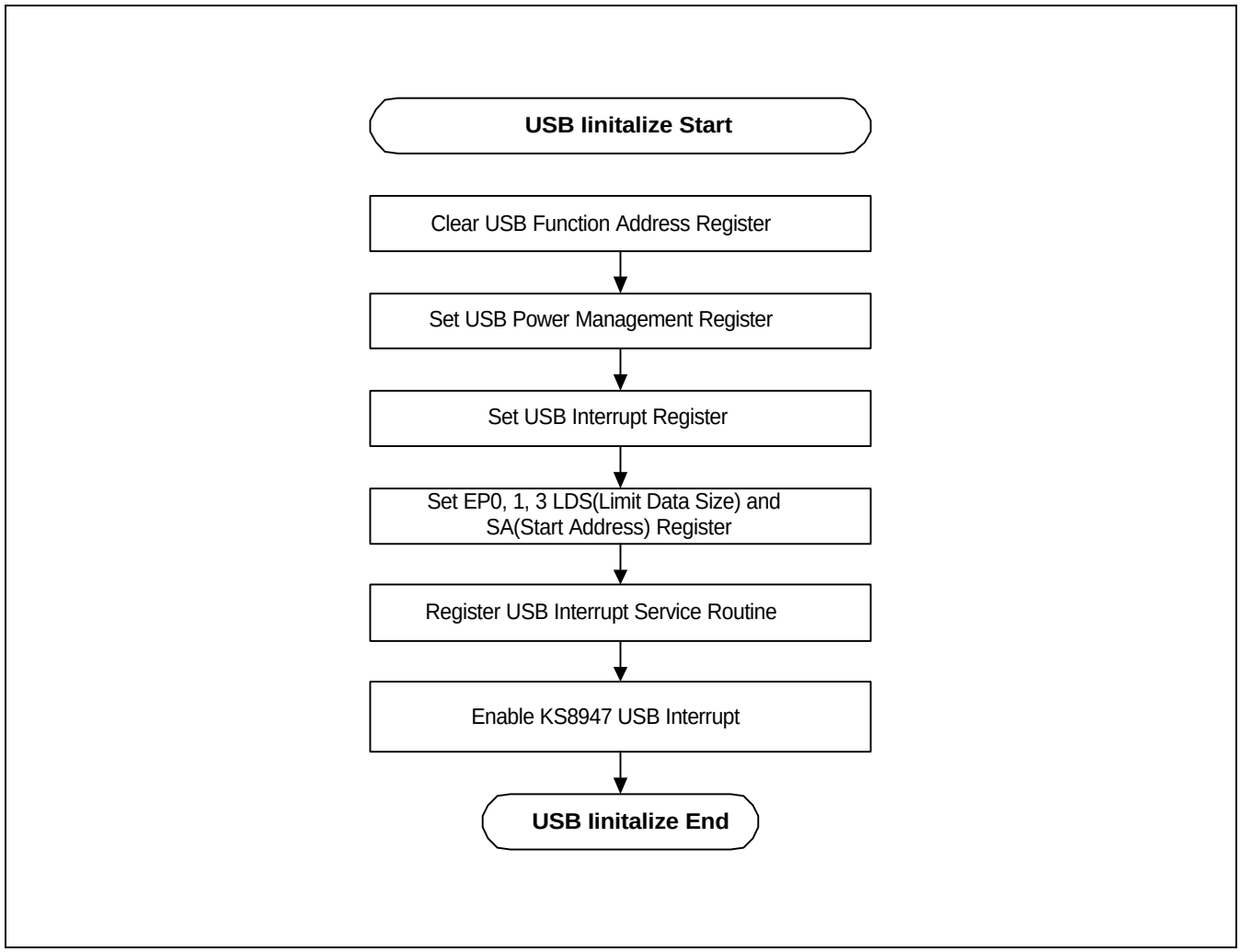


Figure 5-21. Concept Diagram for USB initialization

USB INTERRUPT HANDLER

(1) Read USB Interrupt Register and Clear it.

(2) If Endpoint0 interrupt occurs, call function for Control transfer. (endpointzeroFunction)

(3) If Endpoint1 interrupt occurs, call function for Interrupt Out transfer. (endpointInterruptOutFunction)

(4) If Endpoint3 interrupt occurs, call function for Control transfer. (endpointBulkOutFunction)

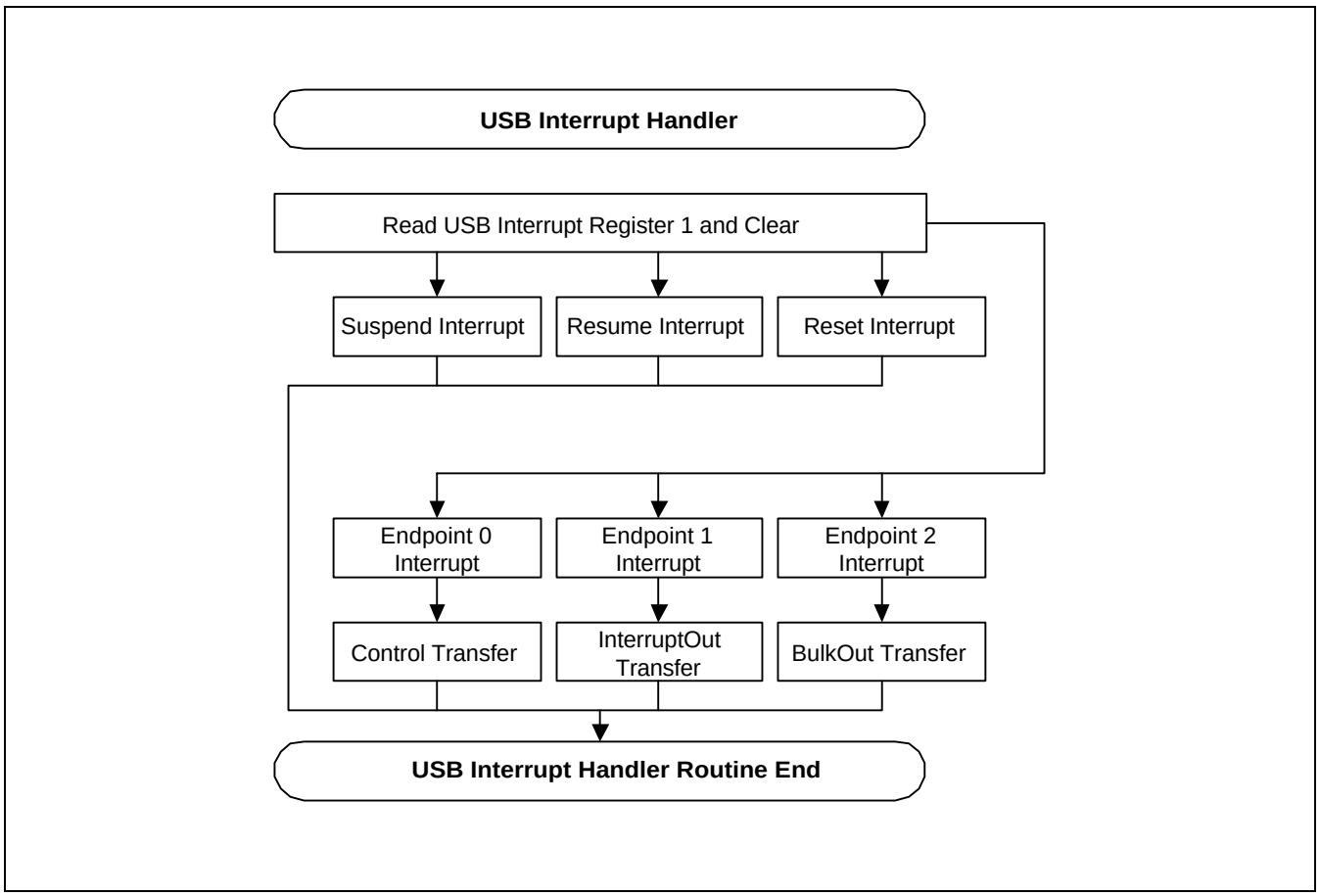


Figure 5-22. Concept Diagram for USB Interrupt Handler

DIAGNOSTIC CODE : USB INTIALIZE <Usb.c>

```
//=====================================================================//
// InitISR()                                                           //
//                                                                     //
// This is function for enable USB Interrupt                           //
//=====================================================================//

void InitISR(void){
      InitializationUSB();  // Initialize USB Register
      SysSetInterrupt(nUSB_INT, USB_IntHandler); // Register USB Interrupt
      Enable_Intr(nUSB_INT); // Enable USB Interrupt
}

//=====================================================================//
// InitializationUSB()                                                 //
//                                                                     //
// This function Initialise the USB device Register and Global Variable //
//=====================================================================//

void InitializationUSB(void){

      endpointZeroState = EP0_STATE_IDLE;
      //Clear FA.

      WriteUsbRegister(USB_FA_REGISTER,0x00);
      //Mode and Endian Setting
      WriteUsbRegister(USB_PM_REGISTER,0x0300);
      //Interrupt Enable
      WriteUsbRegister( USB_INT_REGISTER, 0x0000);
      WriteUsbRegister( USB_INTE_REGISTER, 0x041F);//Reset, Ep0~4 Enable
      //Write Start Address of EP0,1,3
      WriteUsbRegister( USB_E0SA_REGISTER, Ep0InOutdata);
      WriteUsbRegister( USB_E1SA_REGISTER, IntOutdata);
      WriteUsbRegister( USB_E3SA_REGISTER, BulkOutdata);
}

//=====================================================================//
// USB_IntHandler()                                                    //
//                                                                     //
// This function is Service Routine for USB Interrupt.                 //
// USB Interrupts are Endpoint Interrupt and H/W Status Interrupt.     //
//=====================================================================//
```

```
void USB_IntHandler(void){
      ULONG status;


      // Read USB Interrupt Regsiter
      ReadUsbRegister(USB_INT_REGISTER,status);
      WriteUsbRegister(USB_INT_REGISTER,0x00); // Interrupt Clear
      status &= 0x071f;

      if( status & 0x0100 ) {
            Print("\n [USB_Diag_Log]  : Suspend Mode\n");

            //Disable ENABLE_SUSPEND bit in PM register
            ClearUsbRegister(USB_PM_REGISTER, 0x01);
      }
      if( status & 0x0200 ) {
            Print("\n [USB_Diag_Log]  : Resume Mode \n");

            // Enable ENABLE_SUSPEND bit in PM register
            SetUsbRegister(USB_PM_REGISTER,0x01);
      }
      if( status & 0x0400 ) {
            Print("\n [USB_Diag_Log]  : Reset Mode \n");
            InitializationUSB();
      }
      if( status & 0x0001 ) { //EP0
            //Print("\n [USB_Diag_Log]  :  EP0 Control Transfer Interrupt \n");
            endpointZeroFunction();
      }
      if( status & 0x0004 ) { //EP2
            SetUsbRegister( USB_E2SC_REGISTER, 0x04);
      }
      if( status & 0x0002 ) { //EP1
            endpointInterruptOutFunction();
            SetUsbRegister( USB_E1SC_REGISTER, 0x0a0002);
      }
      if( status & 0x0010 ) { //EP4
            SetUsbRegister( USB_E4SC_REGISTER, 0x04);
      }
      if( status & 0x0008 ) { //EP3
            endpointBulkOutFunction();
            SetUsbRegister( USB_E3SC_REGISTER, 0x0a0002);
      }
}
```

USB CONTROL TRANSFER (ENDPOINT 0)

USB endpoint0 (Control Transfer) is special endpoint that can transmit and receive.

(1) Check Endpoint0 State

(2) If State is Transmit, go back to idle state. This state means that USB device send control data to host.

(3) If State is Receive, Call endpointZeroReceiver ( ). This state means that USB device received control data from host.

(4) If State is IDLE, Check Device Request Type of control data and change state of endpoint 0. This device request type is defined USB Spec1.1. After this, run DMA.

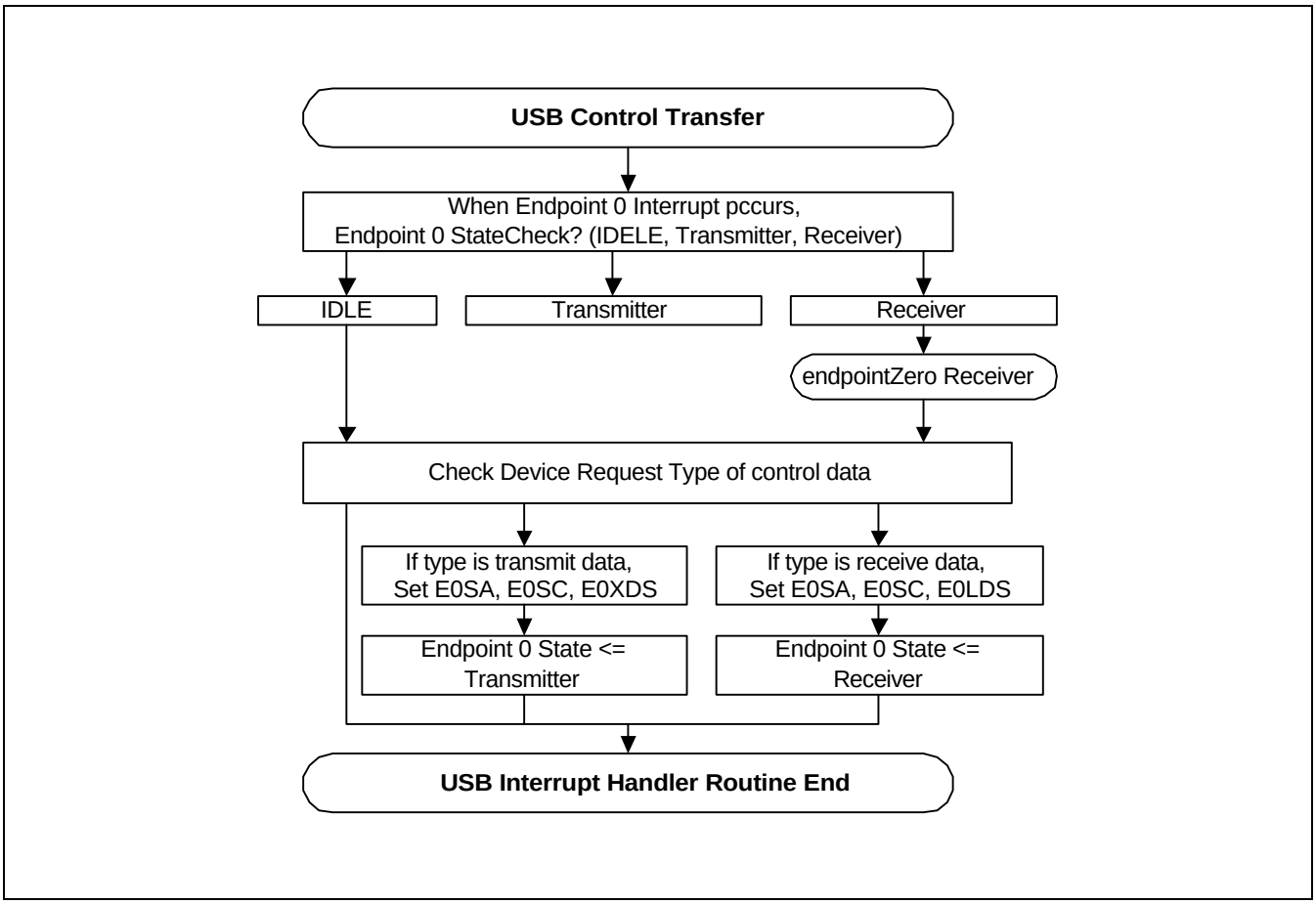


Figure 5-23. Concept Diagram for USB Control Transfer

DIAGNOSTIC CODE : USB CONTROL TRANSFER (ENDPOINT 0)

```
// Structure for Device Request Type ( USB Standard Spec1.1)
static struct DEVICE_REQUEST {
    UCHAR bmRequestType;  // Device Request offset 0
    UCHAR bRequest;       // Device Request offset 1
    UCHAR wValue_L;       // Device Request offset 2
    UCHAR wValue_H;       // Device Request offset 3
    UCHAR wIndex_L;       // Device Request offset 4
    UCHAR wIndex_H;       // Device Request offset 5
    UCHAR wLength_L;      // Device Request offset 6
    UCHAR wLength_H;      // Device Request offset 7
} DeviceRequest;

//=====================================================================//
// endpointZeroFunction()                                              //
//                                                                     //
// This is function for Control Transfer.                              //
// If it occurs endpoint0 interrupt, this function is called.          //
// This function check Device Request for Control Transfer type and    //
// call each other functions.                                          //
//=====================================================================//

void endpointZeroFunction(void){
      int i;

      ReadUsbRegister(USB_E0SC_REGISTER,E0SC_value);
      // EP0 CSR register status check

      if (E0SC_value & 0x00400000){
            Print(" Sent Stall \n");
//Set sent stall
            ClearUsbRegister(USB_E0SC_REGISTER, 0x00400000);
            endpointZeroState = EP0_STATE_IDLE;
      }
      if (E0SC_value & 0x00100000) {
            SetUsbRegister(USB_E0SC_REGISTER,0x00200000);
            Print(" Setup End \n");
            endpointZeroState = EP0_STATE_IDLE;
      }
      //TRAN_END_Int
      if (E0SC_value & 0x00000008) {
            Print(" Tran End \n");
            WriteUsbRegister( USB_E0SA_REGISTER, Ep0InOutdata);
            SetUsbRegister(USB_E0SC_REGISTER,0x00000010);//Tran End Clear
            endpointZeroState = EP0_STATE_IDLE;
      }
      // Device Request type check and Control Transfer, Receiver
      switch (endpointZeroState) {
            case EP0_STATE_IDLE:
```

```
// Read standard request!!
            if(E0SC_value & 0x00000002){
                        DeviceRequest.bmRequestType = *NCA(Ep0InOutdata);
                        DeviceRequest.bRequest = *Ep0InOutdata+1};
                        DeviceRequest.wValue_L = *(Ep0InOutdata+2);
                        DeviceRequest.wValue_H = *(Ep0InOutdata+3);
                        DeviceRequest.wIndex_L = *(Ep0InOutdata+4);
                        DeviceRequest.wIndex_H = *(Ep0InOutdata+5);
                        DeviceRequest.wLength_L = *(Ep0InOutdata+6);
                        DeviceRequest.wLength_H = *(Ep0InOutdata+7);

                              switch (DEVICE_bmREQUEST_TYPE(DeviceRequest)){
                              case STANDARD_TYPE:
                                    Print("\n MCU >> Standard Type Interrupt \n");
   StandardDeviceRequest();
                                    break;
                              case CLASS_TYPE:
                                    Print("\n MCU >>  Class Type Interrupt \n");
                                    break;
                              case VENDOR_TYPE:
                                    break;
                              case RESERVED_TYPE:
                                    break;
                              }
                        }
                        break;
            case EP0_STATE_TRANSFER:
                        Print(" Transfer OK\n");
                        endpointZeroState = EP0_STATE_IDLE;
                        break;
            case EP0_STATE_RECEIVER:
                        Print(" Receive OK \n");
                        endpointZeroReceiver();
                        break;
   }
   }
```

USB BULKOUT TRANSFER (ENDPOINT 1, 3)

USB BulkOut endpoint is endpoint that can receive. (endpoint1,3)

(1) Check endpoint1, 3 Interrupt.

(2) Store received data length in endpoint1, 3.

(3) Call endpoint BulkIn Function.

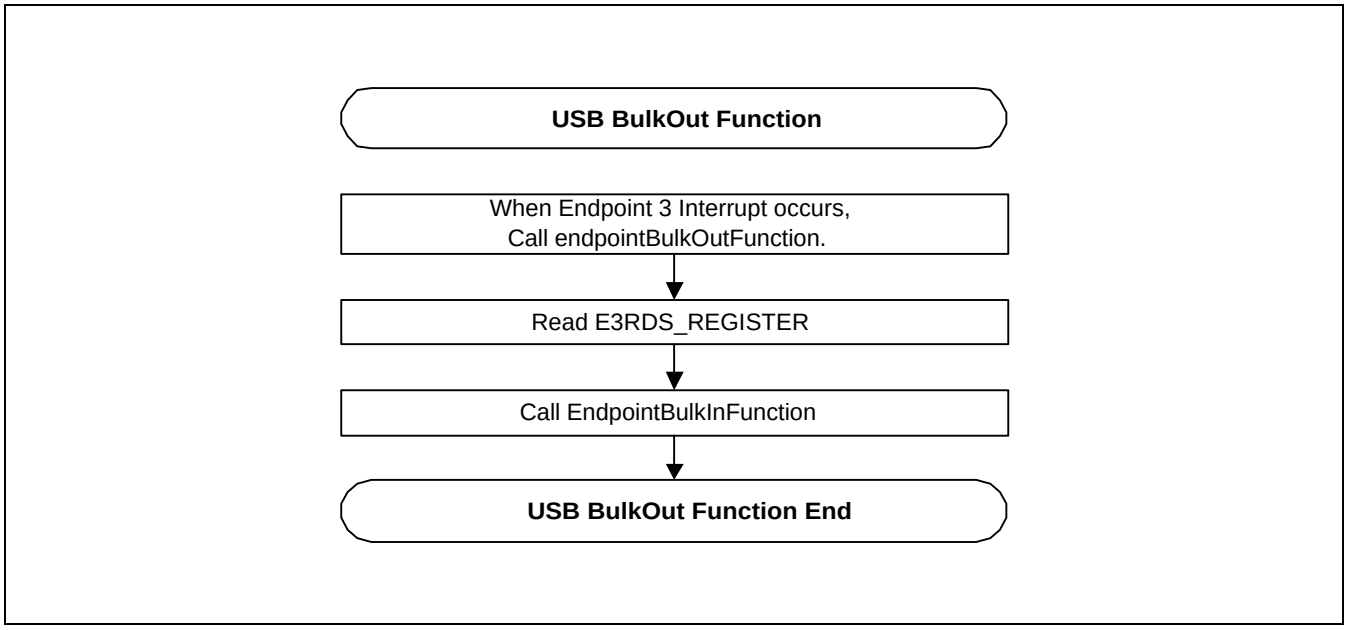


Figure 5-24. Concept Diagram for USB BulkOut Function

DIAGNOSTIC CODE : USB BULKOUT TRANSFER (ENDPOINT 1, 3)

```
//=====================================================================//
// endpointBulkOutFunction()                                          //
//                                                                     //
// This is function for Endpoint Three ( Bulk Out )function routine.   //
//=====================================================================//

void endpointBulkOutFunction(void)
{
      ReadUsbRegister(USB_E3RDS_REGISTER, BulkOutCnt );
      SetUsbRegister(USB_E4SC_REGISTER, 0x4000);//Zero End Enable
      endpointBulkInFunction(BulkOutdata, BulkOutCnt );
}
```

USB BULKIN TRANSFER (ENDPOINT 2, 4)

USB BulkIn endpoint is endpoint that can transmit. (endpoint2,4)

(1) Write the start address.(E4SA_REGISTER)

(2) Write the size of data.(E4TDS_REGISTER)

(3) Run DMA

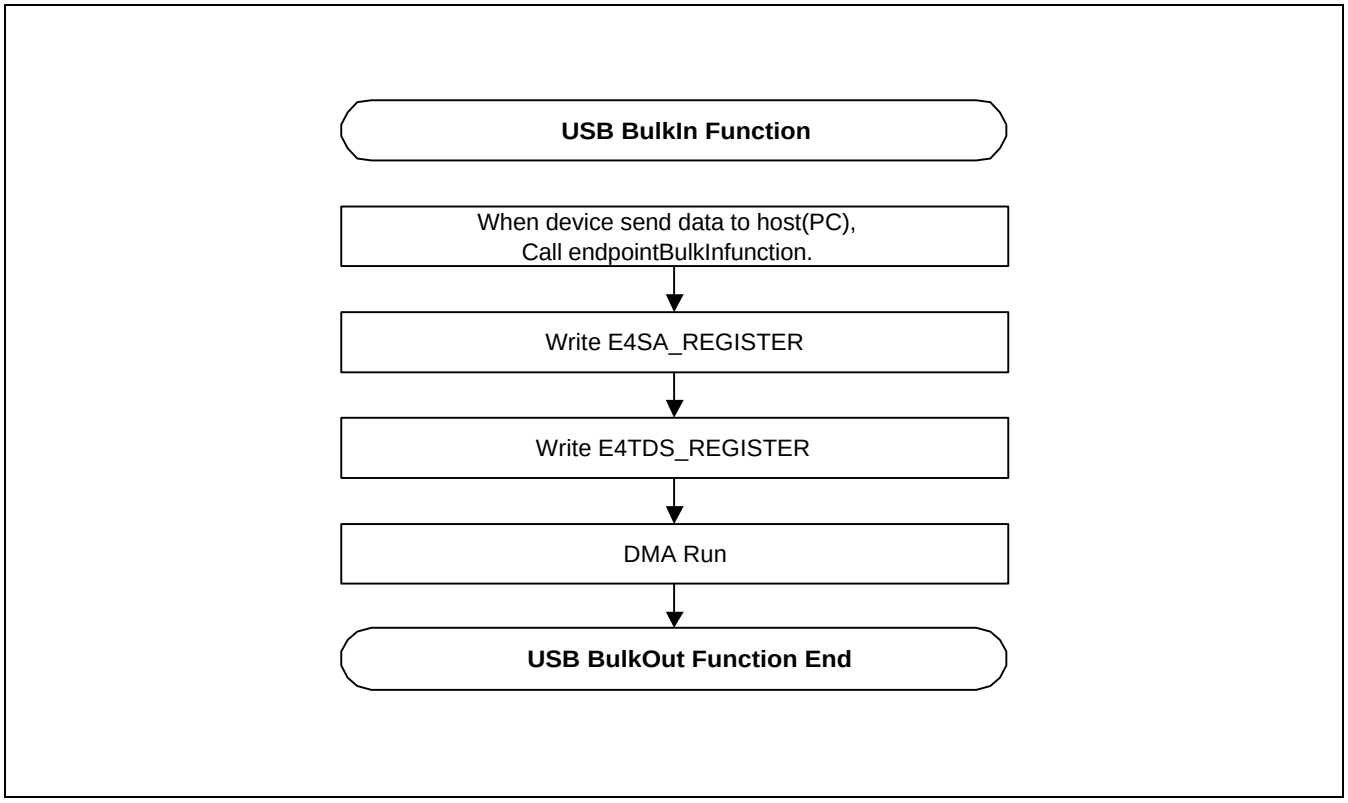


Figure 5-25. Concept Diagram for USB BulkIn Function

DIAGNOSTIC CODE : USB BULKIN TRANSFER (ENDPOINT 2, 4)

```
//=====================================================================//
// endpointBulkInFunction()                                            //
//                                                                     //
// This is function for Endpoint 4 ( Bulk In )function routine         //
//=====================================================================//
void endpointBulkInFunction( UCHAR* BulkIndata, ULONG BulkInLen )
{
      ReadUsbRegister(USB_E4SC_REGISTER, tempReg);
      if(tempReg & 0x08){
            WriteUsbRegister(USB_E4SA_REGISTER, BulkIndata);
            WriteUsbRegister(USB_E4TDS_REGISTER, BulkInLen);
            //Print("\nEP4 DMA Run");
            SetUsbRegister(USB_E4SC_REGISTER, 0x01);
      }else{
            Print("\n\n DMA Not ready");
            Print("\n cannot set EP4SA, EP4TDS, EP4SC(DMA_RUN)\n\n");
      }
}
```

<UsbFunc.C> (Continued)

```
                  // Normal Condition : Data Length = 0
            else if( IterationCnt == 0 && RemainDataCnt != 0 )
                break;

            // Set In Packet Ready Bit
            InReadUsbRegister(4,ADDR_IN_CSR1_REGISTER,data);
            data |= 0x01 ;
            InWriteUsbRegister(4,ADDR_IN_CSR1_REGISTER, data);

            IterationCnt--;
        }
    }
}
```

SAR (SEGMENT AND REASSEMBLY)

The S5N8947 SAR diagnostic code supports configuration and controlling SAR block. It includes managing connection memory and buffer pool also. The full contents are as follows :

- Configure SAR registers.
- Setup SAR queues (Done queue/Pool queue).
- Control connection memory.
- Transmit packet.
- Receive packet.
- Handle SAR interrupt.
- Flow chart of SAR Diagnosis.

This diagnostic code use a few data structures which provide effective solution for S5N8947 SAR control. For example, the structure '`CfgSARChan'` is composed of channel information members, and '`CfgSARregTable'` contains the SAR register information including the clock values and constitution of queue. For more these structures and other tables, refer to the header file, "sar.h".

With this diagnostic code, you can understand and control the S5N8947 SAR more easily.

CONFIGURE SAR REGISTERS

To open a connection and transmit or receive data through the S5N8947 SAR, the following operation has to be performed.

SAR Register Initialization

- Setup clock registers : Set SAR clock and UTOPIA clock as your system.
- Setup connection memory registers : Initialize the base address and size of SAR connection memory.
- Setup SAR queue registers : Allocate the SAR Tx done queue/Rx done queue/Rx pool queues.

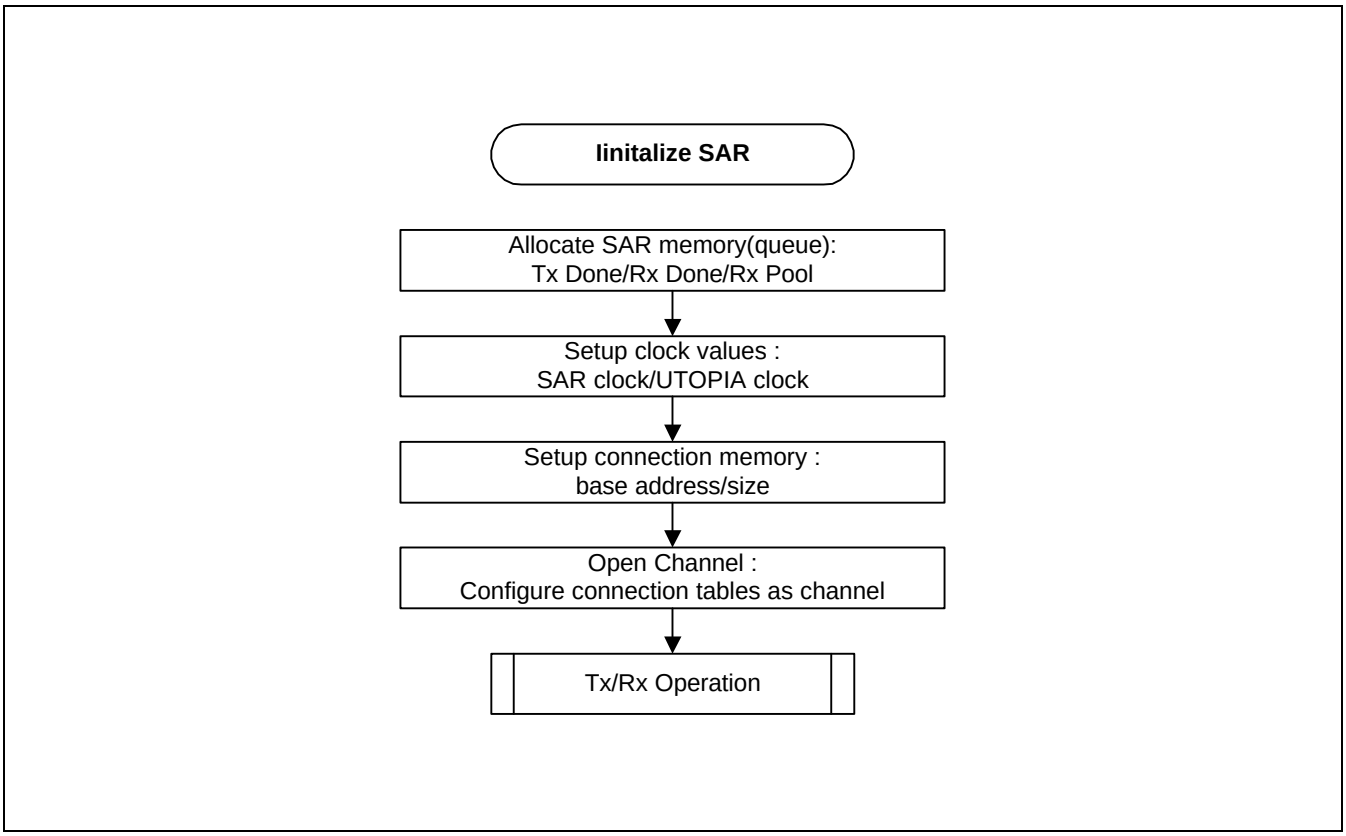


Figure 5-26. SAR Initialization

Allocation memory, opening channel and Tx/Rx operation will be described later in this chapter.

SETUP SAR QUEUES

There are three types of queue in the S5N8947 SAR.

- **Tx Done Queue** : has the addresses of Tx buffer descriptors that have been transmitted.
- **Rx Done Queue** : has the addresses of Rx buffer descriptors that have been received.
- **Rx Pool Queue** : has the addresses of Rx buffer pools.

Transmit Done Queue

The S5N8947 SAR writes the addresses of Tx buffer descriptors that have been transmitted in this queue. At the same time, the SAR requests Tx_Done interrupt. SAR driver reads this address in the SAR Done interrupt service routine. Detailed operation of interrupt service routine will be described later in this chapter.

— The entry size of this queue is one of following values : `256, 1024, 4096, 16384`
The **TX_DONE_SIZE** register has this size value.

— The **TX_DONE_ADDR** register has the base address of this queue.

Receive Done Queue

The S5N8947 supports two receive done queues, and writes the addresses of Rx buffer descriptors that have been received in these queues. The Rx done queue 0 and queue 1 indicates address of the received buffer descriptor through receive pool 0 and pool 1 respectively. At the same time, the SAR requests Rx_Done interrupt if a payload data is received at Rx pool via valid channel. SAR driver reads this address in the SAR Rx_Done interrupt service routine. Detailed operation of interrupt service routine will be described later in this chapter.

— The size of this queue is one of following values : `256, 1024, 4096, 16384`
The **RX_DONEx_SIZE** register has this size value of each queue.

— The **RX_DONEx_ADDR** register has the each base address of these queues.

Receive Pool Queue

The S5N8947 supports four receive pool queues, and the pool 0,2 and pool 1,3 operate separately. When a packet arrives at SAR, the SAR starts copying data to pool 0 or pool1. Which pool is used is determined at AAL connection table of each channel. If the size of payload data is longer than buffer size of pool0 or pool1, the rest data is saved to pool 2 or pool 3 respectively. Details of receive operation will be described later in this chapter.

— The size of each pool queue is one of following values : `256, 1024, 4096, 16384`
The **RX_POOLx_SIZE** register has this size value of each queue.

— The buffer size of each pool queue is one of `128 * 2`^{n} bytes. (where n = 0 ~ 9). Generally the buffer size of pool 0 and pool 1 is smaller than pool 2 and pool3. The **RX_POOLx_SIZE** register has this buffer size value of each queue.

— The **RX_POOLx_ADDR** register has the each base address of these queues.

DIAGNOSTIC CODE : SETUP VARIOUS SAR QUEUES

```
//=====================================================================//
// SARAllocMem                                                         //
//                                                                     //
// Allocate SAR Tx/Rx PoolQ and DoneQ area.                            //
// These size of each queue are defined in the "sar.h"                 //
//=====================================================================//
UINT SARAllocMem(void)
{
    UINT ptrAllocMem, *pRxPoolQ, i, SARmemStart, *ptrMem;

    if(UserSarMemBase == 0)
        // Avoid S5N8947SAR H/W limitiation.
        ptrAllocMem = SARmemStart = ((HeapBase + 0xf) & ~0xf) + 8;
    else
        ptrAllocMem = SARmemStart = UserSarMemBase;

    // Init TxDONE Queue.
    strCFG_SAR_REG.TxDoneQ_Addr     = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    strCFG_SAR_REG.TxDoneQ_Size     = DFLT_QUE_SIZE;
    strCFG_SAR_REG.TxDoneQ_Offset   = DFLT_QUE_OFFSET;
    ptrAllocMem += (strCFG_SAR_REG.TxDoneQ_Size * 4);

    // Init RxDone Queue.
    strCFG_SAR_REG.RxDoneQ_0_Size   = DFLT_QUE_SIZE;
    strCFG_SAR_REG.RxDoneQ_1_Size   = DFLT_QUE_SIZE;

    strCFG_SAR_REG.RxDoneQ_0_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxDoneQ_0_Size * 4);

    strCFG_SAR_REG.RxDoneQ_1_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxDoneQ_1_Size * 4);

    strCFG_SAR_REG.RxDoneQ_Offset   = DFLT_QUE_OFFSET;

    // Init RxPool Queue.
    strCFG_SAR_REG.RxPoolQ_0_Size   = DFLT_QUE_SIZE;
    strCFG_SAR_REG.RxPoolQ_1_Size   = DFLT_QUE_SIZE;
    strCFG_SAR_REG.RxPoolQ_2_Size   = DFLT_QUE_SIZE;
    strCFG_SAR_REG.RxPoolQ_3_Size   = DFLT_QUE_SIZE;
    strCFG_SAR_REG.RxPoolQ_Offset   = DFLT_QUE_OFFSET;

    strCFG_SAR_REG.RxPoolQ_0_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxPoolQ_0_Size * 4);

    strCFG_SAR_REG.RxPoolQ_1_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxPoolQ_1_Size * 4);

    strCFG_SAR_REG.RxPoolQ_2_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxPoolQ_2_Size * 4);
```

```
    strCFG_SAR_REG.RxPoolQ_3_Addr   = (ptrAllocMem | SAR_ADRVALID \
                                      | NON_CACHE_FLAG);
    ptrAllocMem += (strCFG_SAR_REG.RxPoolQ_3_Size * 4);

    // Init RxPool Buffer and Setup RxPool Queue.
    strCFG_SAR_REG.RxPoolQ_0_BufSize = DFLT_RXPOOLBUFSIZE_LOW;
    strCFG_SAR_REG.RxPoolQ_1_BufSize = DFLT_RXPOOLBUFSIZE_LOW;
    strCFG_SAR_REG.RxPoolQ_2_BufSize = DFLT_RXPOOLBUFSIZE_HIGH;
    strCFG_SAR_REG.RxPoolQ_3_BufSize = DFLT_RXPOOLBUFSIZE_HIGH;

    // Clear Tx/Rx Done Queue Area.
    ptrMem = (UINT *)SARmemStart;
    for(ptrMem = (UINT *)SARmemStart; (UINT)ptrMem < ptrAllocMem; ptrMem++)
        *ptrMem = 0;

    // Connect Rx Pool & Buff
    pRxPoolQ = (UINT *)strCFG_SAR_REG.RxPoolQ_0_Addr;

    for(i=0; i<strCFG_SAR_REG.RxPoolQ_0_Size; i++)  {
        *pRxPoolQ = SwapWord(ptrAllocMem | SAR_ADRVALID | NON_CACHE_FLAG);
        pRxPoolQ++;
        ptrAllocMem += strCFG_SAR_REG.RxPoolQ_0_BufSize;
    }

    pRxPoolQ = (UINT *)strCFG_SAR_REG.RxPoolQ_1_Addr;
    for(i=0; i<strCFG_SAR_REG.RxPoolQ_1_Size; i++)  {
        *pRxPoolQ = SwapWord(ptrAllocMem | SAR_ADRVALID | NON_CACHE_FLAG);
        pRxPoolQ++;
        ptrAllocMem += strCFG_SAR_REG.RxPoolQ_1_BufSize;
    }

    pRxPoolQ = (UINT *)strCFG_SAR_REG.RxPoolQ_2_Addr;
    for(i=0; i<strCFG_SAR_REG.RxPoolQ_2_Size; i++)  {
        *pRxPoolQ = SwapWord(ptrAllocMem | SAR_ADRVALID | NON_CACHE_FLAG);
        pRxPoolQ++;
        ptrAllocMem += strCFG_SAR_REG.RxPoolQ_2_BufSize;
    }

    pRxPoolQ = (UINT *)strCFG_SAR_REG.RxPoolQ_3_Addr;
    for(i=0; i<strCFG_SAR_REG.RxPoolQ_3_Size; i++)  {
        *pRxPoolQ = SwapWord(ptrAllocMem | SAR_ADRVALID | NON_CACHE_FLAG);
        pRxPoolQ++;
        ptrAllocMem += strCFG_SAR_REG.RxPoolQ_3_BufSize;
    }

    // Tx Buffer Descriptor Area.
    for(i=0; i< TXBUFDESC_NUM; i++)  {
        gTxBufDescArea[i] = ptrAllocMem;
        ptrAllocMem += BUFDESC_SIZE;
    }

    if(ptrAllocMem > (S5N8947_DRAM_BASE + S5N8947_DRAM_SIZE))
        return FAIL;

    return SUCCESS;
}
```

CONTROL CONNECTION MEMORY

The S5N8947 SAR connection memory is composed of followings :

- 1/Rate Lookup Table
- VP Lookup Table
- UBR Schedule Table
- CBR Schedule Table
- Cell Buffers
- Scheduler Connection Table
- AAL Connection Table
- SAR Connection Table

Initialize Connection Memory

The S5N8947 SAR can use internal 8K bytes SRAM or external memory (DRAM or SRAM) as connection memory. To use internal SRAM as connection memory, set EXT_ONLY bit of **CONFIGURATION** register to 0, and to use external memory, set this bit to 1. When using external memory, the base address of connection memory has to be configured at **EXT_CMBASE** register.

If you use internal SRAM, It's recommended the base address and size of each table to be set as default values. In this case, number of channels that can be opened is limited to 32 channels. The number of channel is dependent on size of connection tables, which are scheduler, aal, sar connection table.

After initializing location of connection memory, SAR driver would setup CAM Contents (or VP Lookup table), Scheduler connection table, AAL connection table, SAR connection table when new channel is opened except 1/Rate Lookup table.

Setup 1/Rate Lookup Table

The following code shows the setup of 1/Rate Lookup table. In this code you would be change the `SAR8947_MAX_BITRATE` value as your system.

DIAGNOSTIC CODE : SETUP 1/RATE LOOKUP TABLE

```
//=====================================================================//
// GenOneOfRateTbl                                                     //
//                                                                     //
// Setup a 1/rate look up table.                                       //
//=====================================================================//
void GenOneOfRateTbl(void)
{
    ULONG PCR, MaxRate;
    double PCR_Cell, power_p2, power_p2val, mantissa;
    int i, n;

    PCR_Cell = SAR8947_MAX_BITRATE * 1e6;
    // de-rate the PCR die to sonet.
    // PCR_Cell = (long double)(PCR_Cell * 260.0/270.0);

    // compute ATM cells per second.
    PCR_Cell = PCR_Cell / (53.0 * 8.0);
    power_p2 = floor(log(PCR_Cell) / log(2.0));
    power_p2val = pow(2.0, power_p2);
    mantissa = PCR_Cell / power_p2val;

    // Create the ABR rate format for the PCR
    MaxRate = power_p2 * pow(2.0, 9);
    MaxRate += (mantissa - 1.0) * 512.0;
    if(PCR_Cell)
        MaxRate += 1.0 * pow(2.0, 14);
    for(n=0.0, i=0; i<DFLT_RATE_LKUPTBL_SIZE; i++, n += 1.0)
        RateLookupTbl[i] = (UINT)(16.0 * mantissa * 512.0 * power_p2val \
                                  / (512.0 + n));
}
```

Open New Channel

Whenever open new channel, the CAM (or VP lookup table) and Connection tables have to be configured as VPI , VCI, PORT value and other conditions.

— **SAR CAM contents** (each table is 8 byte length)

Address : `CAM_VPIVCIx = SAR BASE ADDRESS + 0100h + Connection count * 8`

`CAM_CNx = SAR BASE ADDRESS + 0100h + Connection count * 8 + 4`

`(Where x is connection count)`

SAR refer CAM_VPIVCIx/CAM_CNx for conversion between Scheduler/AAL/SAR connection number and VPI/ VCI/ PORT (Same function can be achieved by VP lookup table).
For full information about SAR CAM, refer to user's manual.

— **Setup Scheduler Connection Table** (each table is 8 word length)

Address : `SCH_CONNTBL_BASE + ((Scheduler connection number*8)+Word Offset) * 4`

For full information about Scheduler connection table, refer to user's manual.

— **Setup AAL Connection Table** (each table is 8 word length)

Address : `AAL_CONNTBL_BASE + ((AAL connection number*8)+Word Offset) * 4`

With this table the buffer pool type and packet type is determined for reception.
For full information about AAL connection table, refer to user's manual.

— **Setup SAR Connection Table** (each table is 8 word length)

Address : `SAR_CONNTBL_BASE + ((SAR connection number*8)+Word Offset) * 4`

For full information about SAR connection table, refer to user's manual.

DIAGNOSTIC CODE : SETUP VP LOOKUP, SCHEDULER/AAL/SAR CONNECTION TABLE

```
//====================================================================//
// SetNewConn                                                         //
//                                                                    //
// Initialize New Connection with default setting.                    //
//====================================================================//
int SetNewConn(UINT nVPI, UINT nVCI, UINT phy_port, UINT Qos, UINT nPCR, UINT nMBS,
UINT nSCR, UINT nAAL)
{
      UINT  ChanNum, i, VCMaskBit = 0;
      UINT  CONNPATTERN;

      if(strCFG_SAR_REG.CAM !=SAR_CAM_ENABLE){  // Using VP Loockup Table
            if(strCFG_CONNMEM.ChanSizePerVP <= nVCI) return OVERSIZE;

            if(strCFG_SAR_REG.UTO_LEVEL ==UTOPIA_MODE_Level1){
                if((ChanNum = strCFG_CONNMEM.ChanSizePerVP *nVPI + nVCI) >
                                                          \CHAN_SIZE)
                    return OVERSIZE;
            }else{
                if((ChanNum = strCFG_CONNMEM.ChanSizePerVP *(phy_port +(nVPI<<3))\
                                                          + nVCI) > CHAN_SIZE)
                    return OVERSIZE;
            }
            for(i=0; i<OpenChanNum; i++)  {
              if(OpenChan[i] == ChanNum)
                 return OPENCHAN;
            }
      }else{                                    // Using CAM
            CONNPATTERN =(nVPI<<15) + (nVCI<<3) + (phy_port<<0);
            // if Channel Number >= CHAN_SIZE, return OVERSIZE
            if(OpenChanNum >=CHAN_SIZE) return OVERSIZE;
            for(i=0; i<OpenChanNum; i++) {
                  ChanNum =OpenChan[i];
                  if(strCFG_SAR_CHAN[ChanNum].CONNPATTERN ==CONNPATTERN)
                        return OPENCHAN;
            }
            //
            // Add the Channel Number Searching Algorithm here
            //
            ChanNum =OpenChanNum;
      }

      // Make New Connection Parameters
      OpenChan[OpenChanNum] =ChanNum;
      strCFG_SAR_CHAN[ChanNum].VPI = nVPI;
      strCFG_SAR_CHAN[ChanNum].VCI = nVCI;
      strCFG_SAR_CHAN[ChanNum].AAL = nAAL;
      strCFG_SAR_CHAN[ChanNum].PTI = DFLT_PTI;
      strCFG_SAR_CHAN[ChanNum].PORT =phy_port;
      strCFG_SAR_CHAN[ChanNum].CONNPATTERN =CONNPATTERN; //to Search Same Connection
      OpenChan[OpenChanNum + 1] = 0xff;
```

```
// When CHAN_SIZE ==32, Initialize CAM Table.
// Else,                    Initialize VP Lookup Table
strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnType = SCHCONNTYPE;
if(strCFG_SAR_REG.CAM ==SAR_CAM_ENABLE)
{
      if(SCHCONNTYPE==VPLT_VPCONN){       // VP Scheduling
          strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnNum  = phy_port;
      strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.AAL_SARConnNum = ChanNum;
      }else{                                                  // VC Scheduling
          strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnNum  = ChanNum;
      strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.AAL_SARConnNum = ChanNum;
      }
}
else
{
    switch(strCFG_CONNMEM.ChanSizePerVP)  {
      case 16   : VCMaskBit = VPLT_VCBITMASK_4; break;
      case 32   : VCMaskBit = VPLT_VCBITMASK_5; break;
      case 64   : VCMaskBit = VPLT_VCBITMASK_6; break;
      case 128  : VCMaskBit = VPLT_VCBITMASK_7; break;
      case 512  : VCMaskBit = VPLT_VCBITMASK_9; break;
      case 1024 : VCMaskBit = VPLT_VCBITMASK_10;      break;
    }
    strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.VC_BitsMask = VCMaskBit;
    strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnNum= ChanNum-nVCI;
    strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.AAL_SARConnNum= ChanNum-nVCI;
}

OpenChanNum++;

// Initialize Scheduler connection Table.
if(strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnType == VPLT_VCCONN) {
  strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.SCH_CONN_NUM = ChanNum;
  strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.ConnLevel = SCHCT_VCCONN;
}
else  {
      //
      // Add the VP connection algorithm here when CHAN_SIZE == 32
      //
  strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.SCH_CONN_NUM = \
          strCFG_SAR_CHAN[ChanNum].strCFG_VPLOOKUP_TBL.SchConnNum;
  strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.ConnLevel = SCHCT_VPCONN;
}
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.PCR = nPCR;     //DFLT_PCR;
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.SchType = Qos;  //SCHCT_CBR;
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.MBS = nMBS;
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.SCR = nSCR;
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.AAL_ConnNum = ChanNum;
strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.PHY_INFO = phy_port;
```

```
      // Initialize AAL connection Table.
      strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.AAL_CONN_NUM =ChanNum;
      strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.VP = nVPI;
      strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.VC = nVCI;
      strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.PHY_PORT = phy_port;
      // AAL0, CRC10 Cell always use Buffer0/2 by H/W
      if((nAAL==AALCT_PKT_AAL0) || (nAAL==AALCT_PKT_CRC10)){
          strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.RxBuffType =AALCT_BUFF_0AND2;
      }else{
          strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.RxBuffType =      \
                      ((ChanNum&0x1)==0) ? AALCT_BUFF_0AND2 : AALCT_BUFF_1AND3;
      }
      strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.RxPktType = nAAL;

    strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.PayLoadOffset = BUFDESC_SIZE+\
                                                    strCFG_SAR_REG.TRXALIGN;
    strCFG_SAR_CHAN[ChanNum].strCFG_AAL_CONNTBL.SCH_ConnNum = \
                  strCFG_SAR_CHAN[ChanNum].strCFG_SCH_CONNTBL.SCH_CONN_NUM;

    // Initialize SAR connection Table.
    strCFG_SAR_CHAN[ChanNum].strCFG_SAR_CONNTBL.SAR_CONN_NUM = ChanNum;
    strCFG_SAR_CHAN[ChanNum].strCFG_SAR_CONNTBL.SEG_STATUS = SARCT_SEG_EN;

    return OK;
}
```

TRANSMIT PACKET

To send a packet, transmit buffer descriptor has to be configured as follows.

- If there are another packet descriptors in this chain, load the address and set the valid bit to 1
- Select packet type
- Write the length of buffer which links to this buffer descriptor
- Write cell header information including GFC, VPI, VCI, PTI and CLP
- Link the address of payload data to transmit

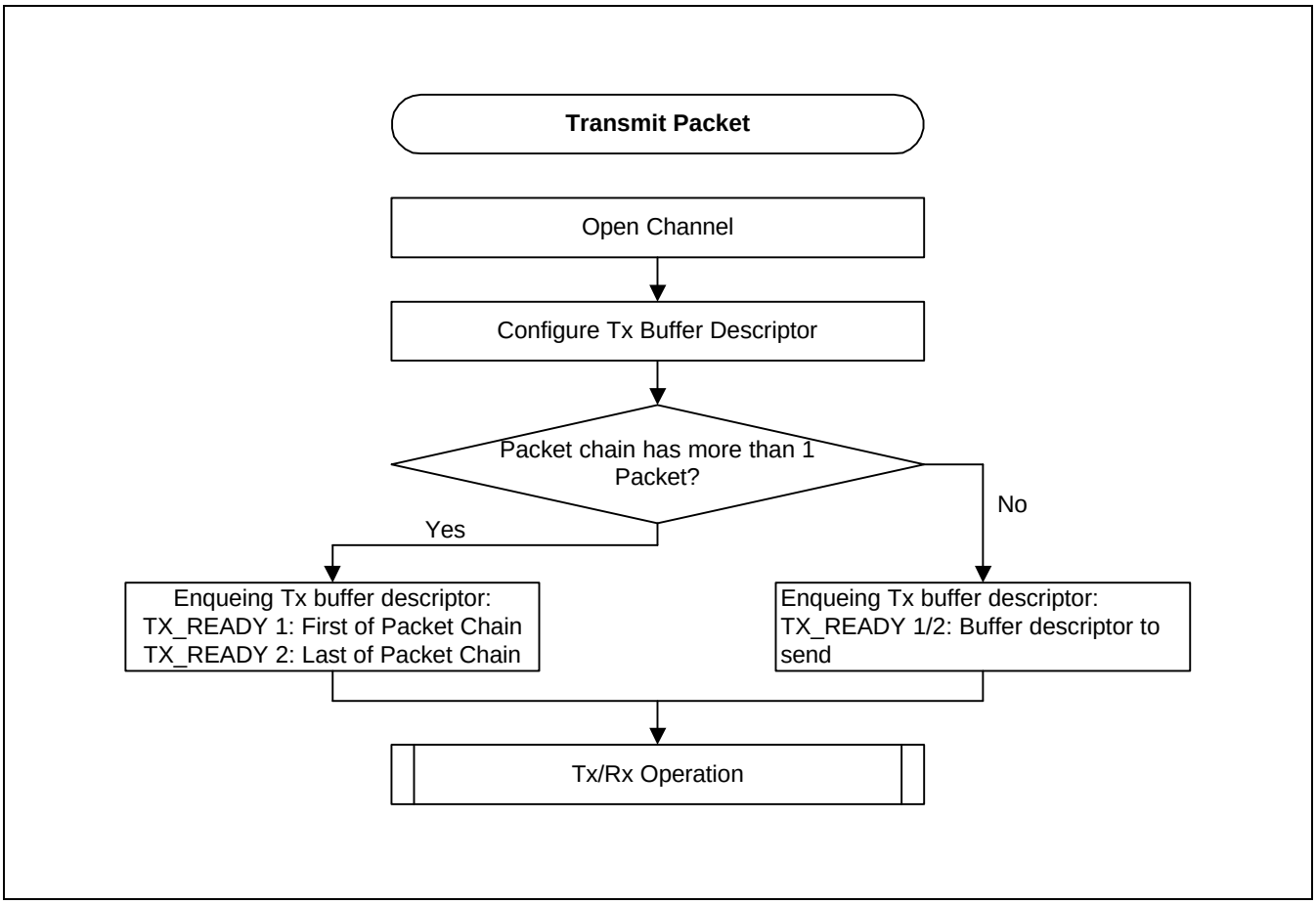


Figure 5-27. SAR Transmit Packet

DIAGNOSTIC CODE : SEND PACKET

```
//=====================================================================//
// SendPacket                                                          //
//                                                                     //
// Enqueing the Tx buf descriptor into the Tx_Ready register.          //
//=====================================================================//
void SendPacketSAR(UINT ChanNum, UINT PktSize, ULONG PktPtn, UINT PktCnt)
{
    strTxBufDescriptor *pTxBufDesc;
    ULONG NextBufDesc = 0;
    int i, j, Cnt;

      .......

    //-----------------------------------------------------------------//
    // Make Tx Buffer Descriptor Link.                                 //
    //-----------------------------------------------------------------//
    pTxBufDesc =(strTxBufDescriptor *)gWSarTxDesc;
    NextBufDesc = (ULONG) gWSarTxDesc;

    for(i=0; i<PktCnt; i++)  {
        // Link payload address to Tx buffer descriptor..
        pTxBufDesc = (strTxBufDescriptor *)(NextBufDesc & ~SAR_ADRVALID);

        // TxBufDescArea[0] : First.
        NextBufDesc =(ULONG)(pTxBufDesc->NEXT_DESC) | (SAR_ADRVALID);

        // [ NEXT_DESC ]
        if(i == (PktCnt-1))
            pTxBufDesc->NEXT_DESC=(strTxBufDescriptor *)\
                                  ((ULONG)NextBufDesc & ~SAR_ADRVALID);
        else
            pTxBufDesc->NEXT_DESC = (strTxBufDescriptor *)(ULONG)NextBufDesc;

        // [ PAYLOAD_ADDR ]
        if(PktPtn <= 0x100){
        // If PktPtn >= 0x100, transmit TxPacket_1/2/3 contents as data
            switch(i%3)  {
                  case 0 : if((i ==0) && ( strCFG_SAR_REG.TRXALIGN < 4))
                              pTxBufDesc->PAYLOAD_ADDR=(ULONG)\
                                    (&TxPacket_1[0] +  strCFG_SAR_REG.TRXALIGN);
                           else
                              pTxBufDesc->PAYLOAD_ADDR = (ULONG)(&TxPacket_1[0]);
                                    break;
                  case 1 : pTxBufDesc->PAYLOAD_ADDR = (ULONG)(&TxPacket_2[0]);
                                    break;
                  case 2 : pTxBufDesc->PAYLOAD_ADDR = (ULONG)(&TxPacket_3[0]);
                                    break;
              }
        } else{
        // If PktPtn is larger than 0x100, PktPtn is used as address of data
            pTxBufDesc->PAYLOAD_ADDR = (ULONG)(PktPtn +  strCFG_SAR_REG.TRXALIGN);
        }
```

```
    // [ PKT_STATUS ]
    if(PktCnt == 1) {
        if((strCFG_SAR_CHAN[ChanNum].PTI ==4)||\
                    (strCFG_SAR_CHAN[ChanNum].PTI ==5) || \
                    (strCFG_SAR_CHAN[ChanNum].PTI ==6)){            // OAM Cell
               pTxBufDesc->PKT_STATUS = BD_PKT_CRC10 |              \
                    (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
           }else if(strCFG_SAR_CHAN[ChanNum].AAL == AALCT_PKT_AAL0){ //AAL0 Cell
               pTxBufDesc->PKT_STATUS = BD_PKT_AAL0 |               \
                    (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
           }else if(strCFG_SAR_CHAN[ChanNum].AAL== AALCT_PKT_CRC10){//CRC10 Cell
               pTxBufDesc->PKT_STATUS = BD_PKT_CRC10 |              \
                    (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
           }else{                                                   // Complete Packet.
               pTxBufDesc->PKT_STATUS = BD_PKT_AAL5_COMP |          \
                    (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
           }
    } else if( i == (PktCnt - 1)) {                                 // End Packet.
        pTxBufDesc->PKT_STATUS = BD_PKT_AAL5_END |                  \
              (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
    }else {                                                         //Start or Middle Packet.
        pTxBufDesc->PKT_STATUS = BD_PKT_AAL5_STPMID |               \
              (strCFG_SAR_CHAN[ChanNum].PORT << BD_PHY_SFT);
    }

    // [ LENGTH ]
    switch(i%3)  {
        case 0:
                if(PktCnt !=1){
                 if(i ==0) pTxBufDesc->LENGTH = 2048- strCFG_SAR_REG.TRXALIGN;
                 else    pTxBufDesc->LENGTH = 2048;
                }else pTxBufDesc->LENGTH = PktSize- strCFG_SAR_REG.TRXALIGN;
                break;

        case 1: if(PktCnt !=2) pTxBufDesc->LENGTH = 2048;
                else         pTxBufDesc->LENGTH = PktSize -2048;
                break;
        case 2: pTxBufDesc->LENGTH = PktSize-4096;
                break;
        default:
                ASSERT(0);
                break;

    }

    // [ CELL_HEAD ]
    pTxBufDesc->CELL_HEAD = (strCFG_SAR_CHAN[ChanNum].VPI << BD_VPI_SFT) | \
                            (strCFG_SAR_CHAN[ChanNum].VCI << BD_VCI_SFT) | \
                            (strCFG_SAR_CHAN[ChanNum].PTI <<BD_PTI_SFT);
}
```

```
    //-------------------------------------------------------------------//
    // Enqueueing : Load TxBufDesc address to Tx Ready registers.      //
    //-------------------------------------------------------------------//
      for(j=0; j<100; j++){
            if((nSAR_TXREADY1&0x80000000) ||(nSAR_TXREADY2&0x80000000)) continue;
            break;
      }
      if((nSAR_TXREADY1&0x80000000) ||(nSAR_TXREADY2&0x80000000)){
            Print("\nTx Error - TX_READY (READY_DONE_BAR ==TRUE)");
            ASSERT(0);
      }

      nSAR_TXREADY1 = (ULONG)gWSarTxDesc | SAR_ADRVALID;
      nSAR_TXREADY2 = (ULONG)pTxBufDesc | SAR_ADRVALID;

      gWSarTxDesc =pTxBufDesc->NEXT_DESC;
}
```

RECEIVE PACKET

The S5N8947 SAR reassembles the received cells into packets. And the payload data is placed receive buffer pool 0 or pool 1 at first. Which pool is selected is determined by AAL connection table of the channel. If the data being received overflows their buffer pool 0 or 1, the rest data is placed to pool 2 or pool 3 respectively.

After reassemble operation is done, the address of first buffer descriptor from received packets is written to Rx Done queue and SAR requests Rx interrupt. The SAR writes the address into Rx done queue in order, setting the valid bit to 1.

To fetch the received data in the Rx interrupt service routine, SAR driver has to recognize the next valid queue location in the queue. The S5N8947 SAR driver calculates this address as follows:

Rx buffer descriptor = [Base address of Rx Done Queue + Received Packet Count]

For full receive operation, refer to following flow chart.

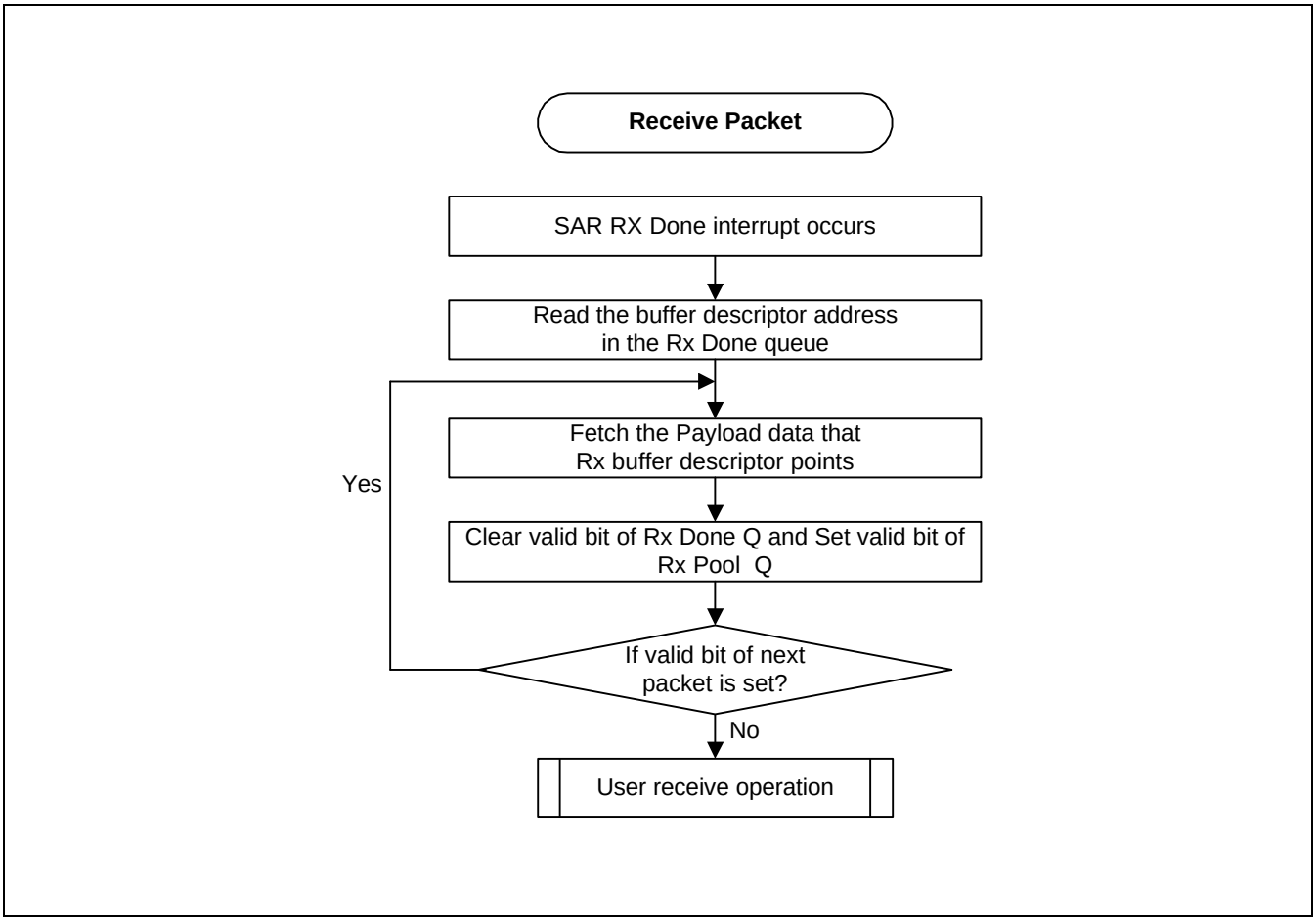


Figure 5-28. SAR Receive Packet

DIAGNOSTIC CODE : RECEIVE PACKET IN THE INTERRUPT SERVICE ROUTINE

```
// Rx Done Interrupt.
   if(ReadSARstatus & INTSTAT_RXDONE0)  {
      SARRxDONEOK++;

      PoolPat = 0;
      #if(DebugFlag_SAR_ISR == YES)
      Print("\n [SARisr] Interrupt - Rx Done 0");
      #endif
      while(1)  {
          RxDonePkt = (ULONG *)strCFG_SAR_REG.RxDoneQ_0_Addr + RxDoneQue0_Index;
          if(!(*RxDonePkt & SAR_ADRVALID))
              break;
          UserDefinedService((strRxBufDescriptor *)(*RxDonePkt), PoolPat);

          RxDoneQue0_Index++;
          if(RxDoneQue0_Index == DFLT_RXDONEQ_0_SIZE)
              RxDoneQue0_Index = 0;

          // Clear SAR address valid bit to release buffer descriptor of
          // Rx done queue.
          *RxDonePkt &= ~SAR_ADRVALID;

          // Rx Pool 0 release.
          RxPoolArea = (ULONG *)strCFG_SAR_REG.RxPoolQ_0_Addr + RxPoolQue0_Index;
          *RxPoolArea |= SAR_ADRVALID;
          RxPoolQue0_Index++;
          if(RxPoolQue0_Index == DFLT_RXPOOLQ_0_SIZE)
              RxPoolQue0_Index = 0;
    }    }

  if(ReadSARstatus & INTSTAT_RXDONE1)  {
      SARRxDONEOK++;

      PoolPat = 1;
      while(1)  {
          RxDonePkt = (ULONG *)strCFG_SAR_REG.RxDoneQ_1_Addr + RxDoneQue1_Index;
          if(!(*RxDonePkt & SAR_ADRVALID))
              break;
          UserDefinedService((strRxBufDescriptor *)(*RxDonePkt), PoolPat);
          RxDoneQue1_Index++;
          if(RxDoneQue1_Index == DFLT_RXDONEQ_1_SIZE)
              RxDoneQue1_Index = 0;

          // Clear SAR address valid bit to release buffer descriptor of
          // Rx done queue.
          *RxDonePkt &= ~SAR_ADRVALID;

          // Rx Pool 1 release.
          RxPoolArea = (ULONG *)strCFG_SAR_REG.RxPoolQ_1_Addr + RxPoolQue1_Index;
          *RxPoolArea |= SAR_ADRVALID;
          RxPoolQue1_Index++;
          if(RxPoolQue1_Index == DFLT_RXPOOLQ_1_SIZE)
               RxPoolQue1_Index = 0;
  }    }
```

HANDLING INTERRUPT

The S5N8947 has two types of SAR interrupt, Done and Error, that are requested by various source. The full SAR interrupt sources are described in the User's manual.

The following flow chart shows the handling SAR interrupt.

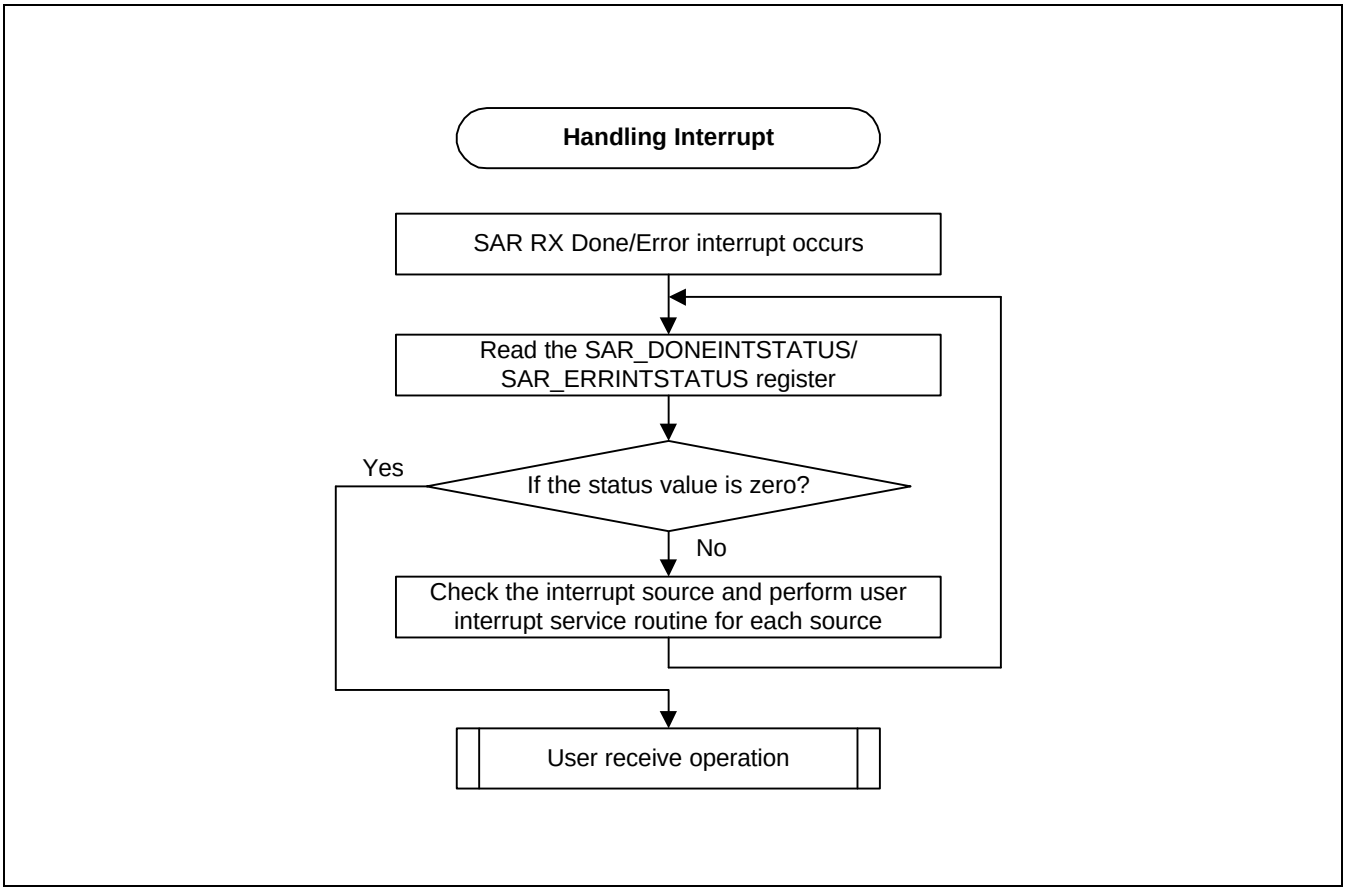


Figure 5-29. SAR Interrupt Service Routine

DIAGNOSTIC CODE : RECEIVE PACKET IN THE INTERRUPT SERVICE ROUTINE

```
//=====================================================================//
// isr_sar_done                                                        //
//                                                                     //
// SAR Done Interrupt service routine.                                 //
//=====================================================================//
void isr_sar_done(void)
{
    volatile ULONG ReadSARstatus = 0;
    ULONG *TxDoneQ, TxIntCnt=0, RxIntCnt0=0, RxIntCnt1=0;
    ULONG *TxDoneAddrValue, PoolPat;
    ULONG *RxDone0AddrValue, *RxDone1AddrValue, *RxDonePkt, *RxPoolArea;

    while(1)   {
        // Checkd SAR Interrupt source.
        ReadSARstatus = nSAR_DONEINTSTATUS;
        if(ReadSARstatus == 0)
            break;

        CntSARintrDone++;

        // Clear SAR Interrupt Status register.
        nSAR_DONEINTSTATUS = ReadSARstatus;

        // Tx Done Interrupt.
        if(ReadSARstatus & INTSTAT_TXDONE)  {

            SARTxDONEOK++;
            SarTxOKFlagCheck=1;

            #if(DebugFlag_SAR_ISR == YES)
            Print("\n [SARisr] Interrupt - Tx Done ");
            #endif

            // Clear SAR address valid bit to release buffer descriptor.
            TxDoneAddrValue = (ULONG *)(nSAR_TXDONEADDR) + CntSARTxDoneintr;
            *TxDoneAddrValue &= ~SAR_ADRVALID;

            CntSARTxDoneintr++;
            if(CntSARTxDoneintr == DFLT_TXDONEQ_SIZE)
                CntSARTxDoneintr = 0;
        }

        // Rx Done Queue Interrupt.
        if(ReadSARstatus & INTSTAT_RXDONEQ0)  {
            CntSARRxDoneQue0_intr++;
            #if(DebugFlag_SAR_ISR == YES)
            //Print("\n [SARisr] Interrupt - Rx Done Que 0");
            #endif
        }

        if(ReadSARstatus & INTSTAT_RXDONEQ1)  {
            CntSARRxDoneQue1_intr++;
            #if(DebugFlag_SAR_ISR == YES)
            //Print("\n [SARisr] Interrupt - Rx Done Que 1");
            #endif
        }
```

```
        // Rx Done Interrupt.
        if(ReadSARstatus & INTSTAT_RXDONE0)  {
            SARRxDONEOK++;

            PoolPat = 0;
            #if(DebugFlag_SAR_ISR == YES)
            Print("\n [SARisr] Interrupt - Rx Done 0");
            #endif
            while(1)  {
                RxDonePkt =(ULONG *)strCFG_SAR_REG.RxDoneQ_0_Addr+ RxDoneQue0_Index;
                if(!(*RxDonePkt & SAR_ADRVALID))
                    break;
                DefaultRxHook((strRxBufDescriptor *)(*RxDonePkt), PoolPat);

                RxDoneQue0_Index++;
                if(RxDoneQue0_Index == DFLT_RXDONEQ_0_SIZE)
                    RxDoneQue0_Index = 0;

                // Clear SAR address valid bit to release buffer descriptor of
                // Rx done queue.
                *RxDonePkt &= ~SAR_ADRVALID;

                // Rx Pool 0 release.
                RxPoolArea=(ULONG*)strCFG_SAR_REG.RxPoolQ_0_Addr + RxPoolQue0_Index;
                *RxPoolArea |= SAR_ADRVALID;
                RxPoolQue0_Index++;
                if(RxPoolQue0_Index == DFLT_RXPOOLQ_0_SIZE)
                    RxPoolQue0_Index = 0;
        }   }


        if(ReadSARstatus & INTSTAT_RXDONE1)  {
            SARRxDONEOK++;

                  PoolPat = 1;
            while(1)  {
                RxDonePkt=(ULONG *)strCFG_SAR_REG.RxDoneQ_1_Addr + RxDoneQue1_Index;
                if(!(*RxDonePkt & SAR_ADRVALID))
                    break;
                DefaultRxHook((strRxBufDescriptor *)(*RxDonePkt), PoolPat);
                RxDoneQue1_Index++;
                if(RxDoneQue1_Index == DFLT_RXDONEQ_1_SIZE)
                    RxDoneQue1_Index = 0;

                // Clear SAR address valid bit to release buffer descriptor of
                // Rx done queue.
                *RxDonePkt &= ~SAR_ADRVALID;

                // Rx Pool 1 release.
                RxPoolArea=(ULONG*)strCFG_SAR_REG.RxPoolQ_1_Addr + RxPoolQue1_Index;
                *RxPoolArea |= SAR_ADRVALID;
                RxPoolQue1_Index++;
                if(RxPoolQue1_Index == DFLT_RXPOOLQ_1_SIZE)
                    RxPoolQue1_Index = 0;
}   }   }   }
```

FLOW CHART OF SAR DIAGNOSIS

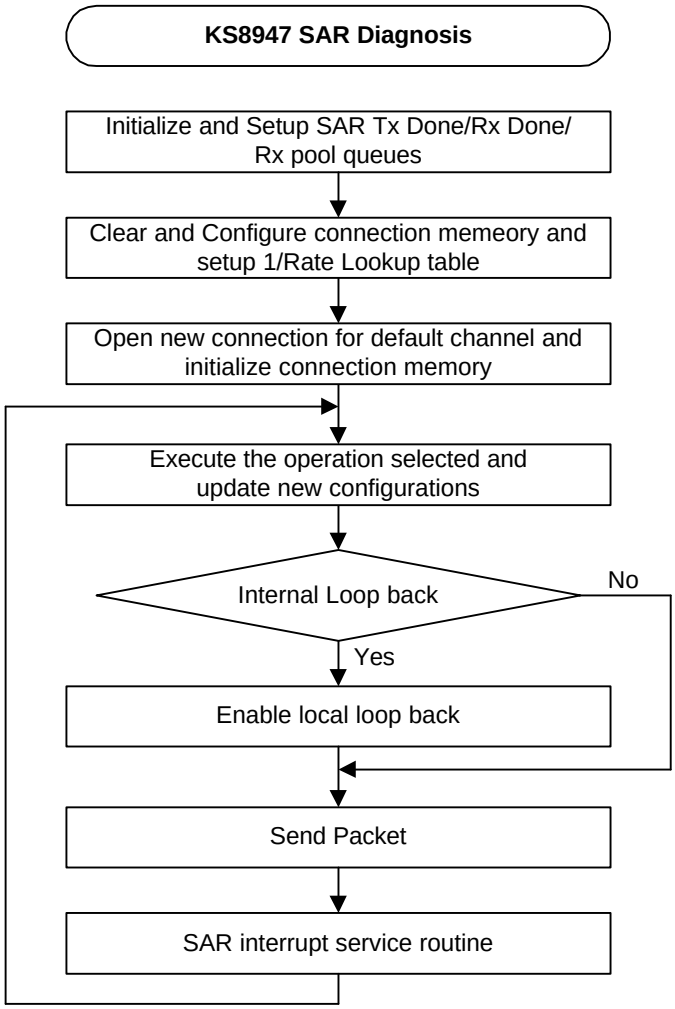


Figure 5-30. SAR Diagnosis Flow

APPENDIX OF SAR

Packet Transmit

1. When SAR used with MAC, MAC and SAR Buffering should be controlled. So, MAC Rx Descriptors must be allocated as many as an amount of the SAR Tx Descriptors. If not, it's possible to be occurred endless packet transmit operation.

   Example)

   #define MAC_Rx_Desc_NUM 1024;

   .

   .

   .

   #define Tx_DoneQ_NUM 1024;

2. When Packet Tx, it can be used with *interrupt mode* or *polling mode*.
   Above S5N8947 SAR S/W guide presents Tx Done interrupt. But we recommand to use *Polling Mode* Tx Done check operation.

   Example)

   # First of all, Don't use Tx Done interrupt service routine on ISR. Then….

// Need MAC Descriptor/Buffer 1024 (when Sar Tx Descriptor is 1024)

ULONG sarTxcount=0;

```
ULONG MacRx2SarTx(ULONG SarFramePtr, ULONG length)
{
      OTHER_PKT_INFO    Info2;
      sMACFrame *pFrame;
      int txIndex, icnt=0, k;
      ULONG currentTxDoneA, nextTxDoneA;
      int returnValue, init;
    // Increment the Done Index and do a modulo
    // to rap around the index to point to the
    // the start of the circular queue, if end of queue
    Aal5Gddb[0].pktmem->TxDoneIdx  = (++Aal5Gddb[0].pktmem->TxDoneIdx) %
TX_POOL_BUFS;
    txIndex = Aal5Gddb[0].pktmem->TxDoneIdx;
    pFrame = (sMACFrame *)SarFramePtr;
    SarFillHead((U8 *)pFrame->SarHeaderAddr, length);
   //=============================================
   // Packet Transmit Polling Mode
   //=============================================
   currentTxDoneA = *((ULONG *)Aal5Gddb[0].Config.TxDoneAddr + sarTxcount);
   if(currentTxDoneA & ADRVALID)
   {
      while(1)
      {     // Tx Done Check & Tx Descriptor Refresh for Next cycle
            *((ULONG *)Aal5Gddb[0].Config.TxDoneAddr + sarTxcount) &= ~ADRVALID;
```

```
            sarTxcount = (++sarTxcount) % TX_POOL_BUFS;     // TX_POOL_BUFS = 1024
            nextTxDoneA = *((ULONG *)Aal5Gddb[0].Config.TxDoneAddr + sarTxcount);
            if(nextTxDoneA & ADRVALID)
                  continue;
            else
                  break;
            } //while inner
      }
    }



    // Sar Tx Over Run Check
    if(txIndex == sarTxcount)
    {
      Print("OverRun");
      return (E_NG);
    }
    Info2.Pti = 0;
    Info2.Clp = 0;
    // Call SarPacketTx function
    returnValue=Aal5PacketTx(0, VPI, VCI, (U8 *)(pFrame->BrgHeaderAddr), &Info2);
    if(returnValue != SUCCESS)
    {
      Print("[Etherrx] Sar Tx Error!\n");
      return      (E_NG);
    }
    return 0;
}
```

SAR Packet Transmit Function : *need to check SAR Tx Linked List Done*

```
int Aal5PacketTx(USHORT DevId, USHORT vpi, USHORT vci, void *PktChain,
OTHER_PKT_INFO* Info)
{
    ......

   // SAR Linked List Done check, not Tx Done Check
    while((rData=Read_SARreg(nSAR_TXREADY1)) & 0x80000000) ;
    // SAR Resgiter Write to send packet.
    Write_SARreg(nSAR_TXREADY1, (ULONG)PktChain | ADRVALID);
    Write_SARreg(nSAR_TXREADY2, (ULONG)LastPktDescPtr | ADRVALID);
}
```

HOW TO DOWNLOAD & EXECUTING USER PROGRAM

S5N8947 Diagnostic ROM program includes “User Program Download” function. This can be found in “Diagnostic ROM Program Main Menu” shown below after booting the evaluation board system. If you have coded a program and want to test it without burning another ROM, you can make your program run on DRAM.

<Main Menu>

```
=========================================================
         CM47-M66-V1.0 Board Diagnostic Ver 1.0
=========================================================
                  [1] Memory TEST
                  [2] UART TEST
                  [3] Timer TEST
                  [4] GDMA TEST
                  [5] I2C BUS TEST
                  [6] I/O Port TEST
                  [7] Ethernet TEST
                  [8] USB Test
                  [S] SAR Test
                  [A] All Test
                  [U] User Program Download
                  [F] FLASH Memory Operation
=========================================================
Select One... :
```

Select *“User Program Download”* with typing 'u' at the cursor and you will see the message below.

```
 ---------------------------------------------
           SYSTEM INFORMATION
 ---------------------------------------------
   ROM0 BASE    : 0x       0
   ROM1 BASE    : 0x  200000
   DRAM BASE    : 0x 1000000
 ---------------------------------------------
  ## Input Download Area Address (default:0x1000050) : 0x
```

Just press Enter. If you want to change the download start address, you have to re-burn your ROM after changing *“ROMOPTS”* in make file.

```
 =========================================================
             DownLoad User's Program to DRAM
=========================================================
                 [x] Using Xmodem
                 [s] Using SFTP
                 [q] Exit
=========================================================
Select One... :
```

[x] Using Xmodem

Select *"Using Xmodem"* with typing '**x**' at the cursor and you will see the message below.

```
$$ Waiting for User Program .....

 * Please Select Menu on your Hyper Terminal
 => Transfer => Send file
 => Browse File => Choose protocol you selected
 => Browse File => Choose File name
```

Click the *"send file"* in pull-down menu of Hyperterminal and you will see the windows below. And then, search the ***"ram.bin(User File)"*** file in your working directory. At last, select the Xmodem protocol.

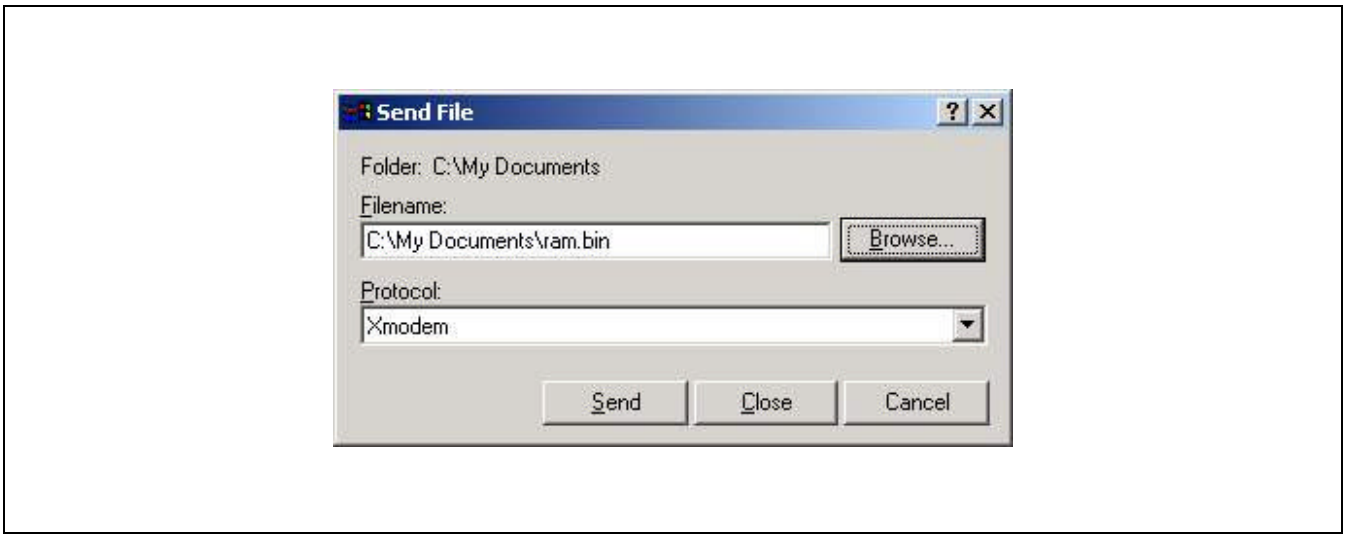


Figure 5-31. Hyperterminal Window Display when Click the Send File in Pull-down Menu

Click “Send” button, and then ***“ram.bin(User File)”*** file send.

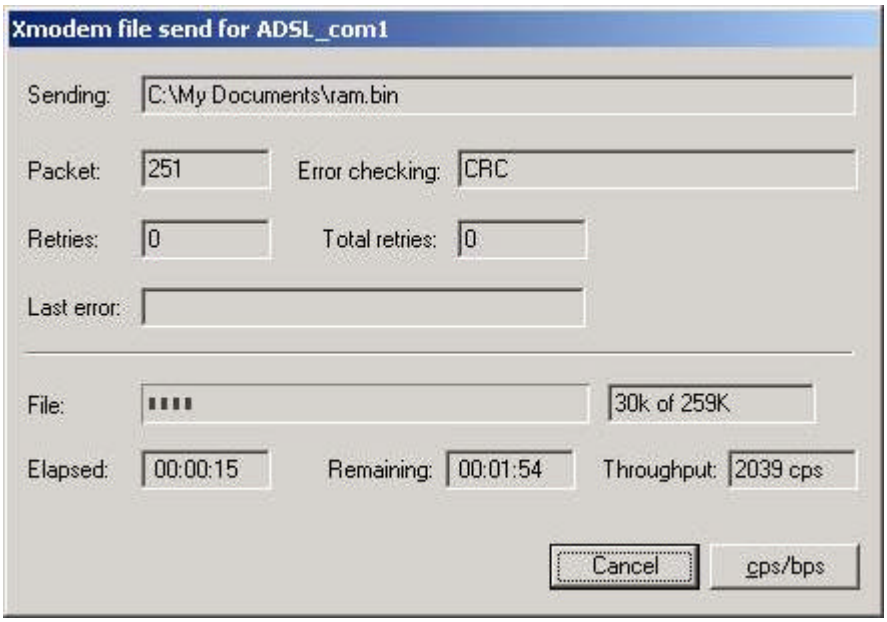


Figure 5-32. Hyperterminal Window Display when Xmodem File Send

After completing downloading and CRC checking following message appears at HyperTerminal window.

```
$$ Waiting for User Program .....
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC OK !
$$ CRC Check Ok ..

------------------------------------------------
          Start  User's  Program
------------------------------------------------
   [s]  Start Program
   [q]  Exit
------------------------------------------------
Select Test Item : s
$$ Now, User program will be started.
```

Press **‘s’** key and your program will start running.

[s] Using SFTP

To download user's program, you also need a DOS application offered by us. This program is called 'sftp' with file name "SFTP.EXE".

Open a DOS window and run Sftp as a format below.

sftp < com port # > < file name >

Ex) sftp 1 ram.bin

Following message asks you if you want to change downloading baud rate. Just press **'n'.**

The downloading procedure starts and the progress shown with '#' mark.

After completing downloading and CRC checking following message appears at HyperTerminal window.

```
$$ Waiting for User Program ..... Ok.
$$ CRC Check Ok ..

------------------------------------------------
          Start  User's  Program
------------------------------------------------
   [s]  Start Program
   [q]  Exit
------------------------------------------------
Select Test Item : s
$$ Now, User program will be started.
```

Press **'s'** key and your program will start running.

SERIAL PERIPHERAL INTERFACE

The S5N8947 provides a Serial Peripheral Interface (SPI), which is used for register access of other devices, EEPROM and A/D converter. S5N8947 dedicate pin used for receive serial data (SPIMISO), transmit serial data (SPIMOSI) and clock (SPICLK).

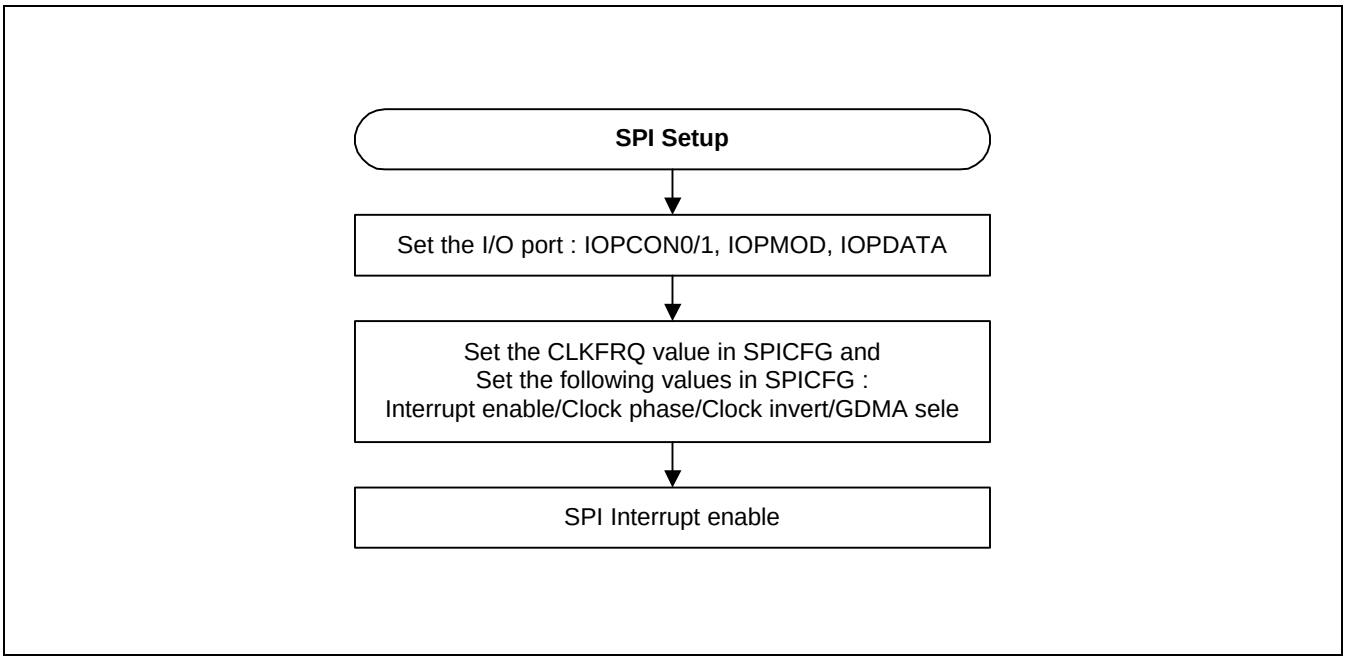


Figure 5-33. Concept Diagram for Setting Up GDMA

EXAMPLE CODE FOR SPI SETUP FUNCTION ROUTINE <spi.c>

```
//=====================================================================//
// SPISetup()                                                          //
//                                                                     //
//  SPI Setup Routine. Initialize SPI control block to use SPI EEPROM  //
//=====================================================================//
void SPISetup(void)
{
          UINT len;
          UINT ret;

          //disable Timer0 interrupt
          Disable_Intr(nTIMER0_INT);

          //I/O Port setting
          IOPCON0 |=(PORT0_SPI_SPICLK_OUTPUT|PORT1_SPI_SPIMOSI_OUTPUT
|PORT2_SPI_SPIMISO_INPUT);
          IOPCON1 &=~CS_port_mode;
          IOPMOD |= CS_port;
          IOPDATA |=CS_port;

          // SPI is disable
          gSPICFG &=~SPI_ENABLE;
          SPICFG = gSPICFG;

          // Set CLKFRQ Value: fSCL is SPI serial clock frequency
          gSPICFG &=0xFFFFFF00;
          SPICFG = gSPICFG;

          gSPICFG |= (ret=SetClkfrq((int)fSPICLK) & 0xff);  //support upto 5MHz
          SPICFG = gSPICFG;

          len=(0x07<<8);
          gSPICFG |= (len & (0x1f00));
          SPICFG = gSPICFG;

          gSPICFG &= ~(EN_SPI_INT|NORMAL_ORDER|TOGGLING_AT_THE_BEGIN|USE_GDMA);
          gSPICFG |= NORMAL_ORDER;
          SPICFG = gSPICFG;

          // Enable SPI Interrupt
          SysSetInterrupt(nSPI_INT,SPIisr) ;
          Enable_Intr(nSPI_INT) ;
}
```

EXAMPLE CODE FOR SPI WRITE FUNCTION ROUTINE <spi.c>

```
//=====================================================================//
// SPIWrite()                                                          //
//                                                                     //
//  The CAT24WC32/64 writes up to 32 bytes of data, in a single write  //
//  cycle, using the page write operation.                             //
//=====================================================================//
void SPIWrite(UINT WriteAddr, char WriteData[], UINT SizeOfData)
{
    SPIPageWrite(WriteAddr,WriteData,SizeOfData);
}

void SPIPageWrite(UINT WriteAddr,char WriteData[],UINT SizeOfData)
{
          UINT WriteDataCnt ;
          UINT ByteAddrMsb;   /* Bytes Address : A15-A8  */
          UINT ByteAddrLsb;   /* Bytes Address : A7-A0  */
          U32 TEMP;

          ByteAddrMsb = ((WriteAddr>>8) & 0xff);
          ByteAddrLsb = (WriteAddr & 0xff);

          // Step1. Setup SPICFG Register
          gSPICFG |= EN_SPI_INT;
          SPICFG = gSPICFG;

          gSPICFG |= SPI_ENABLE;
          SPICFG = gSPICFG;

          // Step2. Send write enable command.
          IOPDATA &= ~CS_port;
          TXCHR = WREN;

          while(SPICMD & START_TRANS);
          SPICMD |= START_TRANS;

          while( !SPIDoneFlag ) ;
          SPIDoneFlag = 0 ;

          IOPDATA |= CS_port;

          // Step3-1. Send write command.
          IOPDATA &= ~CS_port;
          TXCHR = WRITE;

          while(SPICMD & START_TRANS);
          SPICMD |= START_TRANS;

          while( !SPIDoneFlag ) ;
          SPIDoneFlag = 0 ;
```

```
        // Step3-2. Send Bytes Address : A15-A8
        TXCHR = (ByteAddrMsb<<24);

        while(SPICMD & START_TRANS);
        SPICMD |= START_TRANS;

        while( !SPIDoneFlag ) ;
        SPIDoneFlag = 0 ;

        // Step3-3. Send Bytes Address : A7-A0
        TXCHR = (ByteAddrLsb<<24);

        while(SPICMD & START_TRANS);
        SPICMD |= START_TRANS;

        while( !SPIDoneFlag ) ;
        SPIDoneFlag = 0 ;

        // Step3-4. Send Multiple Data
        for(WriteDataCnt=0;WriteDataCnt<SizeOfData;WriteDataCnt++)
        {
                TEMP= WriteData[WriteDataCnt] ;

                TXCHR = (TEMP<<24);

                while(SPICMD & START_TRANS);
                SPICMD |= START_TRANS;

                while( !SPIDoneFlag ) ;
                SPIDoneFlag = 0 ;
        }
        IOPDATA |= CS_port;

        // Step4. Send write disable command.
        IOPDATA &= ~CS_port;
        TXCHR = WRDI;

        while(SPICMD & START_TRANS);
        SPICMD |= START_TRANS;

        while( !SPIDoneFlag ) ;
        SPIDoneFlag = 0 ;

        IOPDATA |= CS_port;

        // Step5. Disable SPI
        gSPICFG &= ~SPI_ENABLE;
        SPICFG = gSPICFG;

        gSPICFG &= ~EN_SPI_INT;
        SPICFG = gSPICFG;
}
```

EXAMPLE CODE FOR SPI READ FUNCTION ROUTINE <spi.c>

```
//=====================================================================//
// SPIRead()                                                           //
//                                                                     //
//  The Sequential READ operation can be initiated by either           //
//  the Immediate Address READ or Selective READ operation             //
//=====================================================================//
void SPIRead(UINT ReadAddr, UINT ReadDataSize, char ReadValue[])
{
          SPISeqRead(ReadAddr, ReadValue, ReadDataSize);
}

void SPISeqRead(UINT ReadAddr,char ReadValue[],UINT ReadDataSize)
{
          UINT ReadCnt ;
          UINT ByteAddrMsb;   /* Bytes Address : A15-A8  */
          UINT ByteAddrLsb;   /* Bytes Address : A7-A0  */
          UINT wait;
          U32 TEMP;

          ByteAddrMsb = ((ReadAddr>>8) & 0xff);
          ByteAddrLsb = (ReadAddr & 0xff);

          // Step1. Setup SPICFG Register
          gSPICFG |= EN_SPI_INT;
          SPICFG = gSPICFG;
          gSPICFG |= SPI_ENABLE;
          SPICFG = gSPICFG;

          // Step2. Send write enable command.
          IOPDATA &= ~CS_port;
                    TXCHR = WREN;

                    while(SPICMD & START_TRANS);
                    SPICMD |= START_TRANS;

                    while( !SPIDoneFlag ) ;
                    SPIDoneFlag = 0 ;
          IOPDATA |= CS_port;

          // Step3-1. Send Read command.
          IOPDATA &= ~CS_port;
                    TXCHR = READ;

                    while(SPICMD & START_TRANS);
                    SPICMD |= START_TRANS;

                    while( !SPIDoneFlag ) ;
                    SPIDoneFlag = 0 ;

                    // Step3-2. Send Bytes Address : A15-A8
                    TXCHR = (ByteAddrMsb<<24);

                    while(SPICMD & START_TRANS);
                    SPICMD |= START_TRANS;

                    while( !SPIDoneFlag ) ;
```

```
                    SPIDoneFlag = 0 ;

                    // Step3-3. Send Bytes Address : A7-A0
                    TXCHR = (ByteAddrLsb<<24);

                    while(SPICMD & START_TRANS);
                    SPICMD |= START_TRANS;

                    while( !SPIDoneFlag ) ;
                    SPIDoneFlag = 0 ;

                    //Step3-4. Receive Multiple data
                    for (ReadCnt=0 ; ReadCnt < ReadDataSize ; ReadCnt++) {

                                        while(SPICMD & START_TRANS);
                                        SPICMD |= START_TRANS;

                                        while( !SPIDoneFlag ) ;
                                        SPIDoneFlag = 0 ;

                                        TEMP = RXCHR;
                                        ReadValue[ReadCnt] = (char)(TEMP>>24);
                    }

          IOPDATA |= CS_port;

          // Step4. Send write disable command.
          IOPDATA &= ~CS_port;
                    TXCHR = WRDI;

                    while(SPICMD & START_TRANS);
                    SPICMD |= START_TRANS;

                    while( !SPIDoneFlag ) ;
                    SPIDoneFlag = 0 ;
          IOPDATA |= CS_port;

          // Step5. Disable SPI
          gSPICFG &= ~SPI_ENABLE;
          SPICFG = gSPICFG;

          gSPICFG &= ~EN_SPI_INT;
          SPICFG = gSPICFG;
}
```

PCMCIA INTERFACE

The S5N8947's System Manager provides the control logic for a PCMCIA socket interface, and requires only additional external analog power switching logic and buffering.
The control signals for PCMCIA are generated through external bus and general I/O ports by configuring GPIO registers. Additional address lines (ADDR[25:22]) for PCMCIA and analog power control signals are made by using one of the memory controller chip-select pins (ex.PnRCS, PnECS).
The PCMCIA controller provides common memory, attribute memory and I/O function region by PRS bits (PCMCON[31:30]). The PCMCIA interface can be configured as big-endian or little-endian by PCMEND bit (PCMCON[29]) regardless of external big/little selection pin (BIGEND, 16). When IOIS16# is not used by PCMCIA card in I/O function region, PCMCIA I/O function port size is depend on PPS bit (PCMCON[28]). The user can control PCMCIA access cycle by configuring PCMCON register (PAST, PSST, PSL, PSHT).
Figure 5-26 shows the PCMCIA socket interface guide. The socket and external bus must be electrically isolated using external buffers and bus transceivers. These buffers also provide voltage conversion required from the 3.3V to 5V cards.

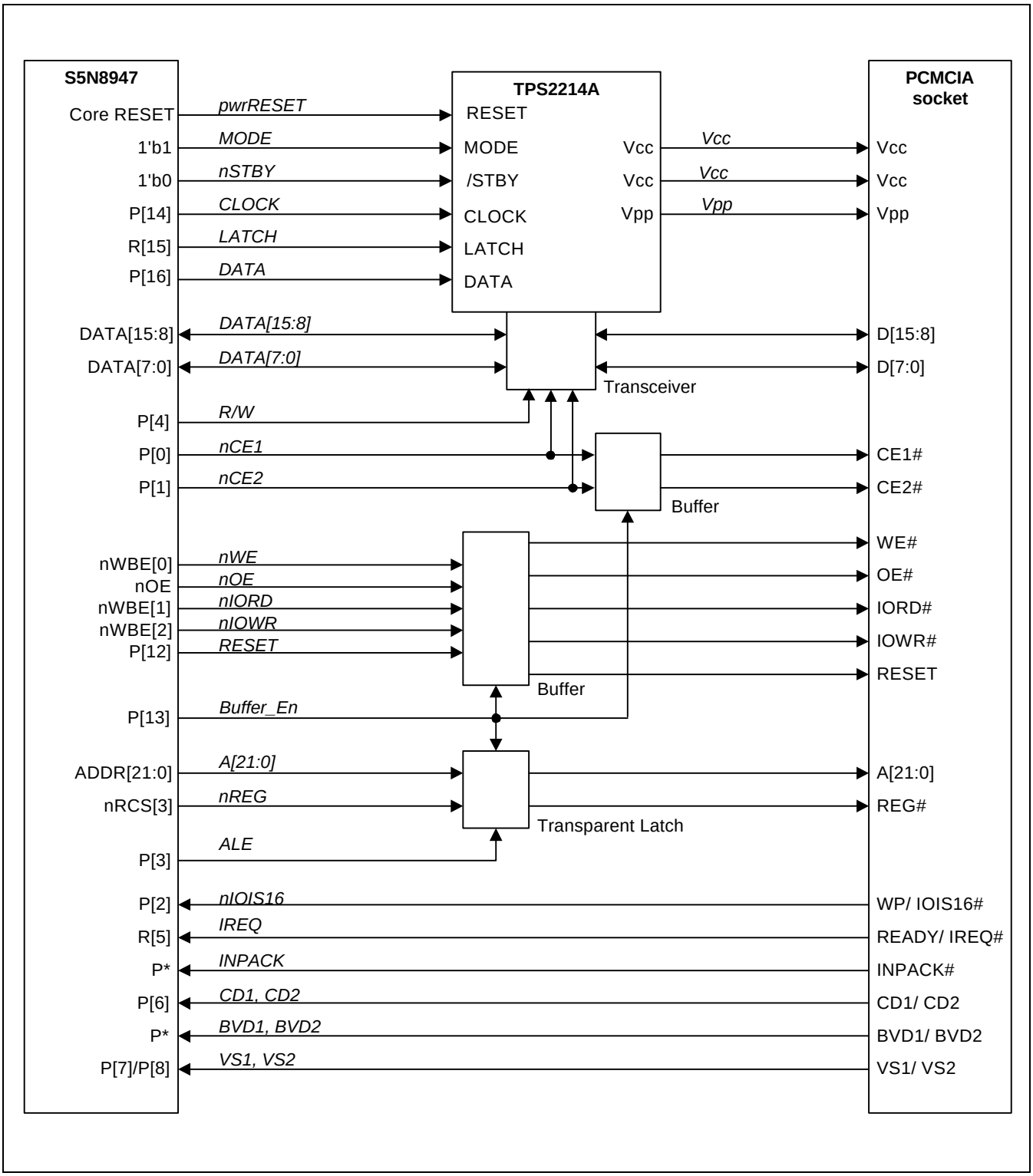


Figure 5-34. PCMCIA Socket Interface

CONTROLLING THE POWER SWITCH TPS2214A

PCMCIA card operates at either 5V or 3.3V. So power switching is required according to the card voltage.

In the reference board, we use the TPS2214A power switching chip to provide the proper power. To control this device, three control signals are used – clock, data and latch. In Figure 1, GPIO signals are used to control the power switching device. But in the reference board, we use data lines instead of GPIOs. Used for the other devices attached to S5N8947, 18 GPIO signals are not many in number.

We assign the data[6:4] to latch, data and clock each

INITIALIZE PCMCIA INTERFACE

For the PCMCIA, we have to configure the I/O port register. I/O port register settings are depicted in I/O ports chapter.

S5N8947 has no PCMCIA host core, but has an interface for PCMCIA. We assign 2 interrupt sources for PCMCIA operation - One for card detection and the other for card interrupt generated by targer card. I/O port 5 and 6 are used for that purpose. These pins can be used as external interrupts.

The I/O ports and interrupts initialization are described below.

```
void PCMCIA_init(void)
{
      // PCMCIA control register setting
      //PCMCON = 0x8007ffff;              // PCMCIA little endian
      PCMCON = 0x80011021;
      //PCMCON = 0xa007ffff;              // PCMCIA big endian
      // I/O port settings
      IOPMOD |= 0x0001f000;         // port 12, 13, 14, 15, 16 as output
      IOPMOD &= 0xfffffe1f;         // port 5,6,7,8 as input
      IOPCON0 =
      PORT0_PCMCIA_NCE1_OUTPUT|PORT1_PCMCIA_NCE2_OUTPUT|PORT2_PCMCIA_NIOIS16_INPUT|P
      ORT3_PCMCIA_ALE_OUTPUT|PORT4_PCMCIA_PCM_RW_OUTPUT;    // I/O port settings for
      PCMCIA
      IOPCON0 |= ((XIRQ_ENABLE|ACTIVE_LOW|FILTERING_OFF|FALLING_EDGE) << 8);
            // xIRQ0 settings for PCMCIA (port5 : PCMCIA card interrupt)
      IOPCON0 |= ((XIRQ_ENABLE|ACTIVE_LOW|FILTERING_OFF|FALLING_EDGE) << 13);
      // xIRQ1 settings for PCMCIA (port6 : card detection interrupt)

      IOPDATA |= PCMCIA_BUFFER_ENABLE;               // buffer disable
      IOPDATA &= ~PCMCIA_RESET;                      // reset signal set LOW -
      PCMCIA reset polarity : active high

      // Interrupt settings
      SysSetInterrupt(nEXT1_INT,  isr_PCMCIA_Card_Detect);  // Ext1 interrupt is
      used for PCMCIA card detection
      SysSetInterrupt(nEXT0_INT, isr_PCMCIA_Card_Int);      // Ext0 interrupt is
      used for PCMCIA card interrupt
      Enable_Intr(nEXT1_INT);
      Enable_Intr(nEXT0_INT);
}
```

INTERRUPT SERVICE ROUTINE FOR CARD DETECTION

As described above, S5N8947 has no PCMCIA host core. When the card is inserted, the card detection interrupt is generated. In the service routine, S5N8947 perform the 3 operation. First, the power switching device, TPS2214A, is initialized. Then S5N8947 reads the voltage sensing pins. These two sense pins are representing the supply voltage. After determining the supply voltage, S5N8947 writes the 11 bit command to TPS2214A. Finally, the MCU sends the reset signal to the card. The basic interrupt service routine is depiced below.

```
void isr_PCMCIA_Card_Detect(void)
{
      U16 vs;
      U32 wait;
      card_detection = 0;
if((INTPEND>>1)&0x00000001)
            Clear_PendingBit(nEXT1_INT);
      for(wait=0;wait<500000;wait++);           // delay for PCMCIA card insertion
      initTPS2214A();
      // voltage sensing with VS1, VS2
      vs = (IOPDATA & 0x0180);
      if((vs >>7) == 0x03)          // 5V operation
      {
            // configuring TPS2214A for 5V operation
            //wrTPS2214A(VCC_5V);         // VCC = 5V , VPP = 5V
      }
      else if((vs >>7) == 0x01)     // 3.3V operation
      {
            // configuring TPS2214A for 3.3V operation
            wrTPS2214A(VCC_3V);           // VCC = 3.3V , VPP = 0V
      }
      else
      {
            card_detection = 1;
            Print("Voltage Detection Fail!!!");
            return;
      }
      for(wait=0;wait<10000;wait++);            // delay for PCMCIA card reset
      // card reset
      PCMCIA_Card_Reset();
      Print("Detection End");

}
```